Bounded Refinement Types

Niki Vazou  Alexander Bakst  Ranjit Jhala
UC San Diego

Abstract

We present a notion of bounded quantification for refinement types and show how it expands the expressiveness of refinement typing by using it to develop typed combinators for: (1) relational algebra and safe database access, (2) Floyd-Hoare logic within a state transformer monad equipped with combinators for branching and looping, and (3) using the above to implement a refined IO monad that tracks capabilities and resource usage. This leap in expressiveness comes via a translation to “ghost” functions, which lets us retain the automated and decidable SMT based checking and inference that makes refinement typing effective in practice.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]; D.3.3 [Language Constructs and Features]; Polymorphism; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying Reasoning about Programs

Keywords  haskell, refinement types, abstract interpretation

1. Introduction

Must program verifiers always choose between expressiveness and automation? On the one hand, tools based on higher order logics and full dependent types impose no limits on expressiveness, but require user-provided (perhaps, tactic-based) proofs. On the other hand, tools based on Refinement Types [22, 30] trade expressiveness for automation. For example, the refinement types

```haskell
  type Pos = {v: Int | /zero.noslash < v}
  type IntGE x = {v: Int | x ≤ v}
```

specify subsets of Int corresponding to values that are positive or larger than some other value x respectively. By limiting the refinement predicates to SMT-decidable logics [17], refinement type based verifiers eliminate the need for explicit proof terms, and thus automate verification.

This high degree of automation has enabled the use of refinement types for a variety of verification tasks, ranging from array bounds checking [21], termination and totality checking [29], protocol validation [2, 9], and securing web applications [10]. Unfortunately, this automation comes at a price. To ensure predictable and decidable type checking, we must limit the logical formulas appearing in specification types to decidable (typically quantifier free) first order theories, thereby precluding higher-order specifications that are essential for modular verification.

In this paper, we introduce Bounded Refinement Types which reconcile expressible higher order specifications with automatic SMT based verification. Our approach comprises two key ingredients. Our first ingredient is a mechanism, developed by [27], for abstracting refinements over type signatures. This mechanism is the analogue of parametric polymorphism in the refinement setting: it increases expressiveness by permitting generic signatures that are universally quantified over the (concrete) refinements that hold at different call-sites. However, we observe that for modular verification, we additionally need to constrain the abstract refinement parameters, typically to specify fine grained dependencies between the parameters. Our second ingredient provides a technique for enriching function signatures with subtyping constraints (or bounds) between abstract refinements that must be satisfied by the concrete refinements at instantiation. Thus, constrained abstract refinements are the analogue of bounded quantification in the refinement setting and in this paper, we show that this simple technique proves to be remarkably effective.

• First, we demonstrate via a series of short examples how bounded refinements enable the specification and verification of diverse textbook higher order abstractions that were hitherto beyond the scope of decidable refinement typing (§ 2).

• Second, we formalize bounded types and show how bounds are translated into “ghost” functions, reducing type checking and inference to the “unbounded” setting of [27], thereby ensuring that checking remains decidable. Furthermore, as the bounds are Horn constraints, we can directly reuse the abstract interpretation of Liquid Typing [21] to automatically infer concrete refinements at instantiation sites (§ 3).

• Third, to demonstrate the expressiveness of bounded refinements, we use them to build a typed library for extensible dictionaries, to then implement a relational algebra library on top of those dictionaries, and to finally build a library for type-safe database access (§ 4).

• Finally, we use bounded refinements to develop a Refined State Transformer monad for stateful functional programming, based upon Filliâtre’s method for indexing the monad with pre- and post-conditions [8]. We use bounds to develop branching and looping combinators whose types signatures capture the derivation rules of Floyd-Hoare logic, thereby obtaining a library for writing verified stateful computations (§ 5). We use this library to develop a refined IO monad that tracks capabilities at a fine-

---

* This work was supported by NSF grants CCF-1422471, C1223850, CCF-1218344, a Microsoft Research Ph.D Fellowship and a generous gift from Microsoft Research.
We have implemented Bounded Refinement Types in LIQUID-HASKELL [29]. The source code of the examples (with slightly more verbose concrete syntax) is at [24]. While the construction of these verified abstractions is possible with full dependent types, bounded refinements keep checking automatic and decidable, use abstract interpretation to automatically synthesize refinements (i.e., pre- and post-conditions and loop invariants), and most importantly enable retroactive or gradual verification as when erase the refinements, we get valid programs in the host language (§ 7). Thus, bounded refinements point a way towards keeping our automation, and perhaps having expressiveness too.

2. Overview

We start with a high level overview of bounded refinement types. To make the paper self contained, we begin by recalling the notions of abstract refinement types. Next, we introduce bounded refinements, and show how they permit modular higher-order specifications. Finally, we describe how they are implemented via an elaboration process that permits automatic first-order verification.

2.1 Preliminaries

Refinement Types let us precisely specify subsets of values, by conjointing base types with logical predicates that constrain the values. We get decidability of type checking, by limiting these predicates to decidable, quantifier-free, first-order logics, including the theory of linear arithmetic, uninterpreted functions, arrays, bit-vectors and so on. Apart from subsets of values, like the Pos and IntGE that we saw in the introduction, we can specify contracts like pre- and post-conditions by suitably refining the input and output types of functions.

Preconditions are specified by refining input types. We specify that the function assert must only be called with True, where the refinement type True contains only the singleton True:

- type True = {v:Bool | v ↔ True}
- assert :: TRUE → a → a
- assert True x = x
- assert False _ = error "Provably Dead Code"

We can specify post-conditions by refining output types. For example, a primitive Int comparison operator leq can be assigned a type that says that the output is True iff the first input is actually less than or equal to the second:

- leq :: x:Int → y:Int → {v:Bool | v ↔ x ≤ y}

Refinement Type Checking proceeds by checking that at each application, the types of the actual arguments are subtypes of those of the function inputs, in the environment (or context) in which the call occurs. Consider the function:

- checkGE :: a:Int → b:IntGE a → Int
- checkGE a b = assert cmp b
  where cmp = a 'leq' b

To verify the call to assert we check that the actual parameter cmp is a subtype of True, under the assumptions given by the input types for a and b. Via subtyping [29] the check reduces to establishing the validity of the verification condition (VC)

\[ a \leq b \Rightarrow (\text{cmp} \leftrightarrow a \leq b) \Rightarrow v \equiv \text{cmp} \Rightarrow (v \equiv \text{true}) \]

The first antecedent comes from the input type of b, the second from the type of cmp obtained from the output of leq, the third from the actual input passed to assert, and the goal comes from the input type required by assert. An SMT solver [17] readily establishes the validity of the above VC, thereby verifying checkGE.

First order refinements prevent modular specifications. Consider the function that returns the largest element of a list:

- maximum :: List Int → Int
  maximum [x] = x
  maximum (x:xs) = max x (maximum xs)
  where max a b = if a < b then b else a

How can one write a first-order refinement type specification for maximum that will let us verify the below code?

- posMax :: List Pos → Pos
- posMax = maximum

Any suitable specification would have to enumerate the situations under which maximum may be invoked breaking modularity.

Abstract Refinements overcome the above modularity problems [27]. The main idea is that we can type maximum by observing that it returns one of the elements in its input list. Thus, if every element of the list enjoys some refinement p then the output value is also guaranteed to satisfy p. Concretely, we can type the function as:

- maximum :: ∀〈p::Int→Bool>. List Int<〈p〉→ Int<〈p〉

where informally, Int<〈p〉 stands for (v:Int | p v), and p is an uninterpreted function in the refinement logic [17]. The signature states that for any refinement p on Int, the input is a list of elements satisfying p and returns as output an integer satisfying p. In the sequel, we will drop the explicit quantification of abstract refinements; all free abstract refinements will be implicitly quantified at the top-level (as with classical type parameters.)

Abstract Refinements Preserve Decidability. Abstract refinements do not require the use of higher-order logics. Instead, abstractly refined signatures (like maximum) can be verified by viewing the abstract refinements p as uninterpreted functions that only satisfy the axioms of congruence, namely:

- ∀ x y. x = y ⇒ p x ⇔ p y

As the quantifier free theory of uninterpreted functions is decidable [17], abstract refinement type checking remains decidable [27].

Abstract Refinements are Automatically Instantiated at call-sites, via the abstract interpretation framework of Liquid Typing [27]. Each instantiation yields fresh refinement variables on which subtyping constraints are generated; these constraints are solved via abstract interpretation yielding the instantiations. Hence, we verify posMax by instantiating:

- p → λ v → 0 < v -- at posMax

2.2 Bounded Refinements

Even with abstraction, refinement types hit various expressiveness walls. Consider the following example from [26]. find takes as input a predicate q, a continuation k and a starting number i; it proceeds to compute the smallest Int (larger than i) that satisfies q, and calls k with that value. exl passes Find a continuation that checks that the “found” value equals or exceeds n.
Can Abstract Refinements Help? Let's try to abstract over the defined and so the type for \( k \) as there is no way to specify that Verification fails as there is no way to specify that \( k \) is only called with arguments greater than \( n \). First, the variable \( n \) is not in scope at the function definition and so we cannot refer to it. Second, we could try to say that \( k \) is invoked with values greater than or equal to \( i \), which gets substituted with \( n \) at the call-site. Alas, due to the currying order, \( i \) too is not in scope at the point where \( k \)'s type is defined and so the type for \( k \) cannot depend upon \( i \).

Can Abstract Refinements Help? Let's try to abstract over the refinement that \( i \) enjoys, and assign \( \text{find} \) the type: 
\[
(\text{Int} \to \text{Bool}) \to (\text{Int}<p> \to \text{a}) \to \text{Int}<p> \to \text{a}
\]
which states that for any refinement \( p \), the function takes an input \( i \) which satisfies \( p \) and hence that the continuation is also only invoked on a value which trivially enjoys \( p \), namely \( i \). At the call-site in \( \text{ex1} \) we can instantiate
\[
p \mapsto \lambda v \mapsto n \leq v \tag{1}
\]
This instantiated refinement is satisfied by the parameter \( n \), and sufficient to verify, via function subtyping, that \( \text{checkGE} \) \( n \) will only be called with values satisfying \( p \), and hence larger than \( n \).

**find is ill-typed** as the signature requires that at the recursive call site, the value \( i+1 \) also satisfies the abstract refinement \( p \). While this holds for the example we have in mind (1), it does not hold for all \( p \), as required by the type of \( \text{find} \)! Concretely, \( \{v: \text{Int}|v=i+1\} \) is in general not a subtype of \( \text{Int}<p> \), as the associated VC
\[
\ldots \Rightarrow p \ i \Rightarrow p \ (i+1) \tag{2}
\]
is invalid – the type checker thus (soundly!) rejects \( \text{find} \).

**We must Bound the Quantification** of \( p \) to limit it to refinements satisfying some constraint, in this case that \( p \) is upward closed. In the dependent setting, where refinements may refer to program values, bounds are naturally expressed as constraints between refinements. We define a bound, \( \text{UpClosed} \) which states that \( p \) is a refinement that is upward closed, i.e., satisfies \( \forall x \ . \ p \ x \Rightarrow p \ (x+1) \), and use it to type \( \text{find} \) as:
\[
\begin{align*}
\text{bound UpClosed} \ (p :: \text{Int} \to \text{Bool}) &= \lambda x \mapsto p \ x \Rightarrow p \ (x+1) \\
\text{find} :: \text{(UpClosed} \ p) \Rightarrow (\text{Int} \to \text{Bool}) \\
&\quad \Rightarrow (\text{Int}<p> \to \text{a}) \\
&\quad \Rightarrow \text{Int}<p> \to \text{a}
\end{align*}
\]
This time, the checker is able to use the bound to verify the VC (2). We do so in a way that refinements (and thus VC's) remain quantifier free and hence, SMT decidable (3).

At the call to \( \text{find} \) in the body of \( \text{ex1} \), we perform the instantiation (1) which generates the additional VC \( n \leq x \Rightarrow n \leq x+1 \) by plugging in the concrete refinements to the bound constraint. The SMT solver easily checks the validity of the VC and hence this instantiation, thereby statically verifying \( \text{ex1} \), i.e., that the assertion inside \( \text{checkGE} \) cannot fail.

### 2.3 Bounds for Higher-Order Functions

Next, we show how bounds expand the scope of refinement typing by letting us write precise modular specifications for various canonical higher-order functions.

**Function Composition**

First, consider \( \text{compose} \). What is a modular specification for \( \text{compose} \) that would let us verify that \( \text{ex2} \) enjoys the given specification?
\[
\begin{align*}
\text{compose} \ f \ g \ x &= f \ (g \ x) \\
\text{type Plus} \ x \ y &= \{v: \text{Int}|v = x + y\} \\
\text{ex2} :: n: \text{Int} \to \text{Plus} \ n \ 2 \\
\text{ex2} &= \text{incr} \ 'compose' \ \text{incr} \\
\text{incr} :: n: \text{Int} \to \text{Plus} \ n \ 1 \\
\text{incr} \ n &= n + 1
\end{align*}
\]

The challenge is to chain the dependencies between the input and output of \( g \) and the input and output of \( f \) to obtain a relationship between the input and output of the composition. We can capture the notion of chaining in a bound:
\[
\begin{align*}
\text{bound Chain} \ p \ q \ r &= \lambda x \ y \ z \mapsto q \ y \ z \Rightarrow p \ x \ y \ r \ x \ z \\
\text{ex2} \ &= \text{compose} \ \text{ex2} \\
\text{compose} :: \ (\text{Chain} \ p \ q \ r) \Rightarrow (y:b \mapsto c<q y>) \\
&\quad \Rightarrow (x:a \mapsto b<q x>) \\
&\quad \Rightarrow (w:a \mapsto c<r w>)
\end{align*}
\]

**To verify** \( \text{ex2} \) we instantiate, at the call to \( \text{compose} \),
\[
\begin{align*}
p, q &\mapsto \lambda x \ v \mapsto v \Rightarrow x + 1 \\
r &\mapsto \lambda x \ v \mapsto v \Rightarrow x + 2
\end{align*}
\]
The above instantiation satisfies the bound, as shown by the validity of the VC derived from instantiating \( p, q, \) and \( r \) in \( \text{Chain} \):
\[
y = x + 1 \Rightarrow z = y + 1 \Rightarrow z = x + 2
\]
and hence, we can check that \( \text{ex2} \) implements its specified type.

**List Filtering**

Next, consider the list \( \text{filter} \) function. What type signature for \( \text{filter} \) would let us check \( \text{positives} \)?
\[
\begin{align*}
\text{filter} \ q \ (x:xs) &= \begin{cases}
q \ x &\mapsto x : \text{filter} \ q \ xs \\
\text{otherwise} &\mapsto \text{filter} \ q \ xs
\end{cases} \\
\text{filter} \ _ \ [] &= [] \\
\text{positives} :: [\text{Int}] &\Rightarrow [\text{Pos}] \\
\text{positives} &= \text{filter} \ \text{isPos} \\
\text{isPos} \ x &= x < x
\end{align*}
\]
Such a signature would have to relate the \( \text{Bool} \) returned by \( f \) with the property of the \( x \) that it checks for. Typed Racket’s latent predicates [25] account for this idiom, but are a special construct limited to \( \text{Bool} \)-valued “type” tests, and not arbitrary invariants. Another approach is to avoid the so-called “Boolean Blindness” that accompanies \( \text{filter} \) by instead using option types and \( \text{mapMaybe} \).

**We overcome blindness using a witness**
\[
\begin{align*}
\text{bound Witness} \ p \ w &= \lambda x \ b \mapsto b \Rightarrow w \ x \ b \Rightarrow p \ x \\
\end{align*}
\]
which says that \( w \) witnesses the refinement \( p \). That is, for any boolean \( b \) such that \( w \ x \ b \) holds, if \( b \) is \text{True} then \( p \ x \) also holds.
Filter can be given a type. Saying that the output values enjoy a refinement \( p \) as long as the test predicate \( q \) returns a boolean witnessing \( p \):

\[
\text{filter} :: (\text{Witness } p \ w) \Rightarrow (x:a \rightarrow \text{Bool} < w \ x) \\
\rightarrow \text{List} \ a \\
\rightarrow \text{List} \ a < p
\]

To verify positives, we infer the following type and instantiations for the abstract refinements \( p \) and \( w \) at the call to \text{filter}:

\[
is\text{Pos} :: x: \text{Int} \rightarrow (v: \text{Bool} | v \leftrightarrow 0 < x) \\
p \mapsto \lambda v \rightarrow 0 < v \\
w \mapsto \lambda x b \rightarrow b \leftrightarrow 0 < x
\]

List Folding

Next, consider the list fold-right function. Suppose we wish to prove the following type for \text{ex3}:

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List} \ a \rightarrow b \\
\text{foldr} \ op \ b \ [x:]xs = x \mapsto op \ \text{foldr} \ op \ b \ xs \\
ex3 :: \text{xs:List} \ a \rightarrow (v: \text{Int} | v = len \ xs) \\
ex3 = \text{foldr} \ (\lambda_\_ \rightarrow \text{incr}) \ 0
\]

where \text{len} is a logical or measure function used to represent the number of elements of the list in the refinement logic [29]:

\[
\text{measure len :: List} \ a \rightarrow \text{Nat} \\
\text{len} \ [\_] = 0 \\
\text{len} \ (x:]xs) = 1 + \text{len} \ xs
\]

We specify induction as a bound. Let (1) \text{inv} be an abstract refinement relating a list \( xs \) and the result \( b \) obtained by folding over it, and (2) \text{step} be an abstract refinement relating the inputs \( x, b \) and output \( b' \) passed to and obtained from the accumulator \( op \) respectively. We state that \text{inv} is closed under \text{step} as:

\[
\text{bound Inductive} \ \text{inv} \ \text{step} = \lambda x b b' \rightarrow \text{inv} \ xs \ b \Rightarrow \text{step} \ x b \ b' \Rightarrow \text{inv} \ (x:]xs) \ b'
\]

We can give \text{foldr} a type. That says that the function \text{outputs} a value that is built inductively over the entire \text{input} list:

\[
\text{foldr} :: (\text{Inductive} \ \text{inv} \ \text{step}) \\
\Rightarrow (x:a \rightarrow \text{acc} \ b \rightarrow b < \text{step} \ x \ \text{acc}) \\
\rightarrow \text{xs:List} \ a \\
\rightarrow b < \text{inv} \ xs
\]

That is, for any invariant \text{inv} that is inductive under \text{step}, if the initial value \( b \) is \text{inv}-related to the empty list, then the folded output is \text{inv}-related to the input list \( xs \).

We verify \text{ex3} by inferring, at the call to \text{foldr}:

\[
\text{inv} \Rightarrow \lambda x s v \rightarrow v = \text{len} \ xs \\
\text{step} \Rightarrow \lambda x b b' \rightarrow b' = b + 1
\]

The SMT solver validates the VC obtained by plugging the above into the bound. Instantiating the signature for \text{foldr} yields precisely the output type desired for \text{ex3}.

Previously, [27] describes a way to type \text{foldr} using abstract refinements that required the operator \text{op} to have one extra ghost argument. Bounds let us express induction without ghost arguments.

2.4 Implementation

To implement bounded refinement typing, we must solve two problems. Namely, how do we (1) \text{check}, and (2) \text{use} functions with bounded signatures? We solve both problems via a unifying insight inspired by the way typeclasses are implemented in Haskell.

1. A \text{Bound Specifies} a function type whose inputs are unconstrained, and whose output is some value that carries the refinement corresponding to the bound’s body.

2. A \text{Bound Is Implemented} by a ghost function that returns \text{true}, but is defined in a context where the bound’s constraint holds when instantiated to the concrete refinements at the context.

We elaborate bounds into ghost functions satisfying the bound’s type. To check bounded functions, we need to call the ghost function to materialize the bound constraint at particular values of interest. Dually, to use bounded functions, we need to create ghost functions whose outputs are guaranteed to satisfy the bound constraint. This elaboration reduces \text{bounded} refinement typing to the simpler problem of \text{unbounded} abstract refinement typing [27]. The formalization of our elaboration is described in § 3. Next, we illustrate the elaboration by explaining how it addresses the problems of checking and using bounded signatures like \text{compose}.

We Translate Bounds into Function Types called the bound-type where the inputs are unconstrained, and the outputs satisfy the bound’s constraint. For example, the bound \text{Chain} used to type \text{compose} in § 2.3 corresponds to a function type, yielding the translated type for \text{compose}:

\[
\text{type ChainTy} \ p q r \\
= x: a \rightarrow y:b \rightarrow z: c \\
\Rightarrow (v: \text{Bool} \ | \ v x y \Rightarrow p y z \Rightarrow r x z)
\]

\text{compose :: (ChainTy} p q r) \rightarrow (y:b \rightarrow c < p y) \\
\Rightarrow (x:a \rightarrow b < q x) \\
\rightarrow (w:a \rightarrow c < r w)
\]

To Check Bounded Functions we view the bound constraints as extra (ghost) function parameters (cf. type class dictionaries), that satisfy the bound-type. Crucially, each expression where a subtyping constraint would be generated by plain refinement typing is wrapped with a “call” to the ghost to materialize the constraint at values of interest. For example we elaborate \text{compose} into:

\[
\text{compose}$\_\text{chain} f g x = \\
\text{let} \ t1 = g x \\
t2 = f t1 \\
_ = \text{chain} \ x t1 t2 \quad -- \text{materialize} \\
in \ t2$
\]

In the elaborated version $\_\text{chain}$ is the ghost parameter corresponding to the bound. As is standard [21], we perform ANF-conversion to name intermediate values, and then wrap the function output with a call to the ghost to materialize the bound’s constraint. Consequently, the output of \text{compose}, namely \text{t2}, is checked to be a subtype of the specified output type, in an environment strengthened with the bound’s constraint instantiated at \( x, t1 \) and \( t2 \). This subtyping reduces to a quantifier free VC:

\[
q x t1 \Rightarrow p t1 t2 \\
\Rightarrow (q x t1 \Rightarrow p t1 t2 \Rightarrow r x t2) \\
\Rightarrow v \Rightarrow t2 \Rightarrow r x v
\]

whose first two antecedents are due to the types of \text{t1} and \text{t2} (via the output types of \text{g} and \text{f} respectively), and the third comes from
the call to $\texttt{compose}$. The output value $v$ has the singleton refinement that states it equals to $t_2$, and finally the VC states that the output value $v$ must be related to the input $x$ via $r$. An SMT solver validates this decidable VC easily, thereby verifying $\texttt{compose}$.

Our elaboration inserts materialization calls for all variables (of the appropriate type) that are in scope at the given point. This could introduce $\ell n^2$ calls where $k$ is the number of parameters in the bound and $n$ the number of variables in scope. In practice (e.g., in compose) this number is small (e.g., 1) since we limit ourselves to variables of the appropriate types.

To preserve semantics we ensure that none of these materialization calls can diverge, by carefully constraining the structure of the arguments that instantiate the ghost functional parameters.

**At Uses of Bounded Functions** our elaboration uses the bound-type to create lambdas with appropriate parameters that just return true. For example, $\texttt{ex2}$ is elaborated to:

$$\texttt{ex2} = \texttt{compose} \left( \lambda x. \_ \_ \rightarrow \texttt{true} \right) \texttt{incr} \texttt{incr}$$

This elaboration seems too naïve to be true: how do we ensure that the function actually satisfies the bound type?

Happily, that is automatically taken care of by function subtyping. Recalling the translated type for $\texttt{compose}$, the elaborated lambda $(\lambda x. \_ \_ \rightarrow \texttt{true})$ is constrained to be a subtype of $\text{ChainTy} p \; q \; r$. In particular, given the call site instantiation

\[
p \rightarrow Ay \; z \rightarrow z \Rightarrow y + 1 \\
q \rightarrow \lambda x \; y \rightarrow y \Rightarrow x + 1 \\
r \rightarrow \lambda x \; z \rightarrow z \Rightarrow x + 2
\]

this subtyping constraint reduces to the quantifier-free VC:

\[
\begin{align*}
\Gamma \Rightarrow \text{true} & \Rightarrow (z \Rightarrow y + 1) \Rightarrow (y \Rightarrow x + 1) \\
& \Rightarrow (z \Rightarrow x + 2) \quad (3)
\end{align*}
\]

where $\Gamma$ contains assumptions about the various binders in scope. The VC is easily proved valid by an SMT solver, thereby verifying the subtyping obligation defined by the bound, and hence, that $\texttt{ex2}$ satisfies the given type.

### 3. Formalism

Next we formalize Bounded Refinement Types by defining a core calculus $\lambda B$ and showing how it can be reduced to $\lambda P$, the core language of Abstract Refinement Types [27]. We start by defining the syntax (§ 3.1) and semantics (§ 3.2) of $\lambda P$ and the syntax of $\lambda B$ (§ 3.3). Next, we provide a translation from $\lambda B$ to $\lambda P$ (§ 3.4). Then, we prove soundness by showing that our translation is semantics preserving (§ 3.5). Finally, we describe how type inference remains decidable in the presence of bounded refinements (§ 3.6).

#### 3.1 Syntax of $\lambda P$

We build our core language on top of $\lambda B$, the language of Abstract Refinement Types [27]. Figure 1 summarizes the syntax of $\lambda P$, a polymorphic $\lambda$-calculus extended with abstract refinements.

**Expressions** of $\lambda P$ include the usual variables $x$, primitive constants $c$, $\lambda$-abstraction $\lambda x. t.e$, application $e \; c$, let bindings let $x.t = e$ in $e$, type abstraction $\Lambda \alpha. e$, and type application $e[\ell]$. (We add let-binders to $\lambda B$ from [27] as they can be reduced to $\lambda$-abstractions in the usual way.) The parameter $\ell$ in the type application is a refinement type, as described shortly. Finally, $\lambda P$ includes refinement abstraction $\Lambda \pi : t.e$, which introduces a refinement variable $\pi$ (with its type $t$), which can appear in refinements inside $e$, and the corresponding refinement application $e[\ell] \phi$ that substitutes an abstract refinement with the parametric refinement $\phi$, i.e., refinements $\pi$ closed under lambda abstractions.

#### Expressions of $\lambda P$

\[
\begin{align*}
\text{Expressions} & \quad e ::= \ x \ | \ c \ | \ \lambda x. t.e \ | \ e \; e \ | \ \text{let } x.t = e \ in \ e \ | \ \Lambda \alpha. e \ | \ e[\ell] \\
\text{Constants} & \quad c ::= \ \text{true} \ | \ \text{false} \ | \ \text{crash} \\
\text{Predicates} & \quad p ::= \ c \ | \ \neg p \ | \ p \; p \ | \ \ldots \\
\text{Atomic Refinements} & \quad a ::= \ p \ | \ \pi \ \tau \\
\text{Refinements} & \quad r ::= \ a \ | \ a \; a \ | \ a \Rightarrow r \\
\text{Basic Types} & \quad b ::= \ \text{Int} \ | \ \text{Bool} \ | \ \alpha \\
\text{Types} & \quad t ::= \ \{ v : b \mid r \} \ | \ \{ v : (x : t) \Rightarrow t \mid r \} \\
\text{Bounded Types} & \quad \rho ::= \ t \\
\text{Schemata} & \quad \sigma ::= \ p \ | \ \forall \alpha. \sigma \ | \ \forall \pi : t. \sigma
\end{align*}
\]

**Figure 2.** Extending Syntax of $\lambda P$ to $\lambda B$

**The Primitive Constants** of $\lambda B$ include $\text{true}$, $\text{false}$, $0$, $1$, etc. In addition, we include a special untypable constant $\text{crash}$ that models “going wrong”. Primitive operations return a crash when invoked with inputs outside their domain, e.g., when $r$ is invoked with 0 as the divisor, or when an $\text{assert}$ is applied to false.

**Atomic Refinements** $a$ are either concrete or abstract refinements. A concrete refinement $p$ is a boolean valued expression (such as a constant, negation, equality, etc.) drawn from a strict subset of the language of expressions which includes only terms that (a) neither diverge nor crash, and (b) can be embedded into an SMT decidable refinement logic including the quantifier free theory of linear arithmetic and uninterpreted functions [29]. An abstract refinement $\pi \ \tau$ is an application of a refinement variable $\pi$ to a sequence of program variables. A refinement $r$ is either a conjunction or implication of atomic refinements. To enable inference, we only allow implications to appear within bounds $\phi$ (§ 3.6).

**The Types of $\lambda P$** written $t$ include basic types, dependent functions, and schemata quantified over type and refinement variables $\alpha$ and $\pi$ respectively. A basic type is one of $\text{Int}$, $\text{Bool}$, or a type variable $\alpha$. A refined type $t$ is either a refined basic type $\{ v : b \mid r \}$, or a dependent function type $\{ v : (x : t) \Rightarrow t \mid r \}$ where the parameter $x$ can appear in the refinements of the output type. (We include refinements for functions, as refined type variables can be replaced by function types. However, typechecking ensures these refinements are trivially true.) In $\lambda B$ bounded types $\rho$ are just a synonym for types $\ell$. Finally, schemata are obtained by quantifying bounded types over type and refinement variables.

#### 3.2 Semantics of $\lambda P$

Figure 2 summarizes the static semantics of $\lambda P$ as described in [22]. Unlike [27] that syntactically separates concrete ($p$) from abstract ($\pi \ \tau$) refinements, here, for simplicity, we merge both concrete and abstract refinements to atomic refinements $a$.

**A type environment** $\Gamma$ is a sequence of type bindings $x : \sigma$. We use environments to define three kinds of judgments:
Well-Formedness

\[\Gamma, v : b \vdash r : \text{Bool} \quad \text{WF-BASE} \]
\[\Gamma \vdash \{v : b \mid r\} \quad \text{WF-FUN} \]
\[\Gamma, \pi : t \vdash \sigma \quad \text{WF-ABS-\pi} \]
\[\Gamma \vdash \forall \pi : t.\sigma \quad \text{WF-ABS-\alpha} \]

Subtyping

\[(\lbrack [\Gamma] \Rightarrow [\rho_1] \Rightarrow [\rho_2] \rbrack)\text{ is valid} \]
\[\Gamma \vdash \{v : b \mid r_1\} \preceq \{v : b \mid r_2\} \quad \text{\preceq-BASE} \]
\[\Gamma, \pi : t \vdash \sigma_1 \preceq \sigma_2 \quad \text{\preceq-RVAR} \]
\[\Gamma \vdash \forall \pi : t.\sigma_1 \preceq \forall \pi : t.\sigma_2 \quad \text{\preceq-POLY} \]

Type Checking

\[\Gamma \vdash e : \sigma_1 \quad \Gamma \vdash e : \sigma_2 \quad \Gamma, \pi : t \vdash \sigma_1 \preceq \sigma_2 \quad \Gamma \vdash \sigma \quad \text{T-SUB} \]
\[\Gamma \vdash \text{T-LET} \quad \Gamma \vdash e : \sigma \quad \Gamma \vdash e : \sigma \quad \Gamma, x : t \vdash e : t \quad \Gamma \vdash \forall \pi : t.\sigma \quad \text{T-APP} \]
\[\Gamma \vdash \text{T-APP} \quad \Gamma \vdash e_1 : (x : t_1) \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1 \quad \Gamma \vdash e : t \quad \text{T-CONV} \]
\[\Gamma \vdash \text{T-CONV} \quad \Gamma \vdash \forall \pi : t.\sigma \quad \Gamma \vdash \sigma : \tau \quad \Gamma \vdash \lambda x : t_1.\, e : (x : t_2) \rightarrow t \quad \text{T-PINST} \]
\[\Gamma \vdash \text{T-PINST} \quad \Gamma, \pi : t \vdash \epsilon : \sigma \quad \Gamma \vdash \forall \pi : t.\sigma \quad \Gamma \vdash \forall \alpha.\epsilon : \forall \alpha.\sigma \quad \text{T-PGEN} \]
\[\Gamma \vdash \text{T-PGEN} \quad \Gamma \vdash e : \forall \alpha.\sigma \quad \Gamma \vdash \forall \alpha.\epsilon : \forall \alpha.\sigma \quad \text{T-INST} \]

![Figure 3. Static Semantics: Well-formedness, Subtyping and Type Checking](image)

- **Well-formedness judgments** \(\Gamma \vdash \sigma\) state that a type schema \(\sigma\) is well-formed under environment \(\Gamma\). That is, the judgment states that the refinements in \(\sigma\) are boolean expressions in the environment \(\Gamma\).

- **Subtyping judgments** \(\Gamma \vdash \sigma_1 \preceq \sigma_2\) state that the type schema \(\sigma_1\) is a subtype of the type schema \(\sigma_2\) under environment \(\Gamma\). That is, the judgment states that when the free variables of \(\sigma_1\) and \(\sigma_2\) are bound to values described by \(\Gamma\), the values described by \(\sigma_1\) are a subset of those described by \(\sigma_2\).

- **Typing judgments** \(\Gamma \vdash e : \sigma\) state that the expression \(e\) has the type schema \(\sigma\) under environment \(\Gamma\). That is, the judgment states that when the free variables in \(e\) are bound to values described by \(\Gamma\), the expression \(e\) will evaluate to a value described by \(\sigma\).

**The Well-formedness rules** check that the concrete and abstract refinements are indeed boolean-valued expressions in the appropriate environment. The key rule is **WF-BASE**, which checks that the refinement \(\tau\) is boolean.

**The Subtyping rules** stipulate when the set of values described by schema \(\sigma_1\) is subsumed by (i.e., contained within) the values described by \(\sigma_2\). The rules are standard except for **\preceq-POLY**, which reduces subtyping of basic types to validity of logical implications, by translating the refinements \(\tau\) and the environment \(\Gamma\) into logical formulas:

\[\lbrack \Gamma \rbrack \triangleq \tau \quad \lbrack \Gamma \rbrack \triangleq \bigwedge \{ r[x/v] \mid (x, \{v : b \mid r\}) \in \Gamma \} \]

Recall that we ensure that the refinements \(r\) belong to a decidable logic so that validity checking can be performed by an off-the-self SMT solver.

**Type Checking Rules** are standard except for **T-PGEN** and **T-PINST**, which pertain to abstraction and instantiation of abstract refinements. The rule **T-PGEN** is the same as **T-FUN**: we simply check the body \(e\) in the environment extended with a binding for the refinement variable \(\pi\). The rule **T-PINST** checks that the concrete refinement is of the appropriate (unrefined) type \(\tau\), and then replaces all (abstract) applications of \(\pi\) inside \(\sigma\) with the appropriate (concrete) refinement \(\tau'\) with the parameters \(\pi\) replaced with arguments at that application. In [21] we prove the following soundness result for \(\lambda_p\) which states that well-typed programs cannot crash:

**Lemma** (Soundness of \(\lambda_p\)). If \(\emptyset \vdash e : \sigma\) then \(e \not\rightarrow_p\text{ crash}\).

3.3 Syntax of \(\lambda_B\)

Figure 2 shows how we obtain the syntax for \(\lambda_B\) by extending the syntax of \(\lambda_p\) with **bounded** types.

**The Types** of \(\lambda_B\) extend those of \(\lambda_p\) with bounded types \(\rho\), which are the types \(t\) guarded by bounds \(\phi\).

**The Expressions** of \(\lambda_B\) extend those of \(\lambda_p\) with **abstraction** over bounds \(\Lambda\{\phi\} e\) and **application** of bounds \(e\{\phi\}\). Intuitively, if an expression \(e\) has some type \(\rho\) then \(\Lambda\{\phi\} e\) has the type \(\{\phi\} \Rightarrow \rho\). We include an explicit bound application form \(e\{\phi\}\) to simplify the formalization; these applied bounds are automatically synthesized from the type of \(e\), and are of the form \(\Lambda x : \rho.\text{true}\).

**Notation.** We write \(b, b(\pi \varphi), \{v : b(\pi \varphi) \mid r\}\) to abbreviate \(\{v : b \mid \text{true}\}, \{v : b \mid \pi \varphi\ v\}, \{v : b \mid r \land \pi \varphi\ v\}\) respectively. We say a type or schema is **non-refined** if all the refinements in it are
true. We get the shape of a type \( t \) (i.e., the System-F type) by the function \( \text{Shape}(t) \) defined:
\[
\text{Shape}\{(v : b \mid r)\} = b \\
\text{Shape}\{(x : t_1 \rightarrow t_2 \mid r)\} = \text{Shape}(t_1) \rightarrow \text{Shape}(t_2)
\]

### 3.4 Translation from \( \lambda_B \) to \( \lambda_P \)

Next, we show how to translate a term from \( \lambda_B \) to one in \( \lambda_P \). We assume, without loss of generality that the terms in \( \lambda_B \) are in Administrative Normal Form (i.e., all applications are to variables.)

**Bounds Correspond To Functions** that explicitly “witness” the fact that the bound constraint holds at a given set of “input” values. That is we can think of each bound as a universally quantified relationship between various (abstract) refinements; by “calling” the function on a set of input values \( x_1, \ldots, x_n \), we get to instantiate the constraint for that particular set of values.

**Bound Environments** \( \Phi \) are used by our translation to track the set of bound-functions (names) that are in scope at each program point. These names are distinct from the regular program variables that will be stored in Variable Environments \( \Gamma \). We give bound functions distinct names so that they cannot appear in the regular source, only in the places where calls are inserted by our translation. The translation ignores refinements entirely; both environments map their names to their non-refined types.

**The Translation is formalized** in Figure 4 via a relation \( \Gamma; \Phi \vdash e \leadsto e' \), that expresses the translation \( e \) in \( \lambda_B \) into \( e' \) in \( \lambda_P \). Most of the rules in figure 4 recursively translate the sub-expressions. Types that appear inside expressions are syntactically restricted to not contain bounds, thus types inside expressions do not require translation. Here we focus on the three interesting rules:

1. **At bound abstractions** \( \Lambda\{\phi\}.e \) we convert the bound \( \phi \) into a bound-function parameter of a suitable type,

2. **At variable binding sites** i.e., \( \lambda \) or let-bindings, we use the bound functions to materialize the bound constraints for all the variables in scope after the binding,

3. **At bound applications** \( e\{\phi\} \) we provide regular functions that witness that the bound constraints hold.

<table>
<thead>
<tr>
<th>Variable Environment</th>
<th>Bound Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma ) := \emptyset</td>
<td>( \Gamma, x : \tau )</td>
</tr>
<tr>
<td>( \Phi ) := \emptyset</td>
<td>( \Phi, x : \tau )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma; \Phi \vdash x \leadsto x )</td>
</tr>
<tr>
<td>( \Gamma; \Phi \vdash e \leadsto c )</td>
</tr>
</tbody>
</table>

\[
\Gamma \vdash \lambda x : t.e \leadsto \lambda x : t.\text{Inst}(\Gamma', \Phi, e')
\]

\[
\Gamma; \Phi \vdash e_x \leadsto e'_x \quad \Gamma = \Gamma, x : \text{Shape}(t) \quad \Gamma'; \Phi \vdash e \leadsto e'
\]

\[
\Gamma; \Phi \vdash \text{let} \ t = e_x \text{ in } e \leadsto \text{let } x : \tau \text{ in } e'_x \text{ in } \text{Inst}(\Gamma', \Phi, e')
\]

\[
\Gamma; \Phi \vdash e_1 \leadsto e'_1 \quad \Gamma; \Phi \vdash e_2 \leadsto e'_2
\]

\[
\Gamma; \Phi \vdash e \leadsto e'
\]

\[
\Gamma; \Phi \vdash e \leadsto e'
\]

**Figure 4. Translation Rules from \( \lambda_B \) to \( \lambda_P + \text{let} \)**

Performed by \( \text{Inst}(\Gamma, \Phi, e) \)

\[
\text{Inst}(\Gamma, \Phi, e) \doteq \text{Wrap}(e, \text{Instances}(\Gamma, \Phi))
\]

\[
\text{Wrap}(e, \{e_1, \ldots, e_n\}) \doteq \text{let } t_1 = e_1 \text{ in } \ldots \text{ let } t_n = e_n \text{ in } e
\]

(Where \( t_i \) are fresh Bool binders)

\[
\text{Instances}(\Gamma, \Phi) \doteq \{ \text{let } x \rightarrow x \text{ in } e \mid \Gamma, f : \tau + f : \tau \}
\]

The function takes the environments \( \Gamma \) and \( \Phi \), an expression \( e \) and a variable \( x \) of type \( \tau \) and uses let-bindings to materialize all the bound functions in \( \Phi \) that accept the variable \( x \). Here, \( \Gamma, f : \tau + f : \tau \) is the standard typing derivation judgment for the non-refined System F and so we elide it for brevity.

**3. Rule CAPP** translates bound applications \( e\{\phi\} \) into plain \( \lambda \) abstractions that witness that the bound constraints hold. That is, as within \( e \), bounds are translated to a bound function (parameter) of type \( \{\phi\} \), we translate \( \phi \) into a \( \lambda \) term that, via subtyping must have the required type \( \{\phi\} \). We construct such a function via \( \text{Const}(\phi) \) that depends only on the shape of the bound, i.e., the non-refined types of its parameters (and not the actual constraint itself).

\[
\text{Const}(r) \doteq \text{true}
\]

\[
\text{Const}(\lambda x : b.\phi) \doteq \lambda x : b.\text{Const}(\phi)
\]

This seems odd: it is simply a constant function, how can it possibly serve as a bound? The answer is that subtyping in the translated \( \lambda_P \) term will verify that in the context in which the common constant function is created, the singleton \text{true} will indeed carry the refinement corresponding to the bound constraint, making this synthesized constant function a valid realization of the bound function.
Recall that in the example \texttt{ex2} of the overview (§2.4) the subtyping constraint that decides is the constant \texttt{true} is a valid bound reduces to the equation \[ \pi \] that is a tautology.

3.5 Soundness

The Small-Step Operational Semantics of \( \lambda_B \) are defined by extending a similar semantics for \( \lambda_P \) which is a standard call-by-value calculus where abstract refinements are boolean valued functions [27]. Let \( \hookrightarrow_P \) denote the transition relation defining the operational semantics of \( \lambda_P \) and \( \hookrightarrow_B \) denote the reflexive transitive closure of \( \hookrightarrow_P \). We thus obtain the transition relation \( \hookrightarrow_B \):

\[
(\Lambda \{ \phi \}. e)(\phi) \hookrightarrow_B e \quad e \hookrightarrow_B e', \text{ if } e \hookrightarrow_P e'.
\]

Let \( \hookrightarrow^*_B \) denote the reflexive transitive closure of \( \hookrightarrow_B \).

The Translation is Semantics Preserving in the sense that if a source term \( e \) of \( \lambda_B \) reduces to a constant then the translated variant of \( e' \) also reduces to the same constant:

Lemma. If \( \emptyset; \emptyset \vdash e \hookrightarrow e' \) and \( e \hookrightarrow_B c \) then \( e' \hookrightarrow_B c \).

The Soundness of \( \lambda_B \) follows by combining the above Semantics Preservation Lemma with the soundness of \( \lambda_P \).

To Typecheck a \( \lambda_B \) program \( e \) we translate it into a \( \lambda_P \) program \( e' \) and then type check \( e' \); if the latter check is safe, then we are guaranteed that the source term \( e \) will not crash:

Theorem (Soundness). If \( \emptyset; \emptyset \vdash e \hookrightarrow e' \) and \( \emptyset \vdash e' : \sigma \) then \( e \hookrightarrow^*_B \sigma \) crash.

3.6 Inference

A critical feature of bounded refinements is that we can automatically synthesize instantiations of the abstract refinements by: (1) generating templates corresponding to the unknown types where fresh variables \( \kappa \) denote the unknown refinements that an abstract refinement parameter \( \pi \) is instantiated with, (2) generating subtyping constraints over the resulting templates, and (3) solving the constraints via abstract interpretation.

Inference Requires Monotonic Constraints. Abstract interpretation only works if the constraints are monotonic [5], which in this case means that the \( \kappa \) variables, and correspondingly, the abstract refinements \( \pi \) must only appear in positive positions within refinements \( i.e., \), not under logical negations. The syntax of refinements shown in Figure 4 violates this requirement via refinements of the form \( \pi \neg \Rightarrow \sigma \).

We restrict implications to bounds \( i.e., \) prohibit them from appearing elsewhere in type specifications. Consequently, the implications only appear in the output type of the (first order) “ghost” functions that bounds are translated to. The resulting subtyping constraints only have implications inside super-types, \( i.e., \) as:

\[
\Gamma \vdash \{ w : b \mid a \} \preceq \{ w : b \mid a_1 \Rightarrow \cdots \Rightarrow a_n \Rightarrow a_q \}
\]

By taking into account the semantics of subtyping, we can push the antecedents into the environment, \( i.e., \) transform the above into an equivalent constraint in the form:

\[
\Gamma, \{ x_1 : b_1 \mid a_1' \}, \ldots, \{ x_n : b_n \mid a_n' \} \vdash \{ w : b \mid a' \} \preceq \{ w : b \mid a_q' \}
\]

where all the abstract refinements variables \( \pi \) (and hence instance variables \( \kappa \)) appear positively, ensuring that the constraints are monotonic, hence permitting inference via Liquid Typing [21].

4. A Refined Relational Database

Next, we use bounded refinements to develop a library for relational algebra, which we use to enable generic, type safe database queries. A relational database stores data in \textit{tables}, that are a collection of \textit{rows}, which in turn are \textit{records} that represent a unit of data stored in the table. The tables’ \textit{schema} describes the types of the values in each row of the table. For example, the table in Figure 5 organizes information about movies, and has the schema:

\[
\begin{align*}
\text{Title} & : \text{String}, \text{Dir} : \text{String}, \text{Year} : \text{Int}, \text{Star} : \text{Double}
\end{align*}
\]

First, we show how to write type safe extensible records that represent rows, and use them to implement database tables (§4.4). Next, we show how bounds let us specify type safe relational operations and how they may be used to write safe database queries (§4.2).

4.1 Rows and Tables

We represent the rows of a database with dictionaries, which are maps from a set of keys to values. In the sequel, each key corresponds to a column, and the mapped value corresponds to a valuation of the column in a particular row.

A dictionary \( \text{Dict} \) maps a key \( k \) of type \( k \) to a value of type \( v \) that satisfies the property \( r \times \)

\[
type \text{Range} k v = k \rightarrow v \rightarrow \text{Bool}
\]

\[
data \text{Dict} k v \triangleq \begin{cases} k : \text{Range} k v & = \text{D} \{ \\
\quad \text{dkeys} : \text{[}k\text{]} \\
\quad \text{dfun} : x : \text{[}k \mid x \in \text{elts dkeys}\} \rightarrow v \triangleq \text{r x} \}
\end{cases}
\]

Each dictionary \( d \) has a domain \( \text{dkeys} \), \( i.e., \) the list of keys for which \( d \) is defined and a function \( \text{dfun} \) that is defined only on elements \( x \) of the domain \( \text{dkeys} \). For each such element \( x \), \( \text{dfun} \) returns a value that satisfies the property \( r \times \).

Propositions about the theory of sets can be decided efficiently by modern SMT solvers. Hence we use such propositions within refinements [28]. The measures (logical functions) \( \text{elts} \) and \( \text{keys} \) specify the set of keys in a list and a dictionary respectively:

\[
\begin{align*}
\text{elts} & : \text{[}a\text{]} \rightarrow \text{Set} a \\
\text{elts} (\text{[}]) & = \emptyset \\
\text{elts} (x : \text{xs}) & = \{x\} \cup \text{elts} \text{xs}
\end{align*}
\]

\[
\begin{align*}
\text{keys} & : \text{Dict} k v \rightarrow \text{Set} k \\
\text{keys} d & \equiv \text{elts} \text{dkeys} d
\end{align*}
\]

Domain and Range of dictionaries. In order to precisely define the domain (\( e.g., \) columns) and range (\( e.g., \) values) of a dictionary (\( e.g., \) row), we define the following aliases:

\[
\begin{align*}
type \text{RD} k v & <\text{dom} : \text{Dom} k v, \text{rng} : \text{Range} k v> \\
= \langle v : \text{Dict} <\text{rng}> k v \mid \text{dom} v \rangle
\end{align*}
\]

\[
type \text{Dom} k v = \text{Dict} k v \rightarrow \text{Bool}
\]

We may instantiate \( \text{dom} \) and \( \text{rng} \) with predicates that precisely describe the values contained with the dictionary. For example,
Thus, \( \text{key} := \text{val} \) (Haskell) type

We define singleton maps as dependent pairs \( x : y \) which denote the mapping from \( x \) to \( y \):

We populate the Movie Schema by extending the empty dictionary with the appropriate pairs of fields and values. For example, here are the rows from the table in Figure 5:

We refine the Movie Schema with concrete instantiations for \( \text{dom} \) and \( \text{rng} \), in order to recover precise specifications for a particular database. For example, MovieSchema is a refined Schema that describes the rows of the Movie table in Figure 5:

4.2 Relational Algebra

Next, we describe the types of the relational algebra operators which can be used to manipulate refined rows and tables. For space reasons, we show the types of the basic relational operators; their (verified) implementations can be found online [24].
union and diff compute the union and difference, respectively of the (rows of) two tables. The types of union and diff state that the operators work on tables with the same schema s and return a table with the same schema.

select takes a predicate p and a table t and filters the rows of t to those which satisfy that predicate. The type of select ensures that p will not reference columns (fields) that are not mapped in t, as the predicate p is constrained to require a dictionary with schema s.

project takes a list of String fields ks and a table t and projects exactly the fields ks at each row of t. project’s type states that for any schema s, the input table has type RefinedTableWith Theks s s i.e., its domain should have at least the fields ks and the result table has type $RTEqFlds$ ks s i.e., its domain has exactly the elements ks.

type $RTEqFlds$ ks s
$\rightarrow RT s$ = $\lambda$ $z \rightarrow elt$s $ks \subseteq keys$ $z$.

The range of the argument and the result tables is the same and equal to $s.rng$.

product takes two tables as input and returns their (Cartesian) product. It takes two Refined Tables with schemata takes two tables as input and returns their (Cartesian) product. It takes two Refined Tables with schemata.

 Then the range of the argument and the result tables is the same and equal to $s.rng$.

product takes two tables as input and returns their (Cartesian) product. It takes two Refined Tables with schemata $s1$ and $s2$ and returns a Refined Table with schema $s$. Intuitively, the output schema is the “concatenation” of the input schema; we formalize this notion using bounds: (1) Disjoint $s1$ $s2$ says the domains of $s1$ and $s2$ should be disjoint, (2) Union $s1$ $s2$ $s$ says the domain of $s$ is the union of the domains of $s1$ and $s2$, (3) Range $s1$ $s$ (resp. Range $s2$ $s$) says the range of $s1$ should imply the result range $s$; together the two imply the output schema $s$ preserves the type of each key in $s1$ or $s2$.

bound Disjoint $s1$ $s2$ $= \lambda x y \rightarrow s1.d0m x \Rightarrow s2.d0m y \Rightarrow keys x \cap keys y \Rightarrow \emptyset$.

bound Union $s1$ $s2$ $s$ $= \lambda x y v \rightarrow s1.d0m x \Rightarrow s2.d0m y \Rightarrow keys v \Rightarrow keys x \cup keys y \Rightarrow s.d0m v$.

bound Range $s1$ $s$ $= \lambda x k v \rightarrow s1.d0m x \Rightarrow k \in keys x \Rightarrow s.rng k v \Rightarrow s.rng k v$.

Thus, bounded refinements enable the precise typing of relational algebra operations. They let us describe precisely when union, intersection, selection, projection and products can be computed, and let us determine, at compile time the exact “shape” of the resulting tables.

We can query Databases by writing functions that use the relational algebra combinators. For example, here is a query that returns the “good” titles – with more than 8 stars – from the movies table:\footnote{More example queries can be found online \cite{24}.

$$good\_titles = project \ ["title"] \ $\ select \ (\lambda d \rightarrow\ toDouble \ (dfun \ d \ "star") \ > \ 8 ) \ movies$$

Finally, note that our entire library – including records, tables, and relational combinators – is built using vanilla Haskell i.e., without any type level computation. All schema reasoning happens at the granularity of the logical refinements. That is if the refinements are erased from the source, we still have a well-typed Haskell program but of course, lose the safety guarantees about operations (e.g., “dynamic” key lookup) never failing at run-time.

5. A Refined IO Monad

Next, we illustrate the expressiveness of Bounded Refinements by showing how they enable the specification and verification of stateful computations. We show how to (1) implement a refined state transformer (RIO) monad, where the transformer is indexed by refinements corresponding to pre- and post-conditions on the state ($\S 5.1$, 2) extend RIO with a set of combinators for imperative programming, i.e., whose types precisely encode Floyd-Hoare style program logics ($\S 5.2$) and (3) use the RIO monad to write safe scripts where the type system precisely tracks capabilities and statically ensures that functions only access specific resources ($\S 6$).

5.1 The RIO Monad

The RIO data type describes stateful computations. Intuitively, a value of type RIO a denotes a computation that, when evaluated in an input World produces a value of type a (or diverges) and a potentially transformed output World. We implement RIO a as an abstractly refined type (as described in \cite{27}).

$$type Pre = World \rightarrow Bool$$

$$type Post a = World \rightarrow a \rightarrow World \rightarrow Boolean$$

$$data RIO a <p :: Pre , q :: Post a> = RIO{runState :: w: World <p> \rightarrow (x:a, World<q w x>) }$$

That is, RIO a is a function World$\rightarrow$(a , World), where World is a primitive type that represents the state of the machine i.e., the console, file system, etc. This indexing notion is directly inspired by the method of \cite{8} (also used in \cite{16}).

Our Post-conditions are Two-State Predicates that relate the input- and output- world (as in \cite{16}). Classical Floyd-Hoare logic, in contrast, uses assertions which are single-state predicates. We use two-states to smoothly account for specifications for stateful procedures. This increased expressiveness makes the types slightly more complex than a direct one-state encoding which is, of course also possible with bounded refinements.

An RIO computation is parameterized by two abstract refinements: (1) $p ::$ Pre, which is a predicate over the input world, i.e., the input world $w$ satisfies the refinement $p w$; and (2) $q ::$ Post a, which is a predicate relating the output world with the input world and the value returned by the computation, i.e., the output world $w'$ satisfies the refinement $q w x w'$ where $x$ is the value returned by the computation. Next, to use RIO as a monad, we define bind and return functions for it, that satisfy the monad laws.

The return operator yields a pair of the supplied value $z$ and the input world unchanged:

$$return :: z:a \rightarrow RIO <p, ret z> a$$

$$return z = RIO \ s \ lambda w \rightarrow \{z, w\}$$

$$ret z = \lambda w x w' \rightarrow \lambda w \&\& x == z$$
The type of return states that for any precondition \( p \) and any supplied value \( z \) of type \( a \), the expression \( \text{return } z \) is an RIO computation with precondition \( p \) and a post-condition \( \text{ret } z \). The postcondition states that: (1) the output \( \text{World} \) is the same as the input, and (2) the result equals to the supplied value \( z \). Note that as a consequence of the equality of the two worlds and congruence, the output world \( w' \) trivially satisfies \( p \ w' \).

**The >>\( = \)** Operator is defined in the usual way. However, to type it precisely, we require bounded refinements.

\[
(\text{>>=}) : \ (\text{Ret } q1 \ r, \text{ Seq } q1 \ p2, \text{ Trans } q1 \ q2 \ q) \\
\Rightarrow m: \text{RIO } \langle p, q1 \rangle \ a \\
\to k: (x:a \langle r \rangle \to \text{RIO } \langle p2 \ x, q2 \ x \rangle \ b) \\
\Rightarrow \text{RIO } \langle p, q \rangle \ b
\]

\[
(\text{RIO } g) \ \text{>>=} f = \text{RIO } \lambda x \to \begin{cases} \\
\text{case } g \times \text{of } \{ (y,s) \to \text{runState } (f \ y) \ s \} \\
\end{cases}
\]

The bounds capture various sequencing requirements (c.f. the Floyd-Hoare rules of consequence). First, the output of the first action \( m \) satisfies the refinement required by the continuation \( k \);

\[
\text{bound } \text{Ret } q1 \ r = \lambda w x w' \to q1 \ w \times w' \Rightarrow r \ x
\]

Second, the computations may be sequenced, *i.e.*, the postcondition of the first action \( m \) implies the precondition of the continuation \( k \) (which may be dependent upon the supplied value \( x \)):

\[
\text{bound } \text{Seq } q1 \ p2 = \lambda w x w' \to \begin{cases} \\
q1 \ w \times w' \Rightarrow p2 \times w' \\
\end{cases}
\]

Third, the transitive composition of the two computations, implies the final postcondition:

\[
\text{bound } \text{Trans } q1 \ q2 \ q = \lambda w x w' y w'' \to \begin{cases} \\
q1 \ w \times w' \Rightarrow q2 \times y w'' \Rightarrow q w y w'' \\
\end{cases}
\]

Both type signatures would be impossible to use if the programmer had to manually instantiate the abstract refinements (c.f. pre- and post-conditions.) Fortunately, Liquid Type inference generates the instantiations making it practical to use LIQUID-HASKELL to verify stateful computations written using do-notation.

### 5.2 Floyd-Hoare Logic in the RIO Monad

Next, we use bounded refinements to derive an encoding of Floyd-Hoare logic, by showing how to read and write (mutable) variables and typing higher order ifM and whileM combinators.

**We Encode Mutable Variables** as fields of the World type. For example, we might encode a global counter as a field:

```haskell
data World = { ... , ctr :: Int, ... }
```

We encode mutable variables in the refinement logic using McCarthy's select and update operators for finite maps and the associated axioms:

```haskell
select :: Map k v -> k -> v
update :: Map k v -> k -> v -> Map k v
\forall m, k1, k2, v. \\
select (update m k1 v) k2 \\
== (if k1 == k2 then v else select m k2 v)
```

The quantifier free theory of select and update is decidable and implemented in modern SMT solvers [1].

**We Read and Write Mutable Variables** via suitable "get" and "set" actions. For example, we can read and write \( \text{ctr} \) via:

```haskell
getCtr :: RIO <true , rdCtr> Int
getCtr = RIO $ \lambda w \to (\text{ctr } w, w)
setCtr :: Int -> RIO <true , wrCtr> n \\
setCtr n = RIO $ \lambda w \to (\{\}, (w \ (\text{ctr } = n))
```

Here, the refinements are defined as:

\[
p\text{True } = \lambda w \rightarrow \text{True} \ 
\text{rdCtr } = \lambda w x w' \to w' = w \&\& \ x = \text{select } w \text{ ctr} \\
\text{wrCtr } n = \lambda w w' \rightarrow w = \text{update } w \text{ ctr } n
\]

Hence, the post-condition of getCtr states that it returns the current value of ctr, encoded in the refinement logic with McCarthy's select operator while leaving the world unchanged. The post-condition of setCtr states that World is updated at the address corresponding to ctr, encoded via McCarthy's update operator.

**The ifM combinator** takes as input a cond action that returns a Bool and, depending upon the result, executes either the then or else actions. We type it as:

```haskell
bound Pure g = \lambda w \times v \to (g w \times v \Rightarrow v = w) \\
bound Then g p1 = \lambda w \times v \to (g w \text{ True} \Rightarrow p1 v) \\
bound Else g p2 = \lambda w \times v \to (g w \text{ False} \Rightarrow p2 v)
```

```
\text{ifM} :: (\text{Pure } g, \text{ Then } g p1, \text{ Else } g p2) \\
\Rightarrow RIO \langle p, g \rangle \text{ Bool} -- \text{cond} \\
\Rightarrow RIO \langle p1, q1 \rangle \ a -- \text{then} \\
\Rightarrow RIO \langle p2, q2 \rangle \ a -- \text{else} \\
\Rightarrow RIO \langle p, q, a \rangle
```

The abstract refinements and bounds correspond exactly to the hypotheses in the Floyd-Hoare rule for the if statement. The bound Pure \( g \) states that the cond action may access but does not modify the World, i.e., the output is the same as the input World. (In classical Floyd-Hoare formulations this is done by syntactically separating terms into pure expressions and side effecting statements). The bound Then \( g \ p1 \) and Else \( g \ p2 \) respectively state that the preconditions of the then and else actions are established when the cond returns True and False respectively.

**We can use ifM to implement a stateful computation that performs a division, after checking the divisor is non-zero.** We specify that div should not be called with a zero divisor. Then, LIQUID-HASKELL verifies that div is called safely:

```
div :: Int \to (v: Int \mid v /= 0) \to Int
ifTest :: RIO Int
ifTest = ifM nonZero divX (return 10) \\
where nonZero = getCtr >>\( = \) return . (/= 0) \\
divX = getCtr >>\( = \) return . (div 42)
```

Verification succeeds as the post-condition of nonZero is instantiated to \( \lambda b \ b \rightarrow b \leftrightarrow \text{select } w \text{ ctr} /= 0 \) and the pre-condition of divX's is instantiated to \( \lambda w \rightarrow \text{select } w \text{ ctr} /= 0 \), which suffices to prove that div is only called with non-zero values.

**The whileM combinator** formalizes loops as RIO computations:

```
whileM :: (OneState q, \text{Inv } p g b, \text{Exit } p g q) \\
\Rightarrow RIO \langle p, g \rangle \text{ Bool} -- \text{cond} \\
\Rightarrow RIO \langle ptrue, b \rangle () -- \text{body} \\
\Rightarrow RIO \langle p, q, () \rangle
```

As with ifM, the hypotheses of the Floyd-Hoare derivation rule become bounds for the signature. Given a condition with precondition \( p \) and post-condition \( g \) and body with a true precondition and post-condition \( b \), the computation whileM cond body has
precondition p and post-condition q as long as the bounds (corresponding to the Hypotheses in the Floyd-Hoare derivation rule) hold. First, p should be a loop invariant; i.e., when the condition returns True the post-condition of the body b must imply the p:

\[ \text{bound Inv: } p \land g \land b \Rightarrow \lambda w. w' \Rightarrow p w \Rightarrow g w \Rightarrow \lambda w. w' \Rightarrow \text{True} \Rightarrow b w' \Rightarrow \text{True} \Rightarrow w'' \Rightarrow p w'' \]

Second, when the condition returns False the invariant p should imply the loop’s post-condition q:

\[ \text{bound Exit: } p \land g \land q \Rightarrow \lambda w. w' \Rightarrow p w \Rightarrow g w \Rightarrow \lambda w. w' \Rightarrow q w \Rightarrow \text{False} \Rightarrow w'' \Rightarrow \text{False} \Rightarrow w'' \rightarrow w' \]

Third, to avoid having to transitively connect the guard and the body, we require that the loop post-condition be a one-state predicate, independent of the input world (as in Floyd-Hoare logic):

\[ \text{bound OneState: } q \Rightarrow \lambda w. w' \Rightarrow q w \Rightarrow \text{False} \Rightarrow w'' \Rightarrow q w' \Rightarrow \text{False} \Rightarrow w'' \]

We can use \textit{whileM} to implement a loop that repeatedly decrements a counter while it is positive, and to then verify that if it was initially non-negative, then at the end the counter is equal to 0.

\[
\begin{align*}
\text{whileTest} & : \text{RIO <posCtr, zeroCtr> ()} \\
\text{whileTest} & = \text{whileM gtZeroX decr} \\
& \text{where gtZeroX = getCtr >>= return . (> 0)} \\
\text{posCtr} & = \lambda w. 0 \leq \text{select w ctr} \\
\text{zeroCtr} & = \lambda w. w = 0 \Rightarrow w = \text{select w ctr}
\end{align*}
\]

Where the decrement is implemented by \textit{decr} with type:

\[
\begin{align*}
\text{decr} & : \text{RIO <True, decctr> ()} \\
\text{decr} & = \text{RIO <True, decctr> ()} \\
\text{decctr} & = \lambda w. w' \rightarrow \\
\text{w''} & = \text{update w ctr} ((\text{select ctr w} - 1))
\end{align*}
\]

\text{LIQUIDHaskell} verifies that at the end of \textit{whileTest} the counter is zero (i.e., the post-condition \textit{zeroCtr}) by instantiating suitable (i.e., inductive) refinements for this particular use of \textit{whileM}.

### 6. Capability Safe Scripting via RIO

Next, we describe how we use the RIO monad to reason about shell scripting, inspired by the Shill\textsuperscript{[15]} programming language.

Shill is a scripting language that restricts the privileges with which a script may execute by using capabilities and dynamic contract checking\textsuperscript{[15]}. Capabilities are run-time values that witness the right to use a particular resource (e.g., a file). A capability is associated with a set of privileges, each denoting the permission to use the capability in a particular way (such as the permission to write to a file). A contract for a Shill procedure describes the required input capabilities and any output values. The Shill run-time guarantees that system resources are accessed in the manner described by its contract.

In this section, we turn to the problem of preventing Shill runtime failures. (In general, the verification of file system resource usage is a rich topic outside the scope of this paper.) That is, assuming the Shill runtime and an API as described in \textsuperscript{[15]}, how can we use Bounded Refinement Types to encode scripting privileges and reason about them statically?

We use \textit{RIO types} to specify Shill’s API operations thereby providing compile-time guarantees about privilege and resource usage. To achieve this, we: connect the state (\textit{World}) of the RIO monad with a \textit{privilege specification} denoting the set of privileges that a program may use (§ 6.1); specify the \textit{file system API} in terms of this abstraction (§ 6.2), and use the above to specify and verify the particular privileges that a \textit{client} of the API uses (§ 6.3).

#### 6.1 Privilege Specification

Figure 6 summarizes how we specify privileges inside \textit{RIO}. We use the type \textit{FH} to denote a file handles, analogous to Shill’s capabilities. An abstract type \textit{Priv} denotes the sets of privileges that may be associated with a particular FH.

To connect Worlds with Privileges, we assume a set of uninterpreted functions of type \textit{Priv \rightarrow Bool} that act as predicates on values of type \textit{Priv}, each denoting a particular privilege. For example, given a value \textit{p :: Priv}, the proposition \textit{pread p} denotes that \textit{p} includes the “read” privilege. The function \textit{caps} associates each \textit{World} with a \textit{Map FH Priv}, a table that associates each FH with its privileges. The function \textit{active} maps each \textit{World} to the \textit{Set} of allocated \textit{FHs}. Given \textit{x:FH and w:World, \textit{pwrite (select (caps w)x)}} denotes that in the state \textit{w}, the file \textit{x} may be written. This pattern is generalized by the predicate \textit{pset pwrite x w}.

#### 6.2 File System API Specification

A privilege tracking file system API can be partitioned into the privilege preserving operations and the privilege extending operations.

To type the privilege preserving operations, we define a predicate \textit{eqP} \textit{w w'} that says that the set of privileges and active handles in worlds \textit{w} and \textit{w'} are equivalent.

\[
\begin{align*}
\textit{eqP} & = \lambda w. w' \Rightarrow \text{caps w == caps w'} \land \text{active w == active w'}
\end{align*}
\]

We can now specify the privilege preserving operations that \textit{read} and \textit{write} files, and list the contents of a directory, all of which require the capabilities to do so in their pre-conditions:

\[
\begin{align*}
\textit{read} & : (- \text{ Read the contents of } h -) \\
& \text{h:FH \rightarrow RIO<pset pread h, eqP > String} \\
\textit{write} & : (- \text{ Write to the file } h -) \\
& \text{h:FH \rightarrow String \rightarrow RIO<pset pwite h, eqP > ()} \\
\textit{contents} & : (- \text{ List the children of } h -) \\
& \text{h:FH \rightarrow RIO<pset pcontents h, eqP > [Path]}
\end{align*}
\]

To type the privilege extending operations, we define predicates that say that the output world is suitably extended. First, each such operation \textit{allocates} a new handle, which is formalized as:

\[
\begin{align*}
\text{alloc w' w x} & = \text{x \notin active w} \land \text{active w' == (x \cup active w)}
\end{align*}
\]

which says that the active handles in (the new \textit{World}) \textit{w'} are those of (the old \textit{World}) \textit{w} extended with the hitherto inactive handle \textit{x}. Typically, after allocating a new handle, a script will want to add privileges to the handle that are obtained from existing privileges.
To create a new file in a directory with handle h we want the new file to have the privileges derived from pcreateFP (select (caps w) h) (i.e., the create privileges of h). We formalize this by defining the post-condition of create as the predicate derivP:

\[
\text{derivP} h = \lambda x \ x' \rightarrow
\text{alloc} w' w x \& \&
\text{caps w'} == \text{store} (\text{caps} w) x
\]

(\text{pcreateFP (select (caps w))) h}

create :: (- Create a file -)

\[
h : \text{FH} \rightarrow \text{Path} \rightarrow \text{RIO} \left< \text{pset} \ pcreateF h, \text{derivP} h \right> \text{FH}
\]

Thus, if h is writable in the old World w (pwrite (pcreateFP (select (caps w) h))) and x is derived from h (derivP w' w x h both hold), then we know that \( x \) is writable in the new World w' (\text{pwrite (select (caps w') x)}).

To lookup existing files or create sub-directories, we want to directly copy the privileges of the parent handle. We do this by using a predicate copyP as the post-condition for the two functions:

\[
\text{copyP} h = \lambda x \ x' \rightarrow
\text{alloc} w' w x \& \&
\text{caps w'} == \text{store} (\text{caps} w) x
\]

\[
\text{lookup} :: (- \text{Open a child of h -})
\]

\[
h : \text{FH} \rightarrow \text{Path} \rightarrow \text{RIO} \left< \text{pset} \ plookup h, \text{copyP} h \right> \text{FH}
\]

createDir :: (- Create a directory -)

\[
h : \text{FH} \rightarrow \text{Path} \rightarrow \text{RIO} \left< \text{pset} \ pcreateD h, \text{copyP} h \right> \text{FH}
\]

6.3 Client Script Verification

We now turn to a client script, the program copyRec that copies the contents of the directory f to the directory d.

\[
\text{copyRec} \text{ recur s d} =
\text{do} \ cs <= \text{contents s}
\text{forM_} cs \ s \ \lambda p \rightarrow \text{do}
\text{x <- flookup s p}
\text{when (isFile x) } \ p \ \rightarrow \text{do}
\text{y <- create d p}
\text{w <- pread x}
\text{write y w}
\text{when ((recur && (isDir x)) } \ p \ \rightarrow \text{do}
\text{recur y x}
\text{copyRec recur y x}
\]

copyRec executes by first listing the contents of f, and then opening each child path p in f. If the result is a file, it is copied to the directory d. Otherwise, copyRec recurses on p, if recur is true.

In a first attempt to type copyRec we give it the following type:

\[
\text{copyRec} :: \text{Bool} \rightarrow \text{s:FH} \rightarrow \text{d:FH} \rightarrow \text{RIO} \left< \text{copySpec s d w} \right> ()
\]

\[
\text{copySpec} \ h \ d = \lambda w \rightarrow
\text{pset} \ \text{pcontents} h w \&\& \text{pset} \ \text{plookup} h w \&\&
\text{pset} \ \text{pread} h w \&\& \text{pset} \ \text{pcreateFile} d w \&\&
\text{pset} \ \text{pwrite} d w \&\& \text{pset} \ \text{pcreateD} d w \&\&
\text{pwrite} \ \text{pcreateFP (select (caps w) d)}
\]

The above specification gives copyRec a minimal set of privileges. Given a source directory handle s and destination handle d, the copyRec must at least: (1) list the contents of s (pcontents), (2) open children of s (plookup), (3) read from children of s (pread), (4) create directories in d (pcreateD), (5) create files in d (pcreateFP), an (6) write to (created) files in d (pwrite).

Furthermore, we want to restrict the privileges on newly created files to the write privilege, since copyRec does not need to read from or otherwise modify these files.

Even though the above type is sufficient to verify the various clients of copySpec it is insufficient to verify copySpec’s implementation, as the postcondition merely states that copySpec s d w holds. Looking at the recursive call in the last line of copySpec’s implementation, the output world w is only known to satisfy copySpec x y w (having substituted the formal parameters s and d with the actual x and y), with no mention of s or d! Thus, it is impossible to satisfy the postcondition of copyRec, as information about s and d has been lost.

Framing is introduced to address the above problem. Intuitively, because no privileges are ever revoked, if a privilege for a file existed before the recursive call, then it exists after as well. We thus introduce a notion of framing – assertions about unmodified state that hold before calling copyRec must hold after copyRec returns. Solidifying this intuition, we define a predicate i to be Stable when assuming that the predicate i holds on w, if i only depends on the allocated set of privileges, then i will hold on a world w’, so long as the set of privileges in w’ contains those in w. The definition of Stable is derived precisely from the ways in which the file system API may modify the current set of privileges:

\[
\text{bound Stable i = } \lambda x \ y \ w \ w' \rightarrow
i \ w' \Rightarrow \text{eqP w} (\text{caps} w) x \&\& \text{eqP y w x w'}
\]

We thus parameterize copyRec by a predicate i, bounded by Stable i, which precisely describes the possible world transformations under which i should be stable:

\[
\text{copyFrame} i \ s \ d = \lambda w \rightarrow \text{i w} \&\& \text{copySpec s d w}
\]

\[
\text{copyRec} :: (\text{Stable i}) \Rightarrow
\text{Bool} \rightarrow \text{s:FH} \rightarrow \text{d:FH} \rightarrow
\text{RIO} \left< \text{copyFrame i s d}, \lambda _- w \rightarrow \text{copyFrame i s d w} \right> ()
\]

Now, we can verify copyRec’s body, as at the recursive call that appears in the last line of the implementation, i is instantiated with \( \lambda w \rightarrow \text{copySpec s d w} \).

7. Related Work

Higher order Logics and Dependent Type Systems including NuPRL [4], Coq [3], Agda [13], and even to some extent, Haskell [14, 20], occupy the maximal extreme of the expressiveness spectrum. However, in these settings, checking requires explicit proof terms which can add considerable programmer overhead. Our goal is to eliminate the programmer overhead of proof construction by restricting specifications to decidable, first order logics and to see how far we can go without giving up on expressiveness. The F* system enables full dependent typing via SMT solvers via a higher-order universally quantified logic that permit specifications similar to ours (e.g., compose, filter and foldr). While this approach is at least as expressive as bounded refinements it has two drawbacks. First, due to the quantifiers, the generated VCs fall outside the SMT decidable theories. This renders the type system undesirable (in theory), forcing a dependency on the solver’s unpredictable quantifier instantiation heuristics (in practice). Second, more importantly, the higher order predicates must be explicitly instantiated, placing a heavy annotation burden on the programmer. In contrast, bounds permit decidable checking, and are automatically instantiated via Liquid Types.
Our notion of Refinement Types has its roots in the predicate subtyping of PVS [22] and indexed types (DML [30]) where types are constrained by predicates drawn from a logic. To ensure decidable checking several refinement type systems including [6, 29, 30] restrict refinements to decidable, quantifier free logics. While this ensures predictable checking and inference [21] it severely limits the language of specifications, and makes it hard to fashion simple higher order abstractions like filter (let alone the more complex ones like relational algebras and state transformers.)

To Reconcile Expressiveness and Decidability CATALYST [11] permits a form of higher order specifications where refinements are relations which may themselves be parameterized by other relations, which allows for example, a way to precisely type filter by suitably composing relations. However, to ensure decidable checking, CATALYST is limited to relations that can be specified as catamorphisms over inductive types, precluding for example, theories like arithmetic. More importantly, (like F*), CATALYST provides no inference: higher order relations must be explicitly instantiated. Bounded refinements build directly upon abstract refinements [27], a form of refinement polymorphism analogous to parametric polymorphism. While [27] adds expressiveness via abstract refinements, without bounds we cannot specify any relationships between the abstract refinements. The addition of bounds makes it possible to specify and verify the examples shown in this paper, while preserving decidability and inference.

Our Relational Algebra Library builds on a long line of work on type safe database access. The HaskellDB [13] showed how phantom types could be used to eliminate certain classes of errors. Haskell’s HList library [12] extends this work with type-level computation features to encode heterogeneous lists, which can be used to encode database schema, and (unlike HaskellDB) statically reject accesses of “missing” fields. The HList implementation is non-trivial, requiring new type-classses for new operations (e.g., appending lists); [19] shows how a dependently typed language greatly simplifies the implementation. Much of this simplicity can be recovered in Haskell using the singleton library [17]. Our goal is to show that bounded refinements are expressive enough to perm the construction of rich abstractions like a relational algebra and generic combinators for safe database access while using SMT solvers to provide decidable checking and inference. Further, unlike the HList based approaches, refinements they can be used to retroactively or gradually verify safety; if we erase the types we still get a valid Haskell program operating over homogeneous lists.

Our Approach for Verifying Stateful Computations using monads indexed by pre- and post-conditions is inspired by the method of Filliâtre [8], which was later enriched with separation logic in Ynot [16]. In future work it would be interesting to use separation logic based refinements to specify and verify the complex sharing and aliasing patterns allowed by Ynot. F* encodes stateful computations in a special Dijkstra Monad [23] that replaces the two assertions with a single (weakest-precondition) predicate transformer which can be composed across sub-computations to yield a transformer for the entire computation. Our RIO approach uses the idea of indexed monads but has two concrete advantages. First, we show how bounded refinements alone suffice to let us fashion the RIO abstraction from scratch. Consequently, second, we automate inference of pre- and post-conditions and loop invariants as refinement instantiation via Liquid Typing.

Acknowledgments
We thank the anonymous reviewers and Colin Gordon for providing invaluable feedback about earlier drafts of this paper.

References