LiquidHaskell: Liquid Types for Haskell

Niki Vazou
nvazou@cs.ucsd.edu
University of California, San Diego

Abstract

Even well-typed programs can go wrong, by returning a wrong answer or throwing a run-time error. A popular response is to allow programmers use *refinement type systems* to express semantic specifications about programs. We study verification in such systems. On the one hand, expressive refinement type systems require run-time checks or explicit proofs to verify specifications. On the other, less expressive type systems allow static and automatic proofs of the specifications. Next, we present abstract refinement types, a means to enhance the expressiveness of a refinement type system without increasing its complexity. Then, we present LIQUIDHASKELL that combines liquidTypes with abstraction over refinements to enhance expressiveness of LiquidTypes. LIQUIDHASKELL is a quite expressive verification tool for Haskell programs that can be used to check termination, totality and general functional correctness. Finally, we evaluate LIQUIDHASKELL in real world Haskell libraries.

1 Introduction

Functional programming languages, like ML and Haskell, come with strong static type systems, which detect a lot of errors at compile-time and enhance code documentation.

The usefulness of these type systems stems from their ability to predict, at compile-time, invariants about the run-time values computed by the program. Unfortunately, traditional type systems only capture relatively coarse invariants. For example, the system can express the fact that a variable `i` is of type `Int`, meaning that it is always an integer, but not that it is always an integer with a certain property, say different than zero. Thus, the type system is unable to statically ensure the safety of critical operations, such as division by `i`. Several authors have proposed the use of refinement types [**?**, **?**, **?**, **?**] as a mechanism for enhancing the expressiveness of type systems.

Refinement types refine a vanilla type with a predicate. For example, one can give `i` the following type:

```
i :: {v:Int | v != 0}
```

that describes a value `v` of type `Int`, while the refinement constraints this value `v` to be different than `0`.

One can use this refinement type to define a safe division operator:

```
safeDiv :: Int -> {v:Int | v != 0} -> Int
```

This type captures that the division operator takes two `Int` arguments and returns an `Int`. Moreover, it restricts the second argument to be different that zero, to eliminate division by zero operations.

At the call site of `safeDiv` the type system should check that the real arguments do not violate its specification. For instance, `safeDiv 8 9` is safe, since `9` is always

different that zero, but `safeDiv 8 0` should raise a type error. Apart from concrete values, `safeDiv` can be applied to arbitrary program expressions: `safeDiv n m` is safe only if `m` is an integer different than zero.

Refinement Function Types. *Refinement function types* [**?**, **?**] allow the specification of the result to depend on the argument. A parameter is used to bind the argument and can appear in the refinement of the result. As an example, we define a `pred` function, which takes as argument a positive integer `n` and returns its predecessor. Refinement function types allow us to give `pred` a type that exactly captures this behaviour:

```
pred :: n : {v:Int | v > 0} -> {v:Int | v = n-1}
pred n = n-1
```

This type denotes that for each positive integer argument `n`, the result is an `Int` exactly equal to `n-1`. When `pred` is applied to a concrete value, the parameter `n` is substituted with this value. For example

```
pred 2 :: {v:Int | v = n-1}[2/n] = {v:Int | v = 1}
```

Thus, for each concrete argument, the result should be the predecessor of this argument.

Specifications. A *specification* [**?**] is the expression in some formal language and at some level of abstraction of a collection of properties some program should satisfy. Specifications can be expressed in various techniques. For example, temporal logic can be used to express history-based specifications, e.g., to reason about program's behaviour over time, while monitors can be used to express state-based specifications, e.g., reasoning about concurrency. In this paper we use refinement type signatures to describe (pure) functional specifications, i.e., the program is specified as a collection of mathematical functions. This way, we limit ourselves to ignore features such as temporal constrains, imperative features and concurrency.

Verification. *Verification* is a procedure that takes as input a program, i.e., definitions for functions and values, and some specifications, i.e., refinement type signatures for functions and values, and decides whether the specifications hold for the program. Informally, it checks that each expression satisfies its type, for example, that the `pred` definition actually returns the predecessor of its argument, or that at each function application the arguments satisfy the function's preconditions, as in the `safeDiv` example. If the specifications hold, the program is *Safe*, otherwise it is *Unsafe*.

Higher-order programming languages, such as ML or Haskell, treat functions as first order objects. Thus, one can use functions in refinements and create higher-order predicates. For instance, the following type

```
f:(a->b) -> {v:Bool|terminates f}
```

describes that an arbitrary functional argument should satisfy a predicate `terminates`. Reasoning in a higher-order logic is undecidable, thus if arbitrary program values appear in the refinements, the verification procedure is undecidable. As we shall see, if the refinement language is restricted, i.e., is less expressive, verification can be decidable.

The rest of this paper is organized as follows: In § **??** we present a core calculus that constitutes the base for many refinement type systems. In § **??** we describe reasoning in undecidable refinement type systems. In § **??** we present less expressive type systems (like LiquidTypes) that are decidable. In § **??** we present how abstraction over refinements enhances expressiveness of decidable type systems. Then § **??** we present LIQUIDHASKELL that combines liquidTypes with abstraction over refinements to enhance

Expressions	e	::=	$x \mid c \mid \lambda x:\tau.e \mid e\ e$
Predicates	p	::=	$\ldots$
Basic Types	b	::=	`int` $\mid$ `bool`
Refinement Types	τ	::=	$\{v:b \mid p\} \mid x:\tau \rightarrow \tau$
Typing Environment	Γ	::=	$\emptyset \mid x:\tau, \Gamma$

Figure 1: **Syntax of λ_C**

expressiveness of LiquidTypes. LIQUIDHASKELL is a quite expressive verification tool for Haskell programs that can be used to check termination (§ **??**), totality (§ **??**) and general functional correctness (§ **??**). Finally, we evaluate LIQUIDHASKELL and conclude.

2 Preliminaries

To formally describe and compare type systems, we define a core calculus, following [**?**, **?**, **?**]. We refer to our calculus as λ_C, and in this section we present its syntax and type system.

2.1 Syntax

The syntax of expressions and types is summarized in Figure **??**. λ_C expressions include variables, constants, typed λ-abstractions and function applications. Constants include primitive integers: $0, 1, 2, \ldots$ and primitive booleans: `true` or `false`, which take the basic types, integer and boolean, respectively. A basic type can be refined with a predicate to construct a basic refinement type. Refinement types also contain function types, in which a variable binds the argument, so that the result refinement can refer to it. The predicate p is not yet defined. As noted earlier, if p contains arbitrary program expressions, the type system is undecidable, but p can be restricted in such a way as to render the type system decidable. Finally, we define a typing environment Γ that maps variables to their type, and will be used in the typing rules that we will discuss.

2.2 Typing

The typing rules used by λ_C are summarized in Figure **??**.

Type Checking. Type checking rules state that an expression e has a type τ under an environment Γ, that is, when the free variables in e are bound to values described by Γ, the expression e will evaluate to a value described by τ. We write $\Gamma \vdash e : \tau$ and create one rule for each program expression.

The rule T-CONST uses a function tc that maps each primitive constant to its predefined type. The rule T-VAR checks the type of a variable, according to the environment Γ. The rule T-FUN checks the type of the function-body in the environment, extended with the argument of the function. Since the argument type is given, it could be any arbitrary type, say $\{v : b \mid 1\}$, which is invalid, as a base type is refined with the value 1, which cannot be a valid predicate. A well-formedness rule is used to check that the argument type is *well-formed*, i.e., its refinements are valid predicate expressions.

Finally, the rule T-APP checks that in an application $e_1\ e_2$ the expression e_1 has a function type whose argument is the type of the argument e_2. As we discussed in the introduction, in the final type, the parameter x should be replaced with the actual argument e_2.

Well-formedness Rules. Well-formedness rules state that a type τ is well-formed under an environment Γ, that is, the refinements in τ are boolean expressions in the environment Γ. We write $\Gamma \vdash \tau$ and create one rule for each type.

The rule WF-BASE checks that in a basic refinement type, the refinement is a valid boolean expression. The environment of this check is extended with the value that is refined; for instance, to check the validity of $\{v : \texttt{int} \mid v > 0\}$, we check that $v > 0$ is a boolean expression, in an environment where v is an integer value. The rule WF-FUN recursively applies the well-formedness rule to the argument type of the function, and to the result type, in an environment extended with the argument parameter.

Subtyping Rules. Consider that the predefined type for the integer 2 is an integer that is exactly equal to 2. The type system can check, via the rule T-CONST, that $\emptyset \vdash 2 : \{v : \texttt{int} \mid v = 2\}$. If 2 is applied to a function that expects a positive integer, say $f :: x : \{v : \texttt{int} \mid v > 0\} \rightarrow \tau$, the type system should also check that $\emptyset \vdash 2 : \{v : \texttt{int} \mid v > 0\}$. There are many ways for this check to succeed. We follow *syntactic subtyping*, in which subtyping reduces to implication checking. In our example, $v = 2 \Rightarrow v > 0$ implies $\{v : \texttt{int} \mid v = 2\} \preceq \{v : \texttt{int} \mid v > 0\}$.

In the general case, subtyping rules state that the type τ_1 is a subtype of the type τ_2 under environment Γ, i.e., when the free variables of τ_1 and τ_2 are bound to values described by Γ, the set of values described by τ_1 is contained in the set of values described by τ_2. We write $\Gamma \vdash \tau_1 \preceq \tau_2$ and create one rule for every type.

The rule $\prec$-BASE serves two purposes. Firstly, it checks that the basic type is the same in the two types. Secondly, it checks that under the environment Γ, the left hand side refinement implies the right hand side. The implication checking is enforced by a predicate `Valid` which varies between the systems that we will describe. The rule $\prec$-FUN relates two function types according to the contravariant rule.

In the rest of this paper, we will use the core calculus λ_C upon which we will build a subset of three typing systems [**?**, **?**, **?**].

3 Undecidable Systems

In systems where the refinement language can have arbitrary program expressions, higher order predicates can be expressed, thus the verification procedure is undecidable. There are many alternatives to reason in such languages; in this section we will present two of them. Firstly, we present *Interactive theorem proving*, where the proofs are statically provided by the user. Secondly, we present *Contracts Calculi*, where the specifications are checked at run time.

3.1 Interactive theorem Proving

One approach to verify that a program satisfies some specifications is for the user to statically prove them. This approach is used by *interactive theorem provers*, such as NuPRL [**?**], Coq [**?**], F* [**?**], Agda [**?**] and Isabelle [**?**] that can express mathematical assertions, mechanically check proofs of these assertions, help to find formal proofs, and extract a certified program from the constructive proof of its formal specification.

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash c : tc\,(c)}\ \textsc{T-Const} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$$

$$\frac{\Gamma, x : \tau_x \vdash e : \tau \qquad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x : \tau_x . e : (x : \tau_x \to \tau)}\ \textsc{T-Fun} \qquad \frac{\Gamma \vdash e_1 : x : \tau_x \to \tau \qquad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau[e_2/x]}\ \textsc{T-App}$$

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

$$\frac{\Gamma, v : b \vdash p : \texttt{bool}}{\Gamma \vdash \{v : b \mid p\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x : \tau_x \to \tau}\ \textsc{WF-Fun}$$

Subtyping $\boxed{\Gamma \vdash \tau \preceq \tau}$

$$\frac{(\Gamma, v : b \vdash \texttt{Valid}(p_1 \Rightarrow p_2))}{\Gamma \vdash \{v : b \mid p_1\} \preceq \{v : b \mid p_2\}}\ \prec\textsc{-Base}$$

$$\frac{\Gamma \vdash \tau_{21} \preceq \tau_{11} \qquad \Gamma, x_2 : \tau_{21} \vdash \tau_{12}[x_2/x_1] \preceq \tau_{22}}{\Gamma \vdash x_1 : \tau_{11} \to \tau_{12} \preceq x_2 : \tau_{21} \to \tau_{22}}\ \prec\textsc{-Fun}$$

Figure 2: **Static Semantics for** λ_C

As an example, consider once again the `pred` function that computes the predecessor of a positive number.

```
pred :: s:{n : nat | n > 0}  -> {v:nat | s = S v}
```

This type signature says that if `pred` is called with a positive number `s`, it will return `s`'s predecessors. There are two different assertions that should be proved:

- The result of the function is the predecessor of the argument. At `pred`'s definitions the programmer should provide a proof that this assertion is indeed satisfied.
- The argument is a positive number. At each call site of `pred`, the user should provide a proof that its argument is positive.

To illustrate programming with refinement types in Coq, we define the `pred` function, as presented in Figure **??**, following [**?**]. In Coq the refinement type is defined in the standard library, and is syntactic sugar, for the type family `sig`. The function `pred` takes an argument `s` which has two components: a natural number `n` and a proof `pf` that this number is positive. There is a case analysis on `s`: if `n` is zero, then the proof `pf` is used to construct a proof that zero is greater that zero (using a lemma `zgtz`) and thus reach a contradiction; so this case can not occur. Otherwise, `n` has a predecessor, say `n'` and the function returns `n'` combined with a proof that its successor is equal to `n`. This proof is constructed by applying `eq_refl`, the only constructor of equality to `S n'`.

In the call site of `pred`, the programmer should provide both the argument and a proof that it is positive. As an example, if `two_gt0` is a proof that two is greater than zero, we can have

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P
Notation
  "{ x : A | P }" := sig (fun x : A => P)

Definition pred (s : {n : nat | n > 0}) : {m : nat |
    proj1_sig s = S m} :=
  match s return {m : nat | proj1_sig s = S m} with
    | exist 0 pf => match zgtz pf with end
    | exist (S n') pf => exist _ n' (eq_refl (S n'))
  end.
```

Figure 3: The pred function in Coq

```
pred (exist _ 2 two_gt0)
```

This application typechecks, as Coq verifies that the argument satisfies `pred`'s precondition, i.e., that `two_gt0` is indeed a proof that `2` is greater than 0.

Even though this example seems tedious, interactive theorem proving can be simplified using inference and tactics. However, the user still needs to provide proofs. We discuss other systems, which remove this burden from the user.

3.2 Contracts

Another approach to verify that a program satisfies some assertions is to dynamically check them. These assertions are called *contracts*, i.e., dynamically enforced pre- and post-assertions that define formal, precise, and verifiable interface specifications for software components. Their use in programming languages dates back to the 1970s, when Eiffel [**?**], an object-oriented programming language, totally adopted assertions and developed the "Design by Contract" philosophy [**?**].

Contracts are of the form: $\langle\{\mathtt{v}:\tau \mid p\}\rangle^l$. The refinement part, as usual, describes the values v, of type τ that satisfy the predicate p. The l superscript is a *blame label*, used to identify the source of failures. As an example, consider a contract for positive integers $\langle\{\mathtt{v}:\mathtt{Int} \mid v > 0\}\rangle^l$ applied to two values, 2 and 0:

$$\langle\{\mathtt{v}:\mathtt{Int} \mid v > 0\}\rangle^{l'}\ 2 \to 2$$
$$\langle\{\mathtt{v}:\mathtt{Int} \mid v > 0\}\rangle^{l}\ 0 \to \Uparrow l$$

If the check succeeds, as in the case for 2, then the application will return the value, so the first application just returns 2. If it fails, then the entire program will "blame" the label l, raising an uncatchable exception $\Uparrow l$, pronounced "blame l".

In 2002, Findler and Felleisen [**?**] were the first to create a system for higher order languages with contracts. In their system, the blame is properly assigned in the higher-order components of the program via a "variance-contravariance" rule. Moreover, they allow dependent contracts, i.e., contracts that have the form of a refinement function type, where the result can depend on the argument. Finally, they treat contracts as first class values, i.e., contracts are values that can be passed to and from functions. In

2004, Blume and McAllester [**?**] formally defined contract satisfaction on Findler and Felleisen's system, and they proved that their system is indeed sound and complete. The contract system is sound if whenever the algorithm blames a contract declaration, that contract declaration is actually wrong. Conversely, it is complete if blame on an expression is explained by the fact that the expression violates one of it contract interfaces.

Since Findler and Felleisen's work a variety of contract calculi systems have been studied. Broadly, these come in two different sorts. In systems with *latent contracts*, types and contracts are orthogonal features. Examples of these systems include Findler and Felleisen's original system, Hinze et al. [**?**], Blume and McAllester [**?**], Chitil and Huch [**?**], Guha et al. [**?**], and Tobin-Hochstadt and Felleisen [**?**]. By contrast, *manifest contracts* are integrated into the type system, which tracks, for each value, the most recently checked contract. Hybrid types [**?**] are a well-known example in this style; others include the work of Ou et al. [**?**], Wadler and Findler [**?**], Gronski et al. [**?**], Belo et al. [**?**], and Grennberg et al. [**?**]. In the rest of this subsection we discuss manifest contracts and present a core calculus for them.

3.2.1 Manifest Contracts

Manifest Contracts Systems [**?**], use casts $\langle \tau_s \Rightarrow \tau_t \rangle^l$ to convert values from the source type τ_s to the target type τ_t and raise $\Uparrow l$ if the cast fails.

As an example, consider a cast from integers to positives:

$$\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v > 0}\} \rangle^l \texttt{ n}$$

The system should statically verify that the value `n` is of the source type `Int`. After the cast, this value is treated as if it has the target type $\{\texttt{v:Int} \mid \texttt{v > 0}\}$. At run-time, a check will be made that `n` is actually a positive integer and if it fails it will raise $\Uparrow l$.

To generalize, for base contracts a cast will behave just like a check on the target type: applied to `n`, the cast either returns `n` or raises $\Uparrow l$. A function application cast $(\langle \tau_{11} \texttt{ -> } \tau_{12} \Rightarrow \tau_{21} \texttt{ -> } \tau_{22} \rangle^l \texttt{ f})\ \texttt{v}$ will reduce to $\langle \tau_{12} \Rightarrow \tau_{22} \rangle^l\ (\texttt{f}\ (\langle \tau_{21} \Rightarrow \tau_{11} \rangle^l\ \texttt{v}))$ wrapping the argument `v` in a (contravariant) cast between the domain types and wrapping the result of the application in a (covariant) cast between the range types.

To illustrate how function casts work let's once more consider the `pred` example. To get the desired type signature for `pred`, we have to wrap the function's definition in a type cast:

$$\texttt{pred'}\ x = x - 1$$

$$\texttt{pred} = \langle \texttt{Int->Int} \Rightarrow x : \{\texttt{v:Int} \mid \texttt{v > 0}\} \texttt{->} \{\texttt{v:Int} \mid \texttt{v = x - 1}\} \rangle^l\ \texttt{pred'}$$

If we apply a positive number, say `2 :: {v:Int | v > 0}`, we will have the following computation:

$$\begin{aligned}
&\texttt{pred 2} \\
&= (\langle \texttt{Int->Int} \Rightarrow x : \{\texttt{v:Int} \mid \texttt{v > 0}\} \texttt{->} \{\texttt{v:Int} \mid \texttt{v = x - 1}\} \rangle^l\ \texttt{pred'})\ \texttt{2} \\
&\rightarrow^\star (\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v = 2 - 1}\} \rangle^l)\ (\texttt{pred'}\ (\langle \{\texttt{v:Int} \mid \texttt{v > 0}\} \Rightarrow \texttt{Int} \rangle^l \texttt{2})) \\
&\rightarrow^\star (\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v = 2 - 1}\} \rangle^l)\ (\texttt{pred' 2}) \\
&\rightarrow^\star (\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v = 2 - 1}\} \rangle^l)\ \texttt{1} \\
&\rightarrow^\star \texttt{1}
\end{aligned}$$

The first line is `pred`'s definition. In the second line the rule for functional cast is applied. Then, the check that `2` is an integer succeeds, and `2` is applied to `pred'` so, we get `1`. Finally, this result is checked to be `1` and since this check succeeds the value is returned. If `pred'` was not returning the correct value, the program would raise a blame:

$$
\begin{aligned}
&\texttt{pred'}\ x = 0 \\
&\texttt{pred}\ 2 \\
&= (\langle \texttt{Int->Int} \Rightarrow x : \{\texttt{v:Int} \mid \texttt{v > 0}\}\texttt{->}\{\texttt{v:Int} \mid \texttt{v = x - 1}\}\rangle^{l}\ \texttt{pred'})\ 2 \\
&\rightarrow^{\star} (\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v = 2 - 1}\}\rangle^{l})\ (\texttt{pred'}\ (\langle \{\texttt{v:Int} \mid \texttt{v > 0}\} \Rightarrow \texttt{Int}\rangle^{l} 2)) \\
&\rightarrow^{\star} (\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v = 2 - 1}\}\rangle^{l})\ (\texttt{pred'}\ 2) \\
&\rightarrow^{\star} (\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v = 2 - 1}\}\rangle^{l})\ 0 \\
&\rightarrow^{\star} \Uparrow l
\end{aligned}
$$

The evaluation is the same as in the previous example, up to the point where the `pred'` application returns. Here, the application returns `0`, thus the final check fails and the program raises the blame `l`.

You may notice that in both cases `pred'` is applied to a positive integer. Since a positive integer is not a primitive type, the only way to get such a type is via a cast. Thus, for this application to statically typecheck, the argument should be wrapped in a cast. But, if we cast a non-positive value to be positive, then the cast will fail:

$$\texttt{pred}\ ((\langle \texttt{Int} \Rightarrow \{\texttt{v:Int} \mid \texttt{v > 0}\}\rangle^{zero})\ 0) \rightarrow^{\star} \Uparrow zero$$

We saw that two distinct casts should be used to satisfy the functions pre- and post-conditions. These casts use different labels, with which we can track the source of failure, if any.

3.2.2 Formal Language

Lets now extend our core calculus λ_c to λ_{cc}, so that it supports manifest contracts. In the expressions of our language we add a blaming expression and a type cast. The refinement language includes any core expression. Everything else remains unchanged.

In the typing judgements we add two rules: a blame expression can have any well-formed type, while a type cast expression behaves as a function from the source to the target type. For a casting expression to typecheck, both types should be well-formed and compatible, i.e., their unrefined types should be the same. We check this with a new compatibility judgement.

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & e & ::= & \ldots \mid \Uparrow l \mid \langle \tau \Rightarrow \tau \rangle^{l} \\
\textbf{\textit{Predicates}} & p & ::= & e
\end{array}
$$

Figure 4: **Syntax** from λ_C to λ_{CC}

Compatibility $\boxed{\tau_1 \| \tau_2}$

$$\frac{}{\{v : b \mid p_1\} \| \{v : b \mid p_2\}} \text{ C-BASE} \qquad \frac{\tau_{x_1} \| \tau_{x_2} \quad \tau_1 \| \tau_2}{x_1 : \tau_{x_1} \to \tau_1 \| x_2 : \tau_{x_2} \to \tau_2} \text{ C-FUN}$$

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \Uparrow l : \tau} \text{ T-LABEL} \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2 \quad \tau_1 \| \tau_2}{\Gamma \vdash \langle \tau_1 \Rightarrow \tau_2 \rangle^l : (x : \tau_1 \to \tau_2)} \text{ T-CAST}$$

Figure 5: **Static Semantics** from λ_C to λ_{CC}

4 Decidable Type Systems

In 1991 Freeman and Pfennning [**?**] introduced a decidable refinement type system for a subset of ML. In their system, the programmer defines refinement types for the vanilla data types; for example, the vanilla list data type can be refined to describe *nil* lists, or *singleton* lists, i.e., lists with exactly one element. These definitions are used to construct a finite datatype lattice of each ML type; a singleton list or a nill list is also a vanilla list, thus both refined lists are less than the unrefined one in the lattice. The datatype lattice is a representation of the subtype relationship that is used in the refinement type inference algorithm. Since each lattice is finite, the subtyping relation is decidable.

Later, they extended [**?**] their language to support linear arithmetic constraints; thus they could encode a list with length some integer n and reason about safety of list indexing. In this system, subtyping reduces to predicate implication and they used a variant of Fourier's method [**?**] for constraint solving. Finally, they created DML(C) [**?**], an extension of ML with refinement types, that supports array bounds check elimination, redundant pattern matching clause removal, tag check elimination and untagged representation of datatypes. Refinements in DML(C) are restricted to a finite and decidable constrain domain C, which renders constraint solving, and thus subtyping decidable.

DML(C) is a practical programming language, in the sense that programs can often be annotated with very little internal change and the resulting constraint simplification problems can be solved efficiently in practice. Its disadvantage is that annotation burden is high for the programmer, as often 10-20% of the code is typing annotations. In order to encourage programmers to use refinement specifications in their programs, Ou et al. [**?**], proposed a language design and type system that allows programmers to add semantic specifications to program fragments bit by bit. More specifically, for certain program components the type checker verifies statically the refinement type specifications. The rest of the components are written as in any ordinary simply-typed programming language. When control passes between different components, data flowing from simply-typed code into refinement-typed code is checked dynamically to ensure that the invariants hold.

Another system that combines static verification with dynamic checks is presented in Flanagan's Hybrid Type Checking [**?**]. Flanagan's type system uses syntactic sybtyping to create implications, as discusses in Section **??**. Moreover, he assumes an algorithm that decides the validity of the implications. For each implication the algo-

rithm runs for limited time: if it answers unsafe, the program is unsafe, but if it does not terminate, a cast is added to postpone the check until runtime. Thus, his system checks implications statically, whenever possible and dynamically, only when necessary.

In Liquid Types [**?**], implication checking always terminates, as implications belong to a decidable subset of first order logic. This is achieved by restricting the refinement language according to a finite set of qualifiers. With this technique, liquid type system allows type inference, as a means of decreasing the annotation burden. We present Liquid Types in the rest of the section.

Many systems discussed so far, including DML(C), Hybrid Type System and Liquid Types, use *syntactic subtyping* for constraint generation and SMT solvers for constraint solving. *Satisfiability Modulo Theories* (SMT) solvers solve implications for (fragments of) first-order logic plus various standard theories such as equality, real and integer (linear) arithmetic, uninterpreted functions, bit vectors, and (extensional) arrays. Some of the leading systems include CVC3 [**?**], Yices [**?**], and Z3 [**?**].

With the advent of SMT solvers, the combination of syntactic subtyping for constraint generation and an SMT solver for constraint solving has been used in various systems: Mandelbaum et al. [**?**], extended the domain of predicates to describe the state and the effects of the verified programs. Suter et al. [**?**] increase the power of reasoning to support user defined recursive functions. Finally, Unno et al. [**?**], created a relatively complete system for higher-order functional programs.

Apart from syntactic subtyping, SMT solvers can be used in other refinement decidable systems: Dminor [**?**] uses *semantic subtyping* where subtyping is totally decided by first order implication checking, while HALO [**?**] uses *denotational semantics* to prove specification checking.

4.1 Liquid Types

In Liquid Types [**?**], Rondon et al. restrict the refinement language according to a finite set of qualifiers, and achieve not only decidable type checking, but also automatic type inference.

The system takes as input a program and a finite set of *logical qualifiers* which are simple boolean predicates that encode the properties to be verified. The system then infers *liquid types*, which are refinement types where the refinement predicates are conjunctions of the logical qualifiers. Type checking and inference are decidable for three reasons. First, they use a conservative but decidable notion of subtyping, where subtyping reduces to implication checks in a decidable logic. Each implication holds if and only if it yields a valid formula in the logic. Second, an expression has a valid liquid type derivation only if it has a valid unrefined type derivation, and the refinement type of every subexpression is a refinement of its vanilla type. Third, in any valid type derivation, the types of certain expressions must be liquid. Thus, inference becomes decidable, as the space of possible types is bounded.

Logical Qualifiers and Liquid Types. A logical qualifier is a boolean-valued expression over the program variables, the special value variable `v` which is distinct from the program variables, and the special placeholder variable $\star$ that can be instantiated with program variables. Let $\mathbb{Q}$ be the set of logical qualifiers `{0 < ⋆, v < ⋆, v = ⋆ + 1}`. A qualifier q matches the qualifier q' if replacing some variables in q with $\star$ yields q'. For example, the qualifier `v < x` matches the qualifier `v < ⋆`. $\mathbb{Q}^\star$ is the set of all qualifiers not containing $\star$ that match some qualifier in $\mathbb{Q}$. For instance, if `x`, `y` and `n` are program variables, $\mathbb{Q}^\star$ includes the qualifiers `{0 < x, v = n + 1,`

`v < n, v < y}`. A liquid type over $\mathbb{Q}$ is a refinement type where the refinement predicates are conjunctions of qualifiers from $\mathbb{Q}^\star$.

Type Inference. Type inference is performed in three steps: (1) the vanilla type of each expression is refined with liquid variables which represent the unknown refinements; (2) syntactic subtyping is used to create implication constraints between the unknown variables and the concrete refinements; (3) a theorem prover is used to find the strongest conjunction of qualifiers in $\mathbb{Q}$ that satisfies the subtyping constraints.

To illustrate this procedure, consider our `pred` example:

```
pred n = n - 1
```

The liquid type for `pred` can be inferred in three steps:

(Step 1) By Hindley-Milner, we can infer that `pred` has the type `Int -> Int`. Using this type we create a template for the liquid type of `pred`,

```
pred :: n:{v:Int | kn} -> {v:Int | kp}
```

where `kn` and `kp` are liquid type variables representing the unknown refinements for the argument `n` and the body of `pred`, respectively.

(Step 2) We assume a descriptive type for minus:

```
(-) :: x:Int -> y:Int -> {v:Int | v = x - y}
```

and use it to construct the type of `pred`'s result:

```
{v:Int | v = x - y}[x/n][y/1] = {v:Int | v = n - 1}
```

This type should be subtype of the template type of the body:

```
{v:Int | v = n - 1} <: {v:Int | kp}
```

The above subtype reduces to the following constraint:

$$\texttt{v = n - 1} \Rightarrow \texttt{kp}$$

(Step 3) Since the program is "open", i.e., there are no calls to `pred`, we assign `kp` true, meaning that any integer argument can be passed, and use a theorem prover to find the strongest conjunction of qualifiers in $\mathbb{Q}$ that satisfies the subtyping constraints. The algorithm infers that $v = n - 1$ is the strongest solution for `kp`. By substituting the solution for `kp` into the template for `pred`, the algorithm infers

```
pred :: n:Int -> {v:Int | v = n-1}
```

Type Checking. As one may notice the inferred type signature of `pred` does not constrain the type of the argument. This is correct, as `pred`'s definition does not constrain its argument. One could give `pred` a more precise type, say:

```
pred :: n:{v:Int | v > 0} -> {v:Int | n - 1}
```

The system can verify that this type holds, following a procedure similar to the one for type inference:

The first step can be skipped, since there exists a concrete type for `pred`. The body of the function will be type-checked against the given signature. In the second step, as

before, we construct the type of the body to be `{v:Int | n - 1}` and constrain this type to be a subtype of `pred`'s result, or

```
{v:Int | v = n - 1} <: {v:Int | v = n - 1}
```

This subtyping reduces to a trivial implication `v = n - 1` ⇒ `v = n - 1` that can be proven in the third step.

Given the above type signature if `pred` is called with some positive integer value, say `2`, then in the call site the constraint `v = 2 ⇒ v >0` will be generated, that can be statically verified. However, if it is called with a non-positive value, say `0`, we will get the unsatisfied constraint `v = 0 ⇒ v > 0`, so the program will be unsafe.

4.1.1 Applications of Liquid Types

Liquid Types, as introduced in [**?**], used OCaml as target language and were used to verify array bounds checking. One year later [**?**], they were extended with recursive and polymorphic refinements to enable static verification of complex data structures; among which list sortedness or Binary Tree ordering. Liquid Types were used to verify properties even in imperative languages. Low-level Liquid Types [**?**] is a refinement type system for C based on Liquid Types to verify memory safety properties, like the absence of array bounds violations and null-dereferences. Finally, Liquid Effects [**?**], is a type-and-effect system based on refinement types which allows for fine-grained, low-level, shared memory multithreading while statically guaranteeing that a program is deterministic.

4.1.2 Formal Language

We extend the core calculus λ_C to λ_L, a calculus that supports liquid type checking.

The crucial difference between the previous systems, is that the refinement language can not contain arbitrary expressions, but is constrained to conjunctions of the logical qualifiers, which form a finite set, as shown in Figure **??**.

Static typing uses syntactic subtyping, as defined in Section **??**. In this setting, the subtyping relation is decidable because the refinement language, and thus the implications created, refer to a decidable logic. Finally, the `Valid` relation is evaluated using the Z3 [**?**] SMT solver.

$$\textbf{\textit{Predicates}} \qquad p ::= \quad \text{true} \mid q \mid p \wedge p \,,\; q \in \mathbb{Q}^\star$$

Figure 6: **Syntax** from λ_C to λ_L

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_2 \preceq \tau_1 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash e : \tau_1} \text{ T-Sub}$$

Figure 7: **Static Semantics** from λ_C to λ_L

5 Abstract Refinement Types

Refinement type systems, as presented so far, fall into two categories. Expressive type systems, as presented in Section **??**, are statically undecidable, while decidable systems, as presented in Section **??**, restrict the refinement language to a subset of first order logic. In this section, we present abstract refinement types [**?**], a means to enhance expressiveness of a refinement system, while preserving (SMT-based) decidability. The key insight is that we avail quantification over the refinements of data- and function-types, simply by encoding refinement parameters as uninterpreted propositions within the refinement logic. We illustrate how this mechanism yields a variety of sophisticated means for reasoning about programs, including: inductive refinements for reasoning about higher-order traversal routines, compositional refinements for reasoning about function composition, index-dependent refinements for reasoning about key-value maps, and recursive refinements for reasoning about recursive data types.

5.1 The key idea

Consider the monomorphic `max` function on `Int` values. We give `max` a refinement type, stating that its result is greater or equal than both its arguments:

```
max     :: x:Int -> y:Int -> {v:Int | v >= x && v >= y}
max x y = if x > y then x else y
```

If we apply `max` to two positive integers, say `n` and `m`, we get that the result is greater or equal to both of them, as `max n m :: {v:Int | v >= n && v >= m}`. However, we can not reason about an arbitrary property: If we apply `max` to two even numbers, can not verify that the result is also even. Thus, even though we have the information that both arguments are even on the input, we lose it on the result.

To solve this problem, we introduce *abstract refinements* which let us quantify or parameterize a type over its constituent refinements. Using abstract refinements, we can type `max` as

```
max :: forall <p::Int->Bool>. Int<p> -> Int<p> -> Int<p>
```

where `Int<p>` is an abbreviation for the refinement type `{v:Int | p(v)}`. Intuitively, an abstract refinement `p` is encoded in the refinement logic as an *uninterpreted function symbol*, which satisfies the *congruence* axiom [**?**]

$$\forall \overline{X}, \overline{Y} : (\overline{X} = \overline{Y}) \Rightarrow P(\overline{X}) = P(\overline{Y})$$

It is trivial to verify, with an SMT solver, that `max` enjoys the above type: the input types ensure that both `p(x)` and `p(y)` hold and hence the returned value in either branch satisfies the refinement `{v:Int | p(v)}`, thereby ensuring the output type.

In a call site, we simply instantiate the *refinement* parameter of `max` with the concrete refinement, after which type checking proceeds as usual. As an example, suppose that we call `max` with two even numbers:

```
n :: {v:Int | even v}
m :: {v:Int | even v}
```

Then, the abstract refinement can be instantiated with a concrete predicate `even`, which will give `max` the type

```
max [even] ::
{v:Int | even v} -> {v:Int | even v} -> {v:Int | even v}
```

where the expression in brackets is the refinement instantiation. Since both `n` and `m` are even numbers, they satisfy the function's preconditions, thus we can apply them to `max`, to get an even result:

```
max [even] n m :: {v:Int |  even v}
```

This is the basic concept of abstract refinements, which as we shall see, have many interesting applications.

5.2 Inductive Refinements

As a first application we present how abstract refinements allow us to formalize induction within the type system.

Consider a `loop` function that takes as arguments a function `f`, an integer `n`, a base case `z` and applies the function `f` to the `z`, `n` times:

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n     = go (i+1) (f i acc)
                 | otherwise = acc
```

Now consider a user function `incr` that uses `loop` and at each iteration increases the accumulator by one:

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

The accumulator is initialized with `z` and at each `loop`'s iteration it is increased by `1`. So, at the `i`th iteration, the accumulator is equal to `z+i`. There will be `n` iterations, thus the final result will be `z+n`. This reasoning constitutes an inductive proof that characterizes `loop`'s behaviour. However, it is unclear how to give `loop` a (first-order) refinement type that describes its inductive behaviour. Hence, it has not been possible to verify that `incr` actually adds its two arguments.

Typing loop. Abstract refinements allow us to solve this problem, while remaining within the boundaries of SMT-based decidability. We give `loop` the following type:

```
loop :: forall <r :: Int -> a -> Bool> .
           f : (i:Int -> a<r i> -> a<r (i+1)>)
        -> n : {v:Int | n >= 0}
        -> z : a<r 0>
        -> a<r n>
```

The trick is to qualify over the invariant `r` that `loop` establishes between the loop iteration and the accumulator. Then the type signature encodes induction on natural numbers: (1) `n` should be a natural number, thus a non-negative integer, (2) the base case `z` should satisfy the invariant at 0, (3) in the inductive step, `f` uses the old accumulator to create the new one, thus if the old accumulator satisfies the invariant on the iteration `i`, the new one, as constructed by `f`, should satisfy the invariant at `i+1`. If these four conditions hold, we conclude that the result satisfies the invariant at `n`. This scheme is not novel [**?**]; what is new is the encoding, via uninterpreted predicate symbols in a SMT-decidable refinement type system.

Using loop. We can use this expressive type of `loop` to verify inductive properties of user functions:

```
incr :: n:{v:Int|v >= 0} -> z:Int -> {v:Int|v = n + z}
incr n z = loop [{\i acc -> acc + i}] f n z
  where f i acc = acc + 1
```

In the above example, the expression in brackets denotes the instantiation of the abstract refinement. For purpose of illustration we make abstract refinement instantiation explicit, but it could be automatically inferred via liquid typing [**?**].

5.3 Function Composition

As a next example, we present how one can use abstract refinements to reason about function composition.

Consider a `plusminus` function that composes a plus and a minus operator:

```
plusminus :: n:Int
          -> m:Int
          -> x:Int
          -> {v:Int | v = (x - m) + n}
plusminus n m x = (x - m) + n
```

In a first order refinement system we can verify that the function's behaviour is captured by its type. However, consider an alternative definition that uses function composition `(.):: (b -> c)-> (a -> b)-> a -> c`.

```
plusminus n m x = plus . minus
  where plus  x = x + n
        minus x = x - m
```

It is unclear how to give `(.)` a (first-order) refinement type that expresses that the result can be refined with the composition of the refinements of both arguments results. Thus, this definition of `plusminus` can not have the previous descriptive type.

Typing function composition. To solve this problem, we can use abstract refinements and give `(.)` a type:

```
(.) :: forall < p :: b -> c -> Bool
              , q :: a -> b -> Bool>.
       f : (x:b -> c<p x>)
    -> g : (x:a -> b<q x>)
    -> x : a
    -> exists[z:b<q x>]. c<p z>
```

The trick is once again to quantify the type over refinements we care about. This time, we use two abstract refinements: the refinement `p` of the result of the first function `f` and the refinement `q` of the result of the second function `g`. For any argument `x`, we use an existential to bind the intermediate result to `z = g x`, so `z` satisfies `q` at `x`, and the result satisfies `p` at the intermediate result.

Using function composition. With this type for function composition, user functions get the concrete refinement of the final result to be the composition of the two refinements of the argument functions.

Back to the `plusminus` example, with the appropriate refinement instantiation we get the concrete refinement type for function composition:

```
(.) [{\x v -> v = x + n}, {\x v -> v = x - m}]
    :: f : (x:b -> {v:c | v = x+n})
    -> g : (x:a -> {v:b | v = x-m})
    -> x : a
    -> exists[z:{v:b | v = x-m}]. {v:c | v = z+n}
```

The result type asserts that there exists a value `z`, which is indeed the intermediate result, with the property `z = x - m`. With this, the final result is equal to `z + n`. If our logic supports equality, as SMT solvers do, we can verify that the final result is indeed equal to `(x - m)+ n`. In other words, we can verify the desired type of `plusminus`.

5.4 Index-Dependent Invariants

Next, we illustrate how abstract invariants allow us to specify and verify index-dependent invariants of key-value maps. To this end, we encode vectors as functions from `Int` to some generic range `a`. Formally, we specify vectors as

```
data Vec a <dom :: Int -> Bool, rng :: Int -> a -> Bool>
  = V (i:Int<dom> -> a <rng i>)
```

Here, we are parameterizing the definition of the type `Vec` with two abstract refinements, `dom` and `rng`, which respectively describe the *domain* and *range* of the vector. That is, `dom` describes the set of valid indices, and `rng` specifies an invariant relating each `Int` index with the value stored at that index.

Describing Vectors. With this encoding, we can describe various vectors. To start with we can have vectors of `Int` defined on positive integers with values equal to their index:

```
Vec <{\v -> v > 0}, {\_ v -> v = x}> Int
```

Or a vector that is defined only on index 1 with value 12:

```
Vec <{\v -> v = 1}, {\_ v -> v = 12}> Int
```

As a more interesting example, we can define a *Null Terminating String* with length `n`, as a vector of `Char` defined on a range `[0, n)` with its last element equal to the terminating character:

```
Vec <{\v -> 0 <= v < n}
    ,{\i v -> i = n-1 => v = '\0'}> Char
```

Finally, we can encode a *Fibonacci memoization vector*, which can be used to efficiently compute a Fibonacci number, that is defined on positive integers and its value on index `i` is either zero or the `i`th Fibonacci number:

```
Vec <{\v -> 0 <= v}
    ,{\i v -> v != 0 => v = fib(i)}> Char
```

Using Vectors. A first step towards using vectors is to supply the appropriate types for vector operations, (e.g., set, get and empty). This usually means qualifying over

the domain and the range of the vectors. Then, the programmer has to specify interesting vector properties, as we did for the Fibonacci memoization, or the null terminating string. Finally, the system can verify that user functions, that transform vectors, preserve these properties. This procedure is applied in [**?**], where, with the appropriate types for vector operations, we reason about functions that transform null terminating strings or efficiently compute a Fibonacci number.

5.5 Recursive Invariants

Finally, we describe how we use abstract refinements to reason about properties of recursive data structures. For the purpose of illustration, we define a refined version of a `List` datatype with values of type `a`:

```
data List a <p :: a -> a -> Bool>
  = N
  | C (hd :: a) (tl :: List <p> (a <p h>))
```

We are parametrizing the `List` over an abstract refinement `p` that relates two elements of type `a`. With this, the list is either the empty list `N`, or contains a head `hd` of type `a` and a tail `tl` which is a list of elements of type `a<p h>`, i.e., these elements satisfy the abstract refinement `p` at the head. Since this definition is recursively applied, the abstract refinement `p` holds between each pair of elements in the list.

Unfolding Lists. To demonstrate the previous property, we will unfold a `List` with three elements that satisfies an abstract refinement `p`. Consider such a list:

```
C h1 (C h2 (C h3 N)) :: List <p> a
```

If we unfold this list once, by the definition of the `C` data constructor, the first element is of type `a`, while the rest is a list with values that satisfy `p` at the first element, i.e., `(C h2 (C h3 N))::` `List <p> a<p h1>`. With a second unfold we get that the second element satisfies `p` at the first element, i.e., `h2::a<p h1>`, while the rest is a list with values that satisfy `p` at both the first and the second element, i.e., `C h3 N :: List <p> a<p h1 && p h2>`. With the last unfold we get that the last element satisfies `p` at all the previous elements, i.e., `h3 :: a<p h1 && p h2>`, while the empty list satisfies `p` at every list element, i.e., `N :: List <p> a<p h1 && p h2 && p3>`, which holds as by its definition the empty list `N` satisfies any refinement.

Thus, `p` holds between every pair of the list, as for any two two elements `hi` and `hj`, with `i < j`, at the `i`th unfold `hj` satisfies `p` at `hi`.

If we instantiate the abstract refinement `p` with the concrete refinement `{\h v -> h <= v}`, that expresses that each values is greater than the head, we get that each element is greater than all its previous in the list. So we describe an increasing list:

```
type IncrList a = List <{\h v -> h <= v}> a
```

We can describe different list properties, by embedding appropriate concrete refinements. For instance, if we use a refinement that expresses that each value is less than the head, i.e., `{\h v -> h >= v}` or different from it, i.e., `{\h v -> h ~= v}`, we can describe decreasing or unique element lists.

Using Lists. We can use the refined type for lists to verify list properties. As an example, our system can verify that the following inserting sort algorithm actually returns an increasing list.

```
insertSort :: (Ord a) => [a] -> IncrList a
insertSort = foldr insert N

insert :: (Ord a) => a -> IncrList a -> IncrList a
insert y N          = C y N
insert y (C x xs) | y <= x        = C y (C x xs)
                  | otherwise     = C x (insert y xs)
```

5.6 Formal Language

We suggest that any refinement system can be extended with abstract refinements without increasing its complexity. First of all, the syntax should be extended to support refinement abstraction and application. In the case of refinement abstraction, we abstract from an expression e the refinement π with type τ, while in refinement application we instantiate an abstrast refinement with a concrete one p that may have some parameters $\bar{x}$. The predicates of the language should be extended to include abstract refinements, applied to program expressions. The types of the language should also be extended to include refinement abstraction.

Since we extended our expressions the relevant typing rules should be added. The refinement abstraction expression is typed as an refinement abstraction type, the abstract refinement is treated as a variable and the checking proceeds in a straightforward way. In the refinement application, the abstract refinement π is replaced with a concrete one over the type τ. A formal definition of this substitution can be found in our paper[**?**].

Similarly, since we extended our types, the well-formedness and subtyping rules should be extended. In both cases, the abstract refinement is added in the environment and the check proceeds in a straightforward way.

We note that abstract refinements can be treated as uninterprented functions in the implication checking algorithm, thus the complexity of the system is not increased. Moreover, they appear only in the types, thus they can be erased at run-time.

$$
\begin{array}{rrcl}
\textbf{\textit{Expressions}} & e & ::= & \ldots \mid \Lambda\pi : \tau.e \mid e\,[\lambda\overline{x : \tau_x}.p] \\
\textbf{\textit{Predicates}} & p & ::= & \ldots \mid \pi\,\overline{e} \\
\textbf{\textit{Types}} & \tau & ::= & \ldots \mid \forall\pi : \tau.\tau
\end{array}
$$

Figure 8: **Syntax of Expressions, Types and Schemas**

6 LIQUIDHASKELL

LIQUIDHASKELL combines Liquid Types (§ **??**) with Abstract Refinements (§ **??**) to give an expressive and decidable verification mechanism for Haskell programs. We will start with a short description of the LIQUIDHASKELL workflow, summarized in

Type Checking $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma, \pi : \tau_\pi \vdash e : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \Lambda\pi : \tau_\pi . e : \forall \pi : \tau_\pi . \tau} \text{ T-GEN} \qquad \frac{\Gamma \vdash e : \forall \pi : \tau_\pi . \tau \quad \Gamma \vdash \lambda \overline{x : \tau_x} . p : \tau_\pi}{\Gamma \vdash e\,[\lambda \overline{x : \tau_x} . p] : \tau[\pi \rhd \lambda \overline{x : \tau_x} . p]} \text{ T-INST}$$

Subtyping $\boxed{\Gamma \vdash \tau \preceq \tau}$

$$\frac{(\Gamma, v : b \vdash \texttt{Valid}(p_1 \Rightarrow p_2))}{\Gamma \vdash \{v : b \mid p_1\} \preceq \{v : b \mid p_2\}} \prec\text{-BASE}$$

Figure 9: **Static Semantics** from λ_C to λ_A

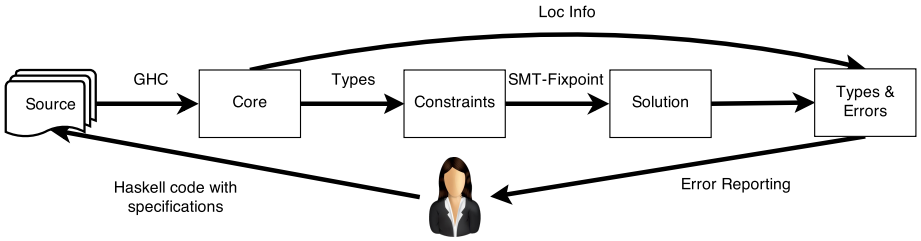

Figure 10: LIQUIDHASKELL Workflow

Figure **??**, and continue with an example driven overview of how properties are specified and verified using the tool.

Source. LIQUIDHASKELL can be run from the command-line[1] or within a web-browser[2]. It takes as *input*: (1) a single Haskell *source* file with code and refinement type specifications including refined datatype definitions, measures (§ **??**), predicate and type aliases, and function signatures; (2) a set of directories containing *imported modules* (including the `Prelude`) which may themselves contain specifications for exported types and functions; and (3) a set of predicate fragments called *qualifiers*, which are used to infer refinement types. This set is typically empty as the default set of qualifiers extracted from the type specifications suffices for inference.

Core. LIQUIDHASKELL uses GHC to reduce the source to the Core IL [**?**], and, to facilitate source-level error reporting, creates a map from Core expressions to locations in the Haskell source.

Constraints. Then, it uses the abstract interpretation framework of Liquid Typing [**?**], modified to ensure soundness under lazy evaluation [**?**], to generate logical constraints from the Core IL.

Solution. Next, it uses a fixpoint algorithm (from [**?**]) combined with an SMT solver to solve the constraints, and hence infers a valid refinement typing for the program. LIQUIDHASKELL can use any solver that implements the SMT-LIB2 standard [**?**], including Z3 [**?**], CVC4 [**?**], and MathSat [**?**].

Types & Errors. If the set of constraints is satisfiable, then LIQUIDHASKELL out-

[1] https://hackage.haskell.org/package/liquidhaskell
[2] http://goto.ucsd.edu/liquid/haskell/demo/

puts SAFE, meaning the program is verified. If instead, the set of constraints is not satisfiable, then LIQUIDHASKELL outputs UNSAFE, and uses the invalid constraints to report refinement type errors at the *source positions* that created the invalid constraints, using the location information to map the invalid constraints to source positions. In either case, LIQUIDHASKELL produces as output a source map containing the *inferred* types for each program expression, which, in our experience, is crucial for debugging the code and the specifications.

LIQUIDHASKELL is best thought of as an *optional* type checker for Haskell. By optional we mean that the refinements have *no* influence on the dynamic semantics, which makes it easy to apply LIQUIDHASKELL to *existing* libraries. To emphasize the optional nature of refinements and preserve compatibility with existing compilers, all specifications appear within comments of the form `{-@ ... @-}`, which we omit below for brevity.

6.1 Specifications

A refinement type is a Haskell type where each component of the type is decorated with a predicate from a (decidable) refinement logic. We use the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [**?**]. For example,

```
{v:Int | 0 <= v && v < 100}
```

describes `Int` values between `0` and `100`.

Type Aliases. For brevity and readability, it is often convenient to define abbreviations for particular refinement predicates and types. For example, we can define an alias for the above predicate

```
predicate Btwn Lo N Hi = Lo <= N && N < Hi
```

and use it to define a *type alias*

```
type Rng Lo Hi = {v:Int | (Btwn Lo v Hi)}
```

We can now describe the above integers as `(Rng 0 100)`.

Contracts. To describe the desired properties of a function, we need simply refine the input and output types with predicates that respectively capture suitable pre- and post-conditions. For example,

```
range :: lo:Int -> hi:{Int | lo <= hi}
      -> [(Rng lo hi)]
```

states that `range` is a function that takes two `Int`s respectively named `lo` and `hi` and returns a list of `Int`s between `lo` and `hi`. There are three things worth noting. First, we have binders to name the function's *inputs* (*e.g.*, `lo` and `hi`) and can use the binders inside the function's *output*. Second, the refinement in the *input* type describes the *pre-condition* that the second parameter `hi` cannot be smaller than the first `lo`. Third, the refinement in the *output* type describes the *post-condition* that all returned elements are between the bounds of `lo` and `hi`.

6.2 Verification

Next, consider the following implementation for `range`:

```
range lo hi
  | lo <= hi  = lo : range (lo + 1) hi
  | otherwise = []
```

When we run LIQUIDHASKELL on the above code, it reports an error at the definition of `range`. This is unpleasant! One way to debug the error is to determine what type has been *inferred* for `range`, *e.g.*,, by hovering the mouse over the identifier in the web interface. In this case, we see that the output type is essentially:

```
[{v:Int | lo <= v && v <= hi}]
```

which indicates the problem. There is an *off-by-one* error due to the problematic guard. If we replace the second <= with a < and re-run the checker, the function is verified.

Holes. It is often cumbersome to specify the Haskell types, as those can be gleaned from the regular type signatures or via GHC's inference. Thus, LIQUIDHASKELL allows the user to leave holes in the specifications. Suppose `rangeFind` has type

```
(Int -> Bool) -> Int -> Int -> Maybe Int
```

where the second and third parameters define a range. We can give `rangeFind` a refined specification:

```
_ -> lo:_ -> hi:{Int | lo <= hi}
  -> Maybe (Rng lo hi)
```

where the _ is simply the unrefined Haskell type for the corresponding position in the type.

Inference. Next, consider the implementation

```
rangeFind f lo hi = find f $ range lo hi
```

where `find` from `Data.List` has the (unrefined) type

```
find :: (a -> Bool) -> [a] -> Maybe a
```

LIQUIDHASKELL uses the abstract interpretation framework of Liquid Typing [**?**] to infer that the type parameter `a` of `find` can be instantiated with `(Rng lo hi)` thereby enabling the automatic verification of `rangeFind`.

Inference is crucial for automatically synthesizing types for polymorphic instantiation sites – note there is another instantiation required at the use of the apply operator `$` – and to relieve the programmer of the tedium of specifying signatures for all functions. Of course, for functions exported by the module, we must write signatures to specify preconditions – otherwise, the system defaults to using the trivial (unrefined) Haskell type as the signature *i.e.*,, checks the implementation assuming arbitrary inputs.

6.3 Measures

So far, the specifications have been limited to comparisons and arithmetic operations on primitive values. We use *measure functions*, or just measures, to specify *inductive properties* of algebraic data types. For example, we define a measure `len` to write properties about the number of elements in a list.

```
measure len :: [a] -> Int
len []     = 0
len (x:xs) = 1 + (len xs)
```

Measure definitions are *not* arbitrary Haskell code but a very restricted subset [**?**]. Each measure has a single equation per constructor that defines the value of the measure for that constructor. The right-hand side of the equation is a term in the restricted refinement logic. Measures are interpreted by generating refinement types for the corresponding data constructors. For example, from the above, LIQUIDHASKELL derives the following types for the list data constructors:

```
[]  :: {v:[a]| len v = 0}
(:) :: _ -> xs:_ -> {v:[a]| len v = 1 + len xs}
```

Here, `len` is an *uninterpreted function* in the refinement logic. We can define multiple measures for a type; LIQUIDHASKELL simply conjoins the individual refinements arising from each measure to obtain a single refined signature for each data constructor.

Using Measures. We use measures to write specifications about algebraic types. For example, we can specify and verify that:

```
append :: xs:[a] -> ys:[a]
       -> {v:[a]| len v = len xs + len ys}

map    :: (a -> b) -> xs:[a]
       -> {v:[b]| len v = len xs}

filter :: (a -> Bool) -> xs:[a]
       -> {v:[a]| len v <= len xs}
```

Propositions. Measures can be used to encode sophisticated invariants about algebraic data types. To this end, the user can write a measure whose output has a special type **`Prop`** denoting propositions in the refinement logic. For instance, we can describe a list that contains a `0` as:

```
measure hasZero :: [Int] -> Prop
hasZero []      = false
hasZero (x:xs)  = x == 0 || (hasZero xs)
```

We can then define lists containing a `0` as:

```
type HasZero = {v : [Int] | (hasZero v)}
```

Using the above, LIQUIDHASKELL will accept

```
xs0 :: HasZero
xs0 = [2,1,0,-1,-2]
```

but will reject

```
xs' :: HasZero
xs' = [3,2,1]
```

6.4 Refined Data Types

Often, we require that *every* instance of a type satisfies some invariants. For example, consider a `CSV` data type, that represents tables:

```
data CSV a = CSV { cols :: [String]
                 , rows :: [[a]]    }
```

With LIQUIDHASKELL we can enforce the invariant that every row in a `CSV` table should have the same number of columns as there are in the header

```
data CSV a = CSV { cols :: [String]
                 , rows :: [ListL a cols] }
```

using the alias

```
type ListL a X = {v:[a]| len v = len X}
```

A refined data definition is *global* in that LIQUIDHASKELL will reject any `CSV`-typed expression that does not respect the refined definition. For example, both of the below

```
goodCSV = CSV [  "Month", "Days"]
              [ ["Jan"  , "31"]
              , ["Feb"  , "28"]
              , ["Mar"  , "31"] ]

badCSV  = CSV [  "Month", "Days"]
              [ ["Jan"  , "31"]
              , ["Feb"  , "28"]
              , ["Mar"        ] ]
```

are well-typed Haskell, but the latter is rejected by LIQUIDHASKELL. Like measures, the global invariants are enforced by refining the constructors' types.

6.5 Refined Type Classes

Next, let us see how LIQUIDHASKELL supports the verification of programs that use ad-hoc polymorphism via type classes. While the implementation of each typeclass instance is different, there is often a common interface that we expect all instances to satisfy.

Class Measures. As an example, consider the class definition

```
class Indexable f where
  size :: f a -> Int
  at   :: f a -> Int -> a
```

For safe access, we might require that `at`'s second parameter is bounded by the `size` of the container. To this end, we define a *type-indexed* measure, using the **`class measure`** keyword

```
class measure sz :: a -> Nat
```

Now, we can specify the safe-access precondition independent of the particular instances of `Indexable`:

```
class Indexable f where
  size :: xs:_ -> {v:Nat | v = sz xs}
  at   :: xs:_ -> {v:Nat | v < sz xs} -> a
```

Instance Measures. For each concrete type that instantiates a class, we require a corresponding definition for the measure. For example, to define lists as an instance of `Indexable`, we require the definition of the `sz` instance for lists:

```
instance measure sz :: [a] -> Nat
sz []     = 0
sz (x:xs) = 1 + (sz xs)
```

Class measures work just like regular measures in that the above definition is used to refine the types of the list data constructors. After defining the measure, we can define the type instance as:

```
instance Indexable [] where
  size []         = 0
  size (x:xs)     = 1 + size xs

  (x:xs) `at` 0   = x
  (x:xs) `at` i   = index xs (i-1)
```

LIQUIDHASKELL uses the definition of `sz` for lists to check that `size` and `at` satisfy the refined class specifications.

Client Verification. At the clients of a type-class we use the refined types of class methods. Consider a client of `Indexables`:

```
sum :: (Indexable f) => f Int -> Int
sum xs = go 0
  where
    go i | i < size xs = xs `at` i + go (i+1)
         | otherwise   = 0
```

LIQUIDHASKELL proves that each call to `at` is safe, by using the refined class specifications of `Indexable`. Specifically, each call to `at` is guarded by a check `i < size xs` and `i` is increasing from 0, so LIQUIDHASKELL proves that `xs `at` i` will always be safe.

6.6 Abstracting Refinements

So far, all the specifications have used *concrete* refinements. Often it is useful to be able to *abstract* the refinements that appear in a specification. For example, consider a monomorphic variant of `max`

```
max     :: Int -> Int -> Int
max x y = if x > y then x else y
```

We would like to give `max` a specification that lets us verify:

```
xPos  :: {v: _ | v > 0}
xPos  = max 10 13

xNeg  :: {v: _ | v < 0}
xNeg  = max (-5) (-8)

xEven :: {v: _ | v mod 2 == 0}
xEven = max 4 (-6)
```

To this end, LIQUIDHASKELL allows the user to *abstract refinements* over types [**?**], for example by typing `max` as:

```
max :: forall <p :: Int -> Prop>.
         Int<p> -> Int<p> -> Int<p>
```

The above signature states that for any refinement `p`, if the two inputs of `max` satisfy `p` then so does the output. LIQUIDHASKELL uses Liquid Typing to automatically instantiate `p` with suitable concrete refinements, thereby checking `xPos`, `xNeg`, and `xEven`.

Dependent Composition. Abstract refinements turn out to be a surprisingly expressive and useful specification mechanism. For example, consider the function composition operator:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Previously, it was not possible to check, *e.g.,* that:

```
plus3 :: x:_ -> {v:_ | v = x + 3}
plus3 = (+ 1) . (+ 2)
```

as the above required tracking the dependency between `a`, `b` and `c`, which is crucial for analyzing idiomatic Haskell. With abstract refinements, we can give the `(.)` operator the type:

```
(.) :: forall < p :: b -> c -> Prop
              , q :: a -> b -> Prop>.
         f:(x:b -> c<p x>)
      -> g:(x:a -> b<q x>)
      -> y:a
      -> exists[z:b<q y>].c<p z>
```

which gets automatically instantiated at usage sites, allowing LIQUIDHASKELL to precisely track invariants through the use of the ubiquitous higher-order operator.

Dependent Pairs. Similarly, we can abstract refinements over the definition of datatypes. For example, we can express dependent pairs in LIQUIDHASKELL by refining the definition of tuples as:

```
data Pair a b <p :: a -> b -> Prop>
  = Pair { fst :: a, snd :: b<p fst>}
```

That is, the refinement `p` relates the `snd` element with the `fst`. Now we can define increasing and decreasing pairs

```
type IncP = Pair <{\x y -> x < y}> Int Int
type DecP = Pair <{\x y -> x > y}> Int Int
```

and then verify that:

```
up :: IncP
up = Pair 2 5

dn :: DecP
dn = Pair 5 2
```

Now that we have a bird's eye view of the various specification mechanisms supported by LIQUIDHASKELL, let us see how we can profitably apply them to statically check a variety of correctness properties in real-world codes.

7 Totality

Well typed Haskell code can go very wrong:

```
*** Exception: Prelude.head: empty list
```

As our first application, let us see how to use LIQUIDHASKELL to statically guarantee the absence of such exceptions, *i.e.,*, to prove various functions *total*.

7.1 Specifying Totality

First, let us see how to specify the notion of totality inside LIQUIDHASKELL. Consider the source of the above exception:

```
head :: [a] -> a
head (x:_) = x
```

Most of the work towards totality checking is done by the translation to GHC's Core, in which every function *is* total, but may explicitly call an *error* function that takes as input a string that describes the source of the pattern-match failure and throws an exception. For example `head` is translated into

```
head d = case d of
           x:xs -> x
           []   -> patError "head"
```

Since every core function is total, but may explicitly call error functions, to prove that the source function is total, it suffices to prove that `patError` will *never* be called. We can specify this requirement by giving the error functions a `false` pre-condition:

```
patError :: {v:String | false } -> a
```

The pre-condition states that the input type is *uninhabited* and so an expression containing a call to `patError` will only type check if the call is *dead code*.

7.2 Verifying Totality

The (core) definition of `head` does not typecheck as is; but requires a pre-condition that states that the function is only called with non-empty lists. Formally, we do so by defining the alias

```
predicate NonEmp X = 0 < len X
```

and then stipulating that

```
head :: {v : [a] | NonEmp v} -> a
```

To verify the (core) definition of `head`, LIQUIDHASKELL uses the above signature to check the body in an environment

```
d :: {0 < len d}
```

When `d` is matched with `[]`, the environment is strengthened with the corresponding refinement from the definition of `len`, *i.e.,*,

```
d :: {0 < (len d) && (len d) = 0}
```

Since the formula above is a contradiction, LIQUIDHASKELL concludes that the call to `patError` is dead code, and thereby verifies the totality of `head`. Of course, now we have pushed the burden of proof onto clients of `head` – at each such site, LIQUIDHASKELL will check that the argument passed in is indeed a `NonEmp` list, and if it successfully does so, then we, at any uses of `head`, can rest assured that `head` will never throw an exception.

Refinements and Totality. While the `head` example is quite simple, in general, refinements make it very easy to prove totality in complex situations, where we must track dependencies between inputs and outputs. For example, consider the `risers` function from [**?**]:

```
risers []       = []
risers [x]      = [[x]]
risers (x:y:zs)
  | x <= y      = (x:s) : ss
  | otherwise   = [x] : (s:ss)
  where
    s:ss    = risers (y:etc)
```

The pattern match on the last line is partial; its core translation is

```
let (s, ss) = case risers (y:etc) of
                s:ss -> (s, ss)
                []   -> patError "..."
```

What if `risers` returns an empty list? Indeed, `risers` *does*, on occasion, return an empty list per its first equation. However, on close inspection, it turns out that *if* the input is non-empty, *then* the output is also non-empty. Happily, we can specify this as:

```
risers :: l:_ -> {v:_ | NonEmp l => NonEmp v}
```

LIQUIDHASKELL verifies that `risers` meets the above specification, and hence that the `patError` is dead code as at that site, the scrutinee is obtained from calling `risers` with a `NonEmp` list.

Non-Emptiness via Measures. Instead of describing non-emptiness indirectly using `len`, a user could a special measure:

```
measure nonEmp  :: [a] -> Prop
nonEmp (x:xs)   = true
nonEmp []       = false

predicate NonEmp X = nonEmp X
```

After which, verification would proceed analagous to the above.

Total Totality Checking. `patError` is one of many possible errors thrown by non-total functions. `Control.Exception.Base` has several others (`recSelError` , `irrefutPatError`, *etc.*) which serve the purpose of making core translations total. Rather than hunt down and specify `false` preconditions one by one, the user may automatically turn on totality checking by invoking LIQUIDHASKELL with the `--totality` command line option, at which point the tool systematically checks that all the above functions are indeed dead code, and hence, that all definitions are total.

7.3 Case Studies

We verified totality of two libraries: `HsColour` and `Data.Map`, earlier versions of which had previously been proven total by `catch` [**?**].

`Data.Map`. is a widely used library for (immutable) key-value maps, implemented as balanced binary search trees. Totality verification of `Data.Map` was quite straightforward. We had previously verified termination and the crucial binary search invariant [**?**]. To verify totality it sufficed to simply re-run verification with the `--totality` argument. All the important specifications were already captured by the types, and no additional changes were needed to prove totality.

This case study illustrates an advantage of LIQUIDHASKELL over specialized provers (*e.g.*, `catch` [**?**]), namely it can be used to prove totality, termination and functional correctness at the same time, facilitating a nice reuse of specifications for multiple tasks.

`HsColour`. is a library for generating syntax-highlighted LATEX and HTML from Haskell source files. Checking `HsColour` was not so easy, as in some cases assumptions are used about the structure of the input data: For example, `ACSS.splitSrcAndAnnos` handles an input list of `String`s and assumes that whenever a specific `String` (say `breakS`) appears then at least two `String`s (call them `mname` and `annots`) follow it in the list. Thus, for a list `ls` that starts with `breakS` the irrefutable pattern `(_:mname:annots) = ls` should be total. Currently it is somewhat cumbersome to specify such properties, and these are interesting avenues for future work. Thus to prove totality, we added a dynamic check that validates that the length of the input `ls` exceeds `2`.

In other cases assertions were imposed via monadic checks, for example `HsColour.hs` reads the input arguments and checks their well-formedness using

```
when (length f > 1) $ errorOut "..."
```

Currently LIQUIDHASKELL does not support monadic reasoning that allows assuming that `(length f <= 1)` holds when executing the action *following* the `when` check. Finally, code modifications were required to capture properties that currently we do not know how to express with LIQUIDHASKELL. For example, `trimContext` checks if there is an element that satisfies `p` in the list `xs`; if so it defines `ys = dropWhile (not . p) xs` and computes `tail ys`. By the check we know that `ys` has at least one element, the one that satisfies `p`, but this is a property that we could not express in LIQUIDHASKELL.

On the whole, while proving totality can be cumbersome (as in `HsColour`) it is a nice side benefit of refinement type checking, and can sometimes be a fully automatic corollary of establishing more interesting safety properties (as in `Data.Map`).

8 Termination

To soundly account for Haskell's non-strict evaluation, a refinement type checker must distinguish between terms that may potentially diverge and those that will not [**?**]. Thus, by default, LIQUIDHASKELL proves termination of each recursive function. Fortunately, refinements make this onerous task quite straightforward. We need simply associate a *well-founded termination metric* on the function's parameters, and then use refinement typing to check that the metric strictly decreases at each recursive call. In

practice, due to a careful choice of defaults, this amounts to about a line of termination-related hints per hundred lines of source. Details about the termination checker may be found in [**?**], we include a brief description here to make the paper self-contained.

Simple Metrics. As a starting example, consider the `fac` function

```
fac :: n:Nat -> Nat / [n]
fac 0 = 1
fac n = n * fac (n-1)
```

The termination metric is simply the parameter `n`; as `n` is non-negative and decreases at the recursive call, LIQUIDHASKELL verifies that `fac` will terminate. We specify the termination metric in the type signature with the `/[n]`.

Termination checking is performed at the same time as regular type checking, as it can be reduced to refinement type checking with a special terminating fixpoint combinator [**?**]. Thus, if LIQUIDHASKELL fails to prove that a given termination metric is well-formed and decreasing, it will report a `Termination Check Error`. At this point, the user can either debug the specification, or mark the function as non-terminating.

Termination Expressions. Sometimes, no single parameter decreases across recursive calls, but there is some *expression* that forms the decreasing metric. For example recall `range lo hi` (from § **??**) which returns the list of `Int`s from `lo` to `hi`:

```
range lo hi
  | lo < hi   = lo : range (lo+1) hi
  | otherwise = []
```

Here, neither parameter is decreasing (indeed, the first one is increasing) but `hi-lo` decreases across each call. To account for such cases, we can specify as the termination metric a (refinement logic) expression over the function parameters. Thus, to prove termination, we could type `range` as:

```
lo:Int -> hi:Int -> [(Btwn lo hi)] / [hi-lo]
```

Lexicographic Termination. The Ackermann function

```
ack m n
  | m == 0    = n + 1
  | n == 0    = ack (m-1) 1
  | otherwise = ack (m-1) (ack m (n-1))
```

is curious as there exists no simple, natural-valued, termination metric that decreases at each recursive call. However `ack` terminates because at each call *either* `m` decreases *or* `m` remains the same and `n` decreases. In other words, the pair `(m,n)` strictly decreases according to a *lexicographic* ordering. Thus LIQUIDHASKELL supports termination metrics that are a *sequence of* termination expressions. For example, we can type `ack` as:

```
ack :: m:Nat -> n:Nat -> Nat / [m, n]
```

At each recursive call LIQUIDHASKELL uses a lexicographic ordering to check that the sequence of termination expressions is decreasing (and well-founded in each component).

Mutual Recursion. The lexicographic mechanism lets us check termination of mutually recursive functions, *e.g.,* `isEven` and `isOdd`

```
isEven 0 = True
isEven n = isOdd $ n-1

isOdd n  = not $ isEven n
```

Each call terminates as either `isEven` calls `isOdd` with a decreasing parameter, *or* `isOdd` calls `isEven` with the same parameter, expecting the latter to do the decreasing. For termination, we type:

```
isEven :: n:Nat -> Bool / [n, 0]
isOdd  :: n:Nat -> Bool / [n, 1]
```

To check termination, LIQUIDHASKELL verifies that at each recursive call the metric of the caller is less than the metric of the callee. When `isEven` calls `isOdd`, it proves that the caller's metric, namely `[n,0]` is greater than the callee's `[n-1,1]`. When `isOdd` calls `isEven`, it proves that the caller's metric `[n,1]` is greater than the callee's `[n,0]`, thereby proving the mutual recursion always terminates.

Recursion over Data Types. The above strategies generalize easily to functions that recurse over (finite) data structures like arrays, lists, and trees. In these cases, we simply use *measures* to project the structure onto `Nat`, thereby reducing the verification to the previously seen cases. For example, we can prove that `map`

```
map f (x:xs) = f x : map f xs
map f []     = []
```

terminates, by typing `map` as

```
(a -> b) -> xs:[a] -> [b] / [len xs]
```

i.e.,, by using the measure `len xs`, from § **??**, as the metric.

Generalized Metrics Over Datatypes. In many functions there is no single argument whose measure provably decreases. Consider

```
merge (x:xs) (y:ys)
  | x < y      = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

from the homonymous sorting routine. Here, neither parameter decreases, but the *sum* of their sizes does. To prove termination, we can type `merge` as:

```
xs:[a] -> ys:[a] -> [a] / [len xs + len ys]
```

Putting it all Together. The above techniques can be combined to prove termination of the mutually recursive quick-sort (from [**?**])

```
qsort (x:xs)   = qpart x xs [] []
qsort []       = []

qpart x (y:ys) l r
  | x > y      = qpart x ys (y:l) r
  | otherwise  = qpart x ys l (y:r)
qpart x [] l r = app x (qsort l) (qsort r)

app k []     z = k : z
app k (x:xs) z = x : app k xs z
```

`qsort (x:xs)` calls `qpart x xs` to partition `xs` into two lists `l` and `r` that have elements less and greater or equal than the pivot `x`, respectively. When `qpart` finishes partitioning it mutually recursively calls `qsort` to sort the two list and appends the results with `app`. LIQUIDHASKELL proves sortedness as well [**?**] but let us focus here on termination. To this end, we type the functions as:

```
qsort :: xs:_ -> _
      / [len xs, 0]

qpart :: _ -> ys:_ -> l:_ -> r:_ -> _
      / [len ys + len l + len r, 1 + len ys]
```

As before, LIQUIDHASKELL checks that at each recursive call the caller's metric is less than the callee's. When `qsort` calls `qpart` the length of the unsorted list `len (x:xs)` exceeds the `len xs + len [] + len []`. When `qpart` recursively calls itself the first component of the metric is the same, but the length of the unpartitioned list decreases, *i.e.,* `1 + len y:ys` exceeds `1 + len ys`. Finally, when `qpart` calls `qsort` we have `len ys + len l + len r` exceeds both `len l` and `len r`, thereby ensuring termination.

Automation: Default Size Measures. The `qsort` example illustrates that while LIQUIDHASKELL is very expressive, devising appropriate termination metrics can be tricky. Fortunately, such patterns are very uncommon, and the vast majority of cases in real world programs are just structural recursion on a datatype. LIQUIDHASKELL automates termination proofs for this common case, by allowing users to specify a *default size measure* for each data type, *e.g.,* `len` for `[a]`. Now, if no explicit termination metric is given, by default LIQUIDHASKELL assumes that the *first* argument whose type has an associated size measure decreases. Thus, in the above, we need not specify metrics for `fac` or `map` as the size measure is automatically used to prove termination. This heuristic suffices to *automatically* prove 67% of recursive functions terminating.

Disabling Termination Checking. In `Haskell`'s lazy setting not all functions are terminating. LIQUIDHASKELL provides two mechanisms the disable termination proving. A user can disable checking a single function by marking that function as lazy. For example, specifying `lazy repeat` tells the tool to not prove `repeat` terminates. Optionally, a user can disable termination checking for a whole module by using the command line argument `--no-termination` for the entire file.

9 Functional Correctness Invariants

So far, we have considered a variety of general, application independent correctness criteria. Next, let us see how we can use LIQUIDHASKELL to specify and statically verify critical application specific correctness properties, using two illustrative case studies: red-black trees, and the stack-set data structure introduced in the `xmonad` system.

9.1 Red-Black Trees

Red-Black trees have several non-trivial invariants that are ideal for illustrating the effectiveness of refinement types, and contrasting with existing approaches based on GADTs [**?**]. The structure can be defined via the following Haskell type:

```
data Col    = R | B
data Tree a = Leaf
            | Node Col a (Tree a) (Tree a)
```

However, a `Tree` is a valid Red-Black tree only if it satisfies three crucial invariants:

- ***Order:*** The keys must be binary-search ordered, *i.e.,* the key at each node must lie between the keys of the left and right subtrees of the node,
- ***Color:*** The children of every *red* `Node` must be colored *black*, where each `Leaf` can be viewed as black,
- ***Height:*** The number of black nodes along any path from each `Node` to its `Leafs` must be the same.

Red-Black trees are especially tricky as various operations create trees that can temporarily violate the invariants. Thus, while the above invariants can be specified with singletons and GADTs, encoding all the properties (and the temporary violations) results in a proliferation of data constructors that can somewhat obfuscate correctness. In contrast, with refinements, we can specify and verify the invariants in isolation (if we wish) and can trivially compose them simply by *conjoining* the refinements.

Color Invariant. To specify the color invariant, we define a *black-rooted tree* as:

```
measure isB           :: Tree a -> Prop
color (Node c x l r)  = c == B
color (Leaf)          = true
```

and then we can describe the color invariant simply as:

```
measure isRB         :: Tree a -> Prop
isRB (Leaf)          = true
isRB (Node c x l r) = isRB l && isRB r &&
                       c = R => (isB l && isB r)
```

The insertion and deletion procedures create intermediate *almost* red-black trees where the color invariant may be violated at the root. Rather than create new data constructors we can define almost red-black trees with a measure that just drops the invariant at the root:

```
measure almostRB         :: Tree a -> Prop
almostRB (Leaf)          = true
almostRB (Node c x l r) = isRB l && isRB r
```

Height Invariant. To specify the height invariant, we define a black-height measure:

```
measure bh         :: Tree a -> Int
bh (Leaf)          = 0
bh (Node c x l r) = bh l
                   + if c = R then 0 else 1
```

and we can now specify black-height balance as:

```
measure isBal         :: Tree a -> Prop
isBal (Leaf)          = true
isBal (Node c x l r) = bh l = bh r
                       && isBH l && isBH r
```

Note that `bh` only considers the left sub-tree, but this is legitimate, because `isBal` will ensure the right subtree has the same `bh`.

Order Invariant. Finally, to encode the binary-search ordering property, we parameterize the datatype with abstract refinements:

```
data Tree a <l::a->a->Prop, r::a->a->Prop>
  = Leaf
  | Node { c   :: Col
         , key :: a
         , lt  :: Tree<l,r> a<l key>
         , rt  :: Tree<l,r> a<r key> }
```

Intuitively, `l` and `r` are relations between the root `key` and *each* element in its left and right subtree respectively. Now the alias:

```
type OTree a
  = Tree <{\k v -> v<k}, {\k v -> v>k}> a
```

describes binary-search ordered trees!

Composing Invariants. Finally, we can compose the invariants, and define a Red-Black tree with the alias:

```
type RBT a = {v:OTree a | isRB v && isBal v}
```

An almost Red-Black tree is the above with `isRB` replaced with `almostRB`, *i.e.,* does not require any new types or constructors. If desired, we can ignore a particular invariant simply by replacing the corresponding refinement above with `true`. Given the above – and suitable signatures LIQUIDHASKELL verifies the various insertion, deletion and rebalancing procedures for a Red-Black Tree library.

9.2 Stack Sets in XMonad

`xmonad` is a dynamically tiling `X11` window manager that is written and configured in Haskell. The set of windows managed by XMonad is organized into a hierarchy of types. At the lowest level we have a *set* of windows `a` represented as a `Stack a`

```
data Stack a = Stack { focus :: a
                     , up    :: [a]
                     , down  :: [a] }
```

The above is a zipper [**?**] where `focus` is the "current" window and `up` and `down` the windows "before" and "after" it. Each `Stack` is wrapped inside a `Workspace` that has additional information about layout and naming:

```
data Workspace i l a = Workspace
   { tag    :: i
   , layout :: l
   , stack  :: Maybe (Stack a) }
```

which is in turn, wrapped inside a `Screen`:

```
data Screen i l a sid sd = Screen
  { workspace    :: Workspace i l a
  , screen       :: sid
  , screenDetail :: sd }
```

The set of all screens is represented by the top-level zipper:

```
data StackSet i l a sid sd = StackSet
  { cur :: Screen i l a sid sd
  , vis :: [Screen i l a sid sd]
  , hid :: [Workspace i l a]
  , flt :: M.Map a RationalRect }
```

Key Invariant: Uniqueness of Windows. The key invariant for the `StackSet` type is that each window `a` should appear at most once in a `StackSet i l a sid sd`. That is, a window should *not be duplicated* across stacks or workspaces. Informally, we specify this invariant by defining a measure for the *set of elements* in a list, `Stack`, `Workspace` and `Screen`, and then we use that measure to assert that the relevant sets are disjoint.

Specification: Unique Lists. To specify that the set of elements in a list is unique, *i.e.,* there are no duplicates in the list we first define a measure denoting the set using Z3's [**?**] built-in theory of sets:

```
measure elts :: [a] -> Set a
elts ([])   = emp
elts (x:xs) = cup (sng x) (elts xs)
```

Now, we can use the above to define uniqueness:

```
measure isUniq :: [a] -> Prop
isUniq ([])   =  true
isUniq (x:xs) =  notIn x xs && isUniq xs
```

where `notIn` is an abbreviation:

```
predicate notIn X S = not (mem X (elts S))
```

Specification: Unique Stacks. We can use `isUniq` to define unique, *i.e.,*, duplicate free, `Stacks` as:

```
data Stack a = Stack
 { focus :: a
 , up    :: {v:[a] | Uniq1 v focus}
 , down  :: {v:[a] | Uniq2 v focus up} }
```

using the aliases

```
predicate Uniq1 V X
  = isUniq V  && notIn X V
predicate Uniq2 V X Y
  = Uniq1 V X && disjoint Y V
predicate disjoint X Y
  = cap (elts X) (elts Y) = emp
```

i.e., the field `up` is a unique list of elements different from `focus`, and the field `down` is additionally disjoint from `up`.

Specification: Unique StackSets. It is straightforward to lift the `elts` measure to the `Stack` and the wrapper types `Workspace` and `Screen`, and then correspondingly lift `isUniq` to `[Screen]` and `[Workspace]`. Having done so, we can use those measures to refine the type of `StackSet` to stipulate that there are no duplicates:

```
type UniqStackSet i l a sid sd
  = {v: StackSet i l a sid sd | NoDups v}
```

using the predicate aliases

```
predicate NoDups V
  =  disjoint3 (hid V) (cur V) (vis V)
  && isUniq (vis V)
  && isUniq (hid V)

predicate disjoint3 X Y Z
  =  disjoint X Y
  && disjoint Y Z
  && disjoint X Z
```

LIQUIDHASKELL automatically turns the record selectors of refined data types to measures that return the values of appropriate fields, hence `hid x` (resp. `cur x`, `vis x`) are the values of the `hid`, `cur` and `vis` fields of a `StackSet` named `x`.

Verification. LIQUIDHASKELL uses the above refined type to verify the key invariant, namely, that no window is duplicated. Three key actions of the, eventually successful, verification process can be summarized as follows:

- *Strengthening library functions.* `xmonad` repeatedly concatenates the lists of a Stack. To prove that for some `s:Stack a`, `(up s ++ down s)` is a unique list, the type of `(++)` needs to capture that concatenation of two unique and disjoint lists is a unique list. For verification, we assumed that Prelude's `(++)` satisfies this property. But, not all arguments of `(++)` are unique disjoint lists: `"StackSet"++"error"` is a trivial example that does not satisfy the assumed preconditions of `(++)` thus creating a type error. Currently, LIQUIDHASKELL does not support intersection types, thus we used an unrefined `(++.)` variant of `(++)` for such cases.

- *Restrict the functions' domain.* `modify` is a `maybe`-like function that, given a default value `x`, a function `f`, and a StackSet `s`, applies `f` on the `Maybe (Stack a)` values inside `s`.

  ```
  modify :: x:{v:Maybe (Stack a) | isNothing v}
         -> (y:Stack a
             -> Maybe {v:Stack a | SubElts v y})
         -> UniqStackSet i l a s sd
         -> UniqStackSet i l a s sd
  ```

  Since inside the StackSet `s` each `y:Stack a` could be replaced with either the default value `x` or with `f y`, we need to ensure that both these alternatives will not insert duplicates. This imposes the curious precondition that the default value should be **`Nothing`**.

- *Code inlining* Given a tag `i` and a StackSet `s`, `view i s` will set the current Screen to the screen with tag `i`, if such a screen exists in `s`. Below is the original definition for `view` in case when a screen with tag `i` exists in visible screens

  ```
  view :: (Eq s, Eq i) => i
       -> StackSet i l a s sd
  ```

```
      -> StackSet i l a s sd
view i s
  | Just x <- find ((i==).tag.workspace)
                   (visible s)
  = s { current = x
      , visible = current s
                : deleteBy (equating screen) x
                           (visible s) }
```

Verification of this code is difficult as we cannot suitably type `find`. Instead we *inline* the call to `find` and the field update into a single recursive function `raiseIfVisible i s` that in-place replaces `x` with the current screen.

Finally, `xmonad` comes with an extensive suite of QuickCheck properties, that were formally verified in Coq [**?**]. In future work, it would be interesting to do a similar verification with LIQUIDHASKELL, to compare the refinement types to proof-assistants.

10 Evaluation

We now turn to a quantitative evaluation of our experiments with LIQUIDHASKELL.

Module	Version	LOC	Mod	Fun	Specs	Annot	Qualif	Time (s)
GHC.LIST	7.4.1	309	1	66	29 / 38	6 / 6	0 / 0	15
DATA.LIST	4.5.1.0	504	1	97	15 / 26	6 / 6	3 / 3	11
DATA.MAP.BASE	0.5.0.0	1396	1	180	125 / 173	13 / 13	0 / 0	174
DATA.SET.SPLAY	0.1.1	149	1	35	27 / 37	5 / 5	0 / 0	27
HSCOLOUR	1.20.0.0	1047	16	234	19 / 40	5 / 5	1 / 1	196
XMONAD.STACKSET	0.11	256	1	106	74 / 213	3 / 3	4 / 4	27
BYTESTRING	0.9.2.1	3505	8	569	307 / 465	55 / 55	47 / 124	294
TEXT	0.11.2.3	3128	17	493	305 / 717	52 / 54	49 / 97	499
VECTOR-ALGORITHMS	0.5.4.2	1218	10	99	76 / 266	9 / 9	13 / 13	89
Total		11512	56	1879	977 / 1975	154 / 156	117 / 242	1336

Table 1: A quantitative evaluation of our experiments. **Version** is version of the checked library. **LOC** is the number of non-comment lines of source code as reported by `sloccount`. **Mod** is the number of modules in the benchmark and **Fun** is the number of functions. **Specs** is the number (/ line-count) of type specifications and aliases, data declarations, and measures provided. **Annot** is the number (/ line-count) of other annotations provided, these include invariants and hints for the termination checker. **Qualif** is the number (/ line-count) of provided qualifiers. **Time (s)** is the time, in seconds, required to run LIQUIDHASKELL.

10.1 Results

We have used the following libraries as benchmarks:

- `GHC.List` and `Data.List`, which together implement many standard list operations; we verify various size related properties,
- `Data.Set.Splay`, which implements a splay-tree based functional set data type; we verify that all interface functions terminate and return well ordered trees,
- `Data.Map.Base`, which implements a functional map data type; we verify that all interface functions terminate and return binary-search ordered trees [**?**],

- `HsColour`, a syntax highlighting program for Haskell code, we verify totality of all functions (§ **??**),
- `XMonad`, a tiling window manager for X11, we verify the uniqueness invariant of the core datatype, as well as some of the QuickCheck properties (§ **??**),
- `Bytestring`, a library for manipulating byte arrays, we verify termination, low-level memory safety, and high-level functional correctness properties (§ **??**),
- `Text`, a library for high-performance unicode text processing; we verify various pointer safety and functional correctness properties (§ **??**), during which we find a subtle bug,
- `Vector-Algorithms`, which includes a suite of "imperative" (*i.e.,* monadic) array-based sorting algorithms; we verify the correctness of vector accessing, indexing, and slicing *etc.*.

Table **??** summarizes our experiments, which covered 56 modules totaling 11,512 non-comment lines of source code and 1,975 lines of specifications. The results are on a machine with an Intel Xeon X5660 and 32GB of RAM (no benchmark required more than 1GB.) The upshot is that LIQUIDHASKELL is very effective on real-world code bases. The total overhead due to hints, *i.e.,* the sum of **Annot** and **Qualif**, is 3.5% of **LOC**. The specifications themselves are machine checkable versions of the comments placed around functions describing safe usage and behavior, and required around two lines to express on average. While there is much room for improving the running times, the tool is fast enough to be used interactively, verify a handful of API functions and associated helpers in isolation.

10.2 Limitations

Our case studies also highlighted several limitations of LIQUIDHASKELL that we will address in future work. In most cases, we could alter the code slightly to facilitate verification.

Ghost parameters. are sometimes needed in order to materialize values that are not needed for the computation, but are necessary to prove various specifications. For example, the `piv` parameter in the `append` function for red-black trees (§ **??**).

Fixed-width integer and floating-point numbers. LIQUIDHASKELL uses the theories of linear arithmetic and real numbers to reason about numeric operations. In some cases this causes us to lose precision, *e.g.,* when we have to approximate the behavior of bitwise operations. We could address this shortcoming by using the theory of bit-vectors to model fixed-width integers, but we are unsure of the effect this would have on LIQUIDHASKELL's performance.

Higher-order functions. must sometimes be *specialized* because the original type is not precise enough. For example, the `concat` function that concatenates a list of input `ByteStrings` pre-allocates the output region by computing the total size of the input.

```
len = sum . map length $ xs
```

Unfortunately, the type for `map` is not sufficiently precise to conclude that the value `len` equals `bLens xs`, se we must manually specialize the above into a single recursive traversal that computes the lengths. Rather than complicating the type system with

a very general higher-order type for `map` we suspect the best way forward will be to allow the user to specify inlining in a clean fashion.

Functions as Data. Several libraries like `Text` encode data structures like (finite) streams using functions, in order to facilitate fusion. Currently, it is not possible to describe sizes of these structures using measures, as this requires describing the sizes of input-output chains starting at a given seed input for the function. In future work, it will be interesting to extend the measure mechanism to support multiple parameters (*e.g.,* a stream and a seed) in order to reason about such structures.

Lazy binders. sometimes get in the way of verification. A common pattern in Haskell code is to define *all* local variables in a single **`where`** clause and use them only in a subset of all branches. LIQUIDHASKELL flags a few such definitions as *unsafe*, not realizing that the values will only be demanded in a specific branch. Currently, we manually transform the code by pushing binders inwards to the usage site. This transformation could be easily automated.

Assumes. which can be thought of as "hybrid" run-time checks, had to be placed in a couple of cases where the verifier loses information. One source is the introduction of assumptions about mathematical operators that are currently conservatively modeled in the refinement logic (*e.g.,* that multiplication is commutative and associative). These may be removed by using more advanced non-linear arithmetic decision procedures.

Error messages. are a crucial part of any type-checker. Currently, we report error locations in the provided source file and output the failed constraint(s). Unfortunately, the constraints often refer to intermediate values that have been introduced during the ANF-transformation, which obscures their relation to the program source. In future work, we may attempt to map these intermediate values back to their source expressions, which should increase the comprehensibility of our error messages. Another interesting possibility would be to search for concrete counterexamples when LIQUIDHASKELL detects an invalid constraint.

11 Conclusion

In this report we presented various refinement type systems. We started with type systems where the refinement language expresses arbitrary program expressions. Even though these systems are expressive, the assertions formed can not be statically verified. To reason in such systems, we presented two alternatives: interactive theorem proving, where the user should provide explicit proofs, and contracts calculi, where the assertions are verified at runtime. Next we presented refinement type system which restrict the refinement language, so as to render type checking decidable. As an example, we presented Liquid Types, in which the refinement language is restricted according to a finite set of qualifiers and allows not only decidable verification, but also automatic type inference. Then, we presented Abstract Refinement Types, which can be used in a refinement type system to enhance expressiveness without increasing complexity. Then, we present LIQUIDHASKELL that combines liquidTypes with abstraction over refinements to enhance expressiveness of LiquidTypes. LIQUIDHASKELL is a quite expressive verification tool for Haskell programs that can be used to check termination, totality and general functional correctness. Finally, we evaluate LIQUIDHASKELL in real world Haskell libraries.