Bounded Refinement Types

Niki Vazou   Alexander Bakst   Ranjit Jhala
UC San Diego

Abstract

We present a notion of bounded quantification for refinement types. We show how bounded quantification expands the *expressiveness* of refinement typing by (1) developing typed combinators for relational algebra and safe database access, (2) encoding Floyd-Hoare logic in 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. Fortunately, we show that by translating bounds into “ghost” functions, the increased expressiveness comes while preserving the automated and decidable SMT based checking and inference that makes refinement typing effective in practice.

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 restricted logics, such as Refinement Types [21, 29], appear to trade expressiveness for automation. In the case of refinement types, SMT solvers can fully automate verification by eliminating the need to provide proof terms. This high degree of automation has enabled the use of refinement types for a variety of verification tasks, ranging from array bounds checking [20], termination and totality checking [28], protocol validation [2, 8], and securing web applications [9]. Unfortunately, this automation comes at a price. To ensure predictable and decidable type checking, one must limit the logical formulas appearing in specifications types to decidable, (typically quantifier free) first order theories, making it hard to build higher-order abstractions and enable modular reasoning.

We introduce *Bounded Refinement Types* which enable *bounded quantification* over refinements. Previously we developed a mechanism for quantifying type signatures over abstract refinement parameters [26]. We preserved decidability of checking and inference by encoding abstractly refined types with uninterpreted functions obeying the decidable axioms of congruence [16]. While useful, refinement quantification was not enough to enable higher order abstractions requiring fine grained *dependencies between* abstract refinements. In this paper, we solve this problem by enriching signatures with bounded quantification. The *bounds* correspond to Horn implications between abstract refinements, which, as in the classical setting, correspond to subtyping constraints that must be satisfied by the concrete refinements used at any call-site. This simple addition 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 [26]. Consequently, checking remains decidable and, as the bounds are Horn constraints, we can directly reuse the abstract interpretation of Liquid Typing [20] 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 (S 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 [7]. 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-granularity, ensuring that functions only access specified resources (§ 6).

We have implemented Bounded Refinement Types in LIQUID-HASKELL [28]. The source code of the examples, with more verbose concrete syntax is at [23]. While the construction of these verified abstractions is possible with full dependent types our encoding makes it possible, to make checking automatic and decidable, and to use abstract interpretation to automatically synthesize refinements (*i.e.,* pre- and post-conditions and loop invariants). 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 conjoining 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. For example, we can write

```
type Pos      = {v:Int | 0 < v}
type IntGE x = {v:Int | x ≤ v}
```

to specify subsets of `Int` corresponding to values that are positive, negative or larger than some other value `x` respectively. 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 [28] the check reduces to establishing the validity of the *verification condition* (VC)

$$\mathtt{a} \leq \mathtt{b} \Rightarrow (\mathtt{cmp} \Leftrightarrow \mathtt{a} \leq \mathtt{b}) \Rightarrow \mathtt{v{=}cmp} \Rightarrow (\mathtt{v} \Leftrightarrow \mathtt{true})$$

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 [16] 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
```

It is difficult to write a first-order refinement type specification for `maximum` that will let us verify that:

```
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 [26]. 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 [16]. 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`.

Notation 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, [26] demonstrates how 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
```

thereby preserving decidable checking [16].

Abstract Refinements are Automatically Instantiated at call-sites, via the abstract interpretation framework of Liquid Typing [26]. 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 [25]. `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. `ex1` passes `find` a continuation that checks that the "found" value equals or exceeds `n`.

```
ex1 :: (Int → Bool) → Int → ()
ex1 q n = find q (checkGE n) n

find q k i
  | q i       = k i
  | otherwise = find q k (i + 1)
```

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? Lets try to abstract over the refinement that `i` enjoys, and assign `find` the type:

```
(Int → Bool) → (Int<p> → a) → Int<p> → 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 `ex1` we can instantiate

$$\mathtt{p} \mapsto \lambda \mathtt{v} \rightarrow \mathtt{n} \leq \mathtt{v} \tag{1}$$

This instantiated refinement is satisfied by the parameter `n`, and sufficient to verify, via function subtyping, that `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* satisfy 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 `find`! Concretely, `{v:Int|v=i+1}` is in general *not* a subtype of `Int<p>`, as the associated VC

$$\ldots \Rightarrow \mathtt{p\ i} \Rightarrow \mathtt{p\ (i{+}1)} \tag{2}$$

is *invalid* – the type checker thus (soundly!) reject `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, `UpClosed` which states that `p` is a refinement that is *upward closed*, *i.e.,* satisfies `∀ x. p x ⇒ p (x+1)`, and use it to type `find` as:

```
bound UpClosed (p :: Int → Bool)
  = λx → p x ⇒ p (x+1)

find :: (UpClosed p) ⇒ (Int → Bool)
                     → (Int<p> → a)
                     →  Int<p> → a
```

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 VCs) remain quantifier free and hence, SMT decidable (§ 2.4).

***At the call to* `find`** in the body of `ex1`, we perform the instantiation (1) which generates the *additional* VC (n ≤x ⇒ n ≤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 verifying `ex1`, *i.e.,* statically verifying that the assertion inside `checkGE` cannot fail.

2.3 Bounds for Higher-Order Functions

Next, we show how bounds stretch the scope of refinement typing by precisely specificying several canonical higher-order functions.

Function Composition

Our first example is the `compose` function. What is a modular specification for `compose` that lets us check `ex2`?

```
compose f g x = f (g x)

type Plus x y = {v:Int | v = x + y}
ex2    :: n:Int → Plus n 2
ex2    = incr ‘compose‘ incr

incr   :: n:Int → Plus n 1
incr n = n + 1
```

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:

```
bound Chain p q r = λx y z →
       q x y ⇒ p y z ⇒ r x z
```

which states that for any `x`, `y` and `z`, if (1) `x` and `y` are related by `q`, and (2) `y` and `z` are related by `p`, then (3) `x` and `z` are related by `r`.

We use `Chain` to type `compose` using three abstract refinements `p`, `q` and `r`, relating the arguments and return values of `f` and `g` to their composed value. (Here, `c<r x>` abbreviates `{v:c | r x v}`.)

```
compose :: (Chain p q r) ⇒ (y:b → c<p y>)
                         → (z:a → b<q z>)
                         → (x:a → c<r x>)
```

***To verify* `ex2`** we instantiate, at the call to `compose`,

```
p, q ↦ λx v → v = x + 1
   r ↦ λx v → v = x + 2
```

The above instantiation satisfies the bound, as shown by the validity of the VC derived from instantiating `p`, `q`, and `r` in `Chain`:

```
y = x + 1 ⇒ z = y + 1 ⇒ z = x + 2
```

and hence, we can check that `ex2` implements its specified type.

List Filtering

Next, consider the List `filter` function. What type signature for `filter` would let us check `positives`?

```
filter q (x:xs)
  | q x          = x : filter q xs
  | otherwise    = filter q xs
filter _ []      = []

positives        :: [Int] → [Pos]
positives        = filter isPos
  where isPos x = 0 < x
```

Such a signature would have to relate the `Bool` returned by `f` with the property of the `x` that it checks for.

Typed Racket’s latent predicates [24] account for this idiom, but are a special construct limited to `Bool`-valued “type” tests, and not arbitrary invariants. Another approach is to avoid the so-called “Boolean Blindness” that accompanies `filter` by instead using option types and `mapMaybe`.

We overcome blindness using a witness bound:

```
bound Witness p w = λx b → b ⇒ w x b ⇒ p x
```

which says that `w` *witnesses* the refinement `p`. That is, for any boolean `b` such that `w x b` holds, if `b` is **`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`:

```
filter :: (Witness p w) ⇒ (x:a → Bool<w x>)
                        → List a
                        → List a<p>
```

***To verify* `positives`** we infer the following type and instantiations for the abstract refinements `p` and `w` at the call to `filter`:

```
isPos :: x:Int → {v:Bool | v ⇔ 0 < v}
p     ↦ λv     → 0 < v
w     ↦ λx b   → b ⇔ 0 < x
```

List Folding

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

```
foldr :: (a → b → b) → b → List a → b
foldr op b []     = b
foldr op b (x:xs) = x ‘op‘ foldr op b xs

ex3 :: xs:List a → {v:Int | v = len xs}
ex3 = foldr (λ_ → incr) 0
```

where `len` is a *logical* or *measure* function used to represent the number of elements of the list in the refinement logic [28]:

```
measure len :: List a → Nat
len []      = 0
len (x:xs)  = 1 + len xs
```

We specify induction as a bound. Let (1) `inv` be an abstract refinement relating a list `xs` and the result `b` obtained by folding over it, and (2) `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 `inv` is closed under `step` thus:

```
bound Inductive inv step = λx xs b b’ →
  inv xs b ⇒ step x b b’ ⇒ inv (x:xs) b’
```

We can give* `foldr` *a type that says that the function *outputs* a value that is built inductively over the entire *input* list:

```
foldr :: (Inductive inv step)
      ⇒ (x:a → acc:b → b<step x acc>)
      → b<inv []>
      → xs:List a
      → b<inv xs>
```

That is, for any invariant `inv` that is inductive under `step`, if the initial value `b` is `inv`-related to the empty list, then the folded output is `inv`-related to the input list `xs`.

***We verify* `ex3`** by inferring, at the call to `foldr`

```
inv  ↦ λxs v   → v = len xs
step ↦ λx b b’ → b’ = b + 1
```

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

Previously, [26] describes a way to type `foldr` using abstract refinements that required the operator 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) *check*, and (2) *use* functions with bounded signatures? We solve both problems via a unifying insight inspired by the way typeclasses are implemented in Haskell.

1. ***A 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 Bound is Implemented*** by a ghost function that returns *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 *bounded* refinement typing to the simpler problem of *unbounded* abstract refinement typing [26]. 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 `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 `Chain` used to type `compose` in § 2.3, corresponds to a function type, yielding the translated type for `compose`:

```
type ChainTy p q r
  = x:a → w:b → z:c →
      {v:Bool | q x w ⇒ p w z ⇒ r x z}

compose :: (ChainTy p q r) → (y:b → c<p y>)
                           → (z:a → b<q z>)
                           → (x:a → c<r x>)
```

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 `compose` into:

```
compose $chain f g x =
  let t1 = g x
      t2 = f t1
      _  = $chain x t1 t2   -- materialize
  in  t2
```

In the elaborated version `$chain` is the ghost parameter corresponding to the bound. As is standard [20], 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 compose is checked to be a subtype of the specified output in an environment *strengthened* with the bound's constraint at `x`, `t1` and `t2`.

Verification of `compose` leads to a quatifier free VC:

```
    q x t1
⇒   p t1 t2
⇒  (q x t1 ⇒ p t1 t2 ⇒ r x t2)
⇒   v = t2 ⇒ r x v
```

whose first two antecedents are due to the types of `t1` and `t2` (via the output types of `g` and `f` respectively), and the third comes from the call to `$chain`. The output value `v` has the singleton refinement that states it equals to `t2`, 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 `compose`.

Our elaboration inserts materialization calls *for all* variables (of the appropriate type) that are in scope at the given point. This could introduce $O(n^k)$ 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, `ex2` is elaborated to:

```
ex2 = compose (λ_ _ _ → true) incr incr
```

This elaboration seems too naiv̈e 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 `compose`, the elaborated lambda `(λ_ _ _ → true)` is constrained to be a subtype of `ChainTy p q r`. In particular, given the call site instantiation

```
p ↦ λy z →z = y + 1
q ↦ λx y →y = x + 1
r ↦ λx z →z = x + 2
```

this subtyping constraint reduces to the quantifier-free VC:

$$\Gamma \Rightarrow \texttt{true} \Rightarrow (z = y + 1) \Rightarrow (y = x + 1) \Rightarrow (z = x + 2) \quad (3)$$

where Γ contains assumptions about the various binders in scope. A tautology, the formula is easily proved valid by an SMT solver. Proving the VC dispatches the subtyping obligation defined by the bound, in turn proving that `ex2` meets its specified type.

3. Formalism

Next we formalize Bounded Refinement Types by defining a core calculus λ_B and showing how it can be reduced to λ_P, the core language of Abstract Refinement Types [26]. We start by defining the syntax of the core calculi λ_P (§ 3.1) and λ_B (§ 3.2). Next, we provide a translation from λ_B to λ_P (3.3). Then, we prove soundness by showing that our translation is semantics preserving (§ 3.4). Finally, we comment on how we preserve decidability of the Abstract Refinement Types inference (§ 3.5).

3.1 Syntax of λ_P

We build our core language on top of λ_P, the language of Abstract Refinement Types [26]. Figure 1 summarizes the syntax of λ_P, a polymorphic λ-calculus extended with abstract refinements.

The Expressions of λ_P include the usual variables x, primitive constants c, λ-abstraction $\lambda x{:}t.e$, application $e\ e$, let bindings let $x{:}t = e$ in e, type abstraction $\Lambda\alpha.e$, and type application $e\,[t]$. (We add let-binders to λ_P from [26] as they can be reduced to λ-abstractions in the usual way.) The parameter t in the type application is a *refinement type*, as described shortly. Finally, λ_P includes refinement abstraction $\Lambda\pi : t.e$, which introduces a refinement variable π (with its type t), which can appear in refinements inside e, and the corresponding refinement application $e\,[\phi]$ that substitutes an abstract refinement with the parametric refinement ϕ, *i.e.,* refinements r closed under lambda abstractions.

$$
\begin{array}{rcll}
\textit{\textbf{Expressions}} & e & ::= & x \mid c \mid \lambda x{:}t.e \mid e\ x \\
 & & \mid & \mathsf{let}\ x{:}t = e\ \mathsf{in}\ e \\
 & & \mid & \Lambda\alpha.e \mid e\,[t] \\
 & & \mid & \Lambda\pi : t.e \mid e\,[\phi] \\
\textit{\textbf{Constants}} & c & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{crash} \\
 & & \mid & 0 \mid 1 \mid -1 \mid \ldots \\
\textit{\textbf{Parametric Refinements}} & \phi & ::= & r \mid \lambda x{:}b.\phi \\
\textit{\textbf{Predicates}} & p & ::= & c \mid \neg p \mid p = p \mid \ldots \\
\textit{\textbf{Atomic Refinements}} & a & ::= & p \mid \pi\ \overline{x} \\
\textit{\textbf{Refinements}} & r & ::= & a \mid a \wedge r \mid a \Rightarrow r \\
\textit{\textbf{Basic Types}} & b & ::= & \mathtt{Int} \mid \mathtt{Bool} \mid \alpha \\
\textit{\textbf{Types}} & t & ::= & \{v : b \mid r\} \\
 & & \mid & \{v : (x : t) \rightarrow t \mid r\} \\
\textit{\textbf{Bounded Types}} & \rho & ::= & t \\
\textit{\textbf{Schemata}} & \sigma & ::= & \rho \mid \forall\alpha.\sigma \mid \forall\pi : t.\sigma
\end{array}
$$

Figure 1. Syntax of λ_P

$$
\begin{array}{rcll}
\textit{\textbf{Bounded Types}} & \rho & ::= & \tau \mid \{\phi\} \Rightarrow \rho \\
\textit{\textbf{Expressions}} & e & ::= & \ldots \mid \Lambda\{\phi\}.e \mid e\{\phi\}
\end{array}
$$

Figure 2. Extending Syntax of λ_P to λ_B

The Primitive Constants of λ_P include *true*, *false*, `0`, `1`, `-1`, *etc.*. In addition, we include a special untypable constant *crash* that models "going wrong". Primitive operations return a crash when invoked with inputs outside their domain, *e.g.,* when `/` is invoked with `0` as the divisor, or when an `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 [28]. An *abstract refinement* $\pi\ \overline{x}$ is an application of a refinement variable π 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 ϕ (§ 3.5).

The Types of λ_P written t include basic types, dependent functions and schemata quantified over type and refinement variables α and π respectively. A basic type is one of `Int`, `Bool`, or a type variable α. A refined type t is either a refined basic type $\{v : b \mid r\}$, or a dependent function type $\mid$ $\{v : (x : t) \rightarrow t \mid r\}$ 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 λ_P bounded types ρ are just a synonym for types t. Finally, schemata are obtained by quantifying bounded types over type and refinement variables.

3.2 Syntax of λ_B

Figure 2 shows how we obtain the syntax for λ_B by extending the syntax of λ_P with *bounded* types.

The Types of λ_B extend those of λ_P with bounded types ρ, which are the types t guarded by bounds ϕ.

The Expressions of λ_B extend those of λ_P with *abstraction* over bounds $\Lambda\{\phi\}.e$ and *application* of bounds $e\{\phi\}$. Intuitively, if an expression e has some type ρ 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 $\overline{\lambda x : \rho.\mathit{true}}$.

Notation. We write b, $b\langle\pi\ \overline{x}\rangle$, $\{v : b\langle\pi\ \overline{x}\rangle \mid r\}$ to abbreviate $\{v : b \mid \mathit{true}\}$, $\{v : b \mid \pi\ \overline{x}\ v\}$, $\{v : b \mid r \wedge \pi\ \overline{x}\ v\}$ respectively. We say a type or schema is *non-refined* if all the refinements in it are *true*. We get the *shape* τ (*i.e.,* the System-F type) of a type t by the function $\mathsf{Shape}(t)$ defined:

$$
\begin{aligned}
\mathsf{Shape}(\{v : b \mid r\}) &\doteq b \\
\mathsf{Shape}(\{v : (x : t_1) \rightarrow t_2 \mid r\}) &\doteq \mathsf{Shape}(t_1) \rightarrow \mathsf{Shape}(t_2)
\end{aligned}
$$

3.3 Translation from λ_B to λ_P

Next, we show how to translate a term from λ_B to one in λ_P. We assume, without loss of generality that the terms in λ_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 Φ 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 Γ. 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 3 via a relation $\Gamma; \Phi \vdash e \leadsto e'$, that translates the expression e in λ_B into e' in λ_P. Most of the rules in figure 3 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 ϕ into a bound-function parameter of a suitable type,
2. ***At variable binding sites*** *i.e.,* λ- 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.

1. Rule **CAbs** translates bound abstractions $\Lambda\{\phi\}.e$ into a plain λ-abstraction. In the translated expression $\lambda f : \langle\!|\phi|\!\rangle.e'$ the bound becomes a function named f with type $\langle\!|\phi|\!\rangle$ defined:

$$
\begin{aligned}
\langle\!|\lambda x : b.\phi|\!\rangle &\doteq (x : b) \rightarrow \langle\!|\phi|\!\rangle \\
\langle\!|r|\!\rangle &\doteq \{v : \mathtt{Bool} \mid r\}
\end{aligned}
$$

That is, $\langle\!|\phi|\!\rangle$ is a function type whose final output carries the refinement corresponding to the constraint in ϕ. Note that the translation generates a fresh name f for the bound function (ensuring that it cannot be used in the regular code) and saves it in the bound environment Φ to let us materialize the bound constraint when translating the body e of the abstraction.

2. Rules **Fun** ***and*** **Let** materialize bound constraints at variable binding sites (λ-abstractions and let-bindings respectively.) If we

$$\begin{array}{rrcl}\textbf{\textit{Variable Environment}} & \Gamma & ::= & \emptyset \mid \Gamma, x:\tau \\ \textbf{\textit{Bound Environment}} & \Phi & ::= & \emptyset \mid \Phi, x:\tau\end{array}$$

Translation $\boxed{\Gamma; \Phi \vdash e \rightsquigarrow e}$

$$\frac{}{\Gamma; \Phi \vdash x \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\Gamma; \Phi \vdash c \rightsquigarrow c}\ \textsc{Con}$$

$$\frac{\Gamma' = \Gamma, x:\mathsf{Shape}(t) \qquad \Gamma'; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \lambda x:t.e \rightsquigarrow \lambda x:t.\mathsf{Inst}(\Gamma', \Phi, e')}\ \textsc{Fun}$$

$$\frac{\Gamma; \Phi \vdash e_x \rightsquigarrow e'_x \qquad \Gamma' = \Gamma, x:\mathsf{Shape}(t) \qquad \Gamma'; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \mathsf{let}\ x:t = e_x\ \mathsf{in}\ e \rightsquigarrow \mathsf{let}\ x:\tau = e'_x\ \mathsf{in}\ \mathsf{Inst}(\Gamma', \Phi, e')}\ \textsc{Let}$$

$$\frac{\Gamma; \Phi \vdash e_1 \rightsquigarrow e'_1 \qquad \Gamma; \Phi \vdash e_2 \rightsquigarrow e'_2}{\Gamma; \Phi \vdash e_1\ e_2 \rightsquigarrow e'_1\ e'_2}\ \textsc{App}$$

$$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda\alpha.e \rightsquigarrow \Lambda\alpha.e'}\ \textsc{TAbs} \qquad \frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash e\,[t] \rightsquigarrow e'\,[t]}\ \textsc{TApp}$$

$$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda\pi : t.e \rightsquigarrow \Lambda\pi : t.e'}\ \textsc{PAbs}$$

$$\frac{\Gamma; \Phi \vdash e_1 \rightsquigarrow e'_2 \qquad \Gamma; \Phi \vdash e_1 \rightsquigarrow e'_2}{\Gamma; \Phi \vdash e_1\,[e_2] \rightsquigarrow e'_1\,[e'_2]}\ \textsc{PApp}$$

$$\frac{\text{fresh } f \qquad \Gamma; \Phi, f:\mathsf{Shape}(\langle\!|\phi|\!\rangle) \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda\{\phi\}.e \rightsquigarrow \lambda f : \langle\!|\phi|\!\rangle.e'}\ \textsc{CAbs}$$

$$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash e\{\phi\} \rightsquigarrow e'\ \mathsf{Const}(\phi)}\ \textsc{CApp}$$

Figure 3. Translation Rules from λ_B to $\lambda_{P+\mathtt{let}}$

view the bounds as universally quantified constraints over the (abstract) refinements, then our translation exhaustively and eagerly *instantiates* the constraints at each point that a new binder is introduced into the variable environment, over all the possible candidate sets of variables in scope at that point. The instantiation is performed by $\mathsf{Inst}(\Gamma, \Phi, e)$

$$\begin{array}{rcl}\mathsf{Inst}(\Gamma, \Phi, e) & \doteq & \mathsf{Wrap}(e, \mathsf{Instances}(\Gamma, \Phi)) \\ \mathsf{Wrap}(e, \{e_1, \ldots, e_n\}) & \doteq & \mathsf{let}\ t_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ t_n = e_n\ \mathsf{in}\ e \\ & & \text{(where } t_i \text{ are fresh } \mathtt{Bool} \text{ binders)} \\ \mathsf{Instances}(\Gamma, \Phi) & \doteq & \{\ f\ \overline{x} \mid f:\tau \leftarrow \Phi,\ \overline{x:\_} \leftarrow \Gamma \\ & & \quad ,\ \Gamma, f:\tau \vdash_B f\ \overline{x}:\mathtt{Bool}\}\end{array}$$

The function takes the environments Γ and Φ, an expression e and a variable x of type t and uses let-bindings to materialize all the bound functions in Φ that accept the variable x. Here, $\Gamma \vdash_B e:\tau$ 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 λ abstractions that witness that the bound constraints hold. That is, as within e, bounds are translated to a bound function (parameter) of type $\langle\!|\phi|\!\rangle$, we translate ϕ into a λ-term that, via subtyping must have the required type $\langle\!|\phi|\!\rangle$. We construct such a function via $\mathsf{Const}(\phi)$ that depends only on the *shape* of the bound, *i.e.,* the non-refined types of its parameters (and not the actual constraint itself).

$$\begin{array}{rl}\mathsf{Const}(r) & \doteq \mathit{true} \\ \mathsf{Const}(\lambda x:b.\phi) & \doteq \lambda x:b.\mathsf{Const}(\phi)\end{array}$$

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 λ_P term will verify that in the context in which the above constant function is created, the singleton *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 `ex2` of the overview (§ 2.4) the subtyping constraint that decides is the constant *true* is a valid bound reduces to the equation 3 that is a tautology.

3.4 Soundness

The Small-Step Operational Semantics of λ_B are defined by extending a similar semantics for λ_P which is a standard call-by-value calculus where abstract refinements are boolean valued functions [26]. Let $\hookrightarrow_P$ denote the transition relation defining the operational semantics of λ_P and $\hookrightarrow^\star_P$ denote the reflexive transitive closure of $\hookrightarrow_P$. We thus obtain the transition relation $\hookrightarrow_B$:

$$(\Lambda\{\phi\}.e)\{\phi\} \hookrightarrow_B e \qquad\qquad e \hookrightarrow_B e', \text{if } e \hookrightarrow_P e'$$

Let $\hookrightarrow^\star_B$ denote the reflexive transitive closure of $\hookrightarrow_B$.

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

Lemma. *If $\emptyset; \emptyset \vdash e \rightsquigarrow e'$ and $e \hookrightarrow^\star_B c$ then $e' \hookrightarrow^\star_P c$.*

The Soundness of λ_B follows by combining the above Semantics Preservation Lemma with the soundness of λ_P as stated in [26]:

Lemma (Soundness of λ_P)**.** *If $\emptyset \vdash e : \sigma$ then $e \not\hookrightarrow^\star_P$ crash.*

To Typecheck a λ_B ***program*** e we translate it into a λ_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 $\emptyset; \emptyset \vdash e \rightsquigarrow e'$ and $\emptyset \vdash e' : \sigma$ then $e \not\hookrightarrow^\star_B$ crash.*

3.5 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 κ denote the unknown refinements that an abstract refinement parameter π 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* [4], which in this case means that the κ variables, and correspondingly, the abstract refinements π must only appear in *positive* positions within refinements (*i.e.,* not under logical negations). The syntax of refinements shown in Figure 1 violates this requirement via refinements of the form $\pi\ \overline{x} \Rightarrow r$.

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:

$$\Gamma \vdash \{v{:}b \mid a\} \preceq \{v{:}b \mid a_1 \Rightarrow \cdots \Rightarrow a_n \Rightarrow a_q\}$$

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:

$$\Gamma, \{x_1{:}b_1 \mid a'_1\}, \ldots, \{x_n{:}b_n \mid a'_n\} \vdash \{v{:}b \mid a'\} \preceq \{v{:}b \mid a'_q\}$$

Title	Director	Year	Star
“Birdman”	“Iñárritu”	2014	8.1
“Persepolis”	“Paronnaud”	2007	8.0

Figure 4. Example Table of Movies

where all the abstract refinements variables π (and hence instance variables κ) appear positively, ensuring that the constraints are monotonic, hence permitting inference via Liquid Typing [20].

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 *tables*, that are a collection of *rows*, which in turn are *records* that represent a unit of data stored in the table. The tables's *schema* describes the types of the values in each row of the table. For example, the table in Figure 4 organizes information about movies, and has the schema:

```
Title:String, Dir:String, Year:Int, Star:Double
```

First, we show how to write type safe extensible records that represent rows, and use them to implement database tables (§ 4.1). 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 valuation of the column in a particular row.

A dictionary `Dict <r> k v` maps a key `x` of type `k` to a value of type `v` that satisfies the property `r x`

```
type Range k v = k → v → Bool

data Dict k v <r :: Range k v> = D {
    dkeys :: [k]
  , dfun  :: x:{k | x ∈ elts dkeys} → v<r x>
  }
```

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

Propositions about the theory of sets can be decided efficiently by modern SMT solvers. Hence we use such propositions within refinements [27]. The measures (logical functions) `elts` and `keys` specify the set of keys in a list and a dictionary respectively:

```
elts        :: [a] → Set a
elts ([])   = ∅
elts (x:xs) = {x} ∪ elts xs

keys        :: Dict k v → Set k
keys d      = elts (dkeys d)
```

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:

```
type RD k v <dom :: Domain k v, rng :: Range k v>
  = {v:Dict <rng> k v | dom v}

type Domain k v = Dict k v → Bool
```

We may instantiate `dom` and `rng` with predicates that precisely describe the values contained with the dictionary. For example,

```
RD<λd → keys d = {"x"}, λk v→ 0 < v> String Int
```

describes dictionaries with a single field `"x"` whose value (as determined by `dfun`) is stricly greater than 0. We will define schemas by appropriately instantiating the abstract refinements `dom` and `rng`.

An empty dictionary has an empty domain and a function that will never be called:

```
empty   :: RD <emptyRD, rFalse> k v
empty   = D [] (λx → error "calling empty")

emptyRD = λd → keys d == ∅
rFalse  = λk v → false
```

We define singleton maps as dependent pairs `x := y` which denote the mapping from `x` to `y`:

```
data P k v <r :: Range k v>
  = (:=) {pk :: k, pv :: v<r pk>}
```

Thus, `key := val` has type `P<r> k v` only if `r key val`.

A dictionary may be extended with a singleton binding (which maps the new key to its new value).

```
(+=)    :: bind:P<r> k v
        → dict:RD<pTrue, r> k v
        → RD <addKey (pk bind) dict, r> k v

(k := v) += (D ks f)
        = D (k:ks)
            (λi → if i == k then v else f i)

addKey = λk d d’ → keys d’ = {k} ∪ keys d
pTrue  = λ_ → true
```

Thus, `(k := v) += d` evaluates to a dictionary `d’` that extends `d` with the mapping from `k` to `v`. The type of `(+=)` constrains the new binding `bind`, the old dictionary `dict` and the returned value to have the same range invariant `r`. The return type states that the output dictionary’s domain is that of the domain of `dict` extended by the new key `(pk bind)`.

To model a row in a table *i.e.,* a schema, we define the unrefined (Haskell) type `Schema`, which is a dictionary mapping `String`s, *i.e.,* the names of the fields of the row, to elements of some universe `Univ` containing `Int`, `String` and `Double`. (A closed universe is not a practical restriction; most databases support a fixed set of types.)

```
data Univ   = I Int | S String | D Double

type Schema = RD String Univ
```

We refine Schema with concrete instantiations for `dom` and `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 4:

```
type MovieSchema = RD <md, mr> String Univ

md = λd →
      keys d={"year","star","dir","title"}
mr = λk v →
      k = "year" ⇒ isI v && 1888 < toInt v
   && k = "star" ⇒ isD v && 0 ≤ toDouble v ≤ 10
   && k = "dir"  ⇒ isS v && k = "title" ⇒ isS v

isI (I _)    = True
isI _        = False

toInt        :: {v: Univ | isI v} → Int
toInt (I n) = n
...
```

The predicate `md` describes the *domain* of the movie schema, restricting the keys to exactly `"year"`, `"star"`, `"dir"`, and `"title"`. The range predicate `mr` describes the types of the values in the schema: a dictionary of type `MovieSchema` must map `"year"` to an Int, "star" to a Double, and "dir" and "title" to Strings. The range predicate may be used to impose additional constraints on the values stored in the dictionary. For instance, mr restricts the year to be not only an integer but also greater than 1888.

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 4

```
movie1, movie2 :: MovieScheme
movie1 = ("title" := S "Persepolis")
      += ("dir"   := S "Paronnaud")
      += ("star"  := D 8)
      += ("year"  := I 2007)
      += empty

movie2 = ("title" := S "Birdman")
      += ("star"  := D 8.1)
      += ("dir"   := S "Inarritu")
      += ("year"  := I 2014)
      += empty
```

Typing movie1 (and movie2) as MovieSchema boils down to proving: That keys movie1 = {"year", "star", "dir", "title"}; and that each key is mapped to an appropriate value as determined by mr. For example, declaring movie1's year to be I 1888 or even misspelling "dir" as "Dir" will cause the movie1 to become ill-typed. As the (sub)typing relation depends on logical implication (unlike in HList based approaches [11]) key ordering *does not* affect type-checking; in movie1 the star field is added before the director, while movie2 follows the opposite order.

Database Tables are collections of rows, *i.e.,* collections of refined dictionaries. We define a type alias RT s (Refined Table) for the list of refined dictionaries from the field type String to the Universe.

```
type RT (s :: {dom::TDomain, rng::TRange})
  = [RD <s.dom, s.rng> String Univ]

type TDomain = Domain String Univ
type TRange  = Range  String Univ
```

For brevity we pack both the domain and the range refinements into a record s that describes the schema refinement; *i.e.,* each row dictionary has domain s.dom and range s.rng.

For example, the table from Figure 4 can be represented as a type MoviesTable which is an RT refined with the domain and range md and mr described earlier (§ 4.1):

```
type MoviesTable = RT {dom = md, rng = mr}

movies :: MovieTable
movies = [movie1, movie2]
```

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 [23].

```
union   :: RT s → RT s → RT s
diff    :: RT s → RT s → RT s
select  :: (RD s → Bool) → RT s → RT s
project :: ks:[String] → RTSubEqFlds ks s
        → RTEqFlds ks s
product :: ( Disjoint s1 s2, Union s1 s2 s
           , Range s1 s, Range s2 s)
        ⇒ RT s1 → RT s2 → RT s
```

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 that satisfy p. 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 RTSubEqFlds ks 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 RTSubEqFlds ks s
  = RT s{dom = λz → elts ks ⊆ keys z}

type RTEqFlds ks s
  = RT s{dom = λz → elts ks = 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 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 s2) 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 = λx y →
  s1.dom x ⇒ s2.dom y ⇒ keys x ∩ keys y = ∅

bound Union s1 s2 s = λx y v →
  s1.dom x ⇒ s2.dom y ⇒keys v = keys x ∪ keys y
    ⇒ s.dom v

bound Range si s = λx k v →
  si.dom x ⇒ k ∈ keys x ⇒ si.rng k v
    ⇒ 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 [1]

```
good_titles = project ["title"] $ select (λd →
                 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

[1] More example queries can be found online [23]

state (§ 5.1), (2) extend `RIO` with a set of combinators for *imperative* programming, *i.e.,* whose types precisely encode Floyd-Hoare style program logics (§ 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 (§ 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 [26])

```
  type Pre    = World → Bool
  type Post a = World → a → World → Bool

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

That is, `RIO a` is a function `World→(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 [7] (also used in [15]).

Our Post-conditions are Two-State Predicates that relate the input- and output- world (as in [15]). 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 → RIO <p, ret z> a
   return z = RIO $ λw → (z, w)

   ret z    = λw x w' → w' = w && x == z
```

The type of `return` states that for any precondition `p` and any supplied value `z` of type `a`, the expression `return z` is an `RIO` computation with precondition `p` and a post-condition `ret z`. The postcondition states that: (1) the output `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.

```
  (>>=) :: (Ret q1 r, Seq r q1 p2, Trans q1 q2 q)
        ⇒ m:RIO <p, q1> a
        → k:(x:a<r> → RIO <p2 x, q2 x> b)
        → RIO <p, q> b

  (RIO g) >>= f = RIO $ λx →
    case g x of { (y, s) → runState (f y) s }
```

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`;

```
  bound Ret q1 r = λw x w' → q1 w x w' ⇒ 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 depend upon the supplied value `x`):

```
bound Seq q1 p2 = λw x w' → q1 w x w' ⇒ p2 x w'
```

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

```
  bound Trans q1 q2 q = λw x w' y w'' →
    q1 w x w' ⇒ q2 x w' y w'' ⇒ q w y w''
```

Both type signatures would be impossible to use if the programmer had to manually instantiate the abstract refinements (*i.e.,* pre- and post-conditions.) Fortunately, Liquid Type inference generates the instantiations making it practical to use LIQUIDHASKELL 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:

```
    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 axiom:

```
select :: Map k v → k → v
update :: Map k v → k → v → Map k v

∀ 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 `ctr` via:

```
    getCtr   :: RIO <pTrue, rdCtr> Int
    getCtr   = RIO $ λw → (ctr w, w)

    setCtr   :: Int → RIO <pTrue, wrCtr n> ()
    setCtr n = RIO $ λw → ((), w { ctr = n })
```

Here, the refinements are defined as:

```
pTrue    = λw → True
rdCtr    = λw x w' → w' = w && x = select w ctr
wrCtr n = λw _ w' → w' = update w 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:

```
bound Pure g = λw x v  →(g w x v ⇒ v = w)
bound Then g p1 = λw v → (g w True  v ⇒ p1 v)
bound Else g p2 = λw v → (g w False v ⇒ p2 v)

    ifM :: (Pure g, Then g p1, Else g p2)
        ⇒ RIO <p , g> Bool       -- cond
        → RIO <p1, q> a          -- then
        → RIO <p2, q> a          -- else
        → RIO <p , q> a
```

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, LIQUIDHASKELL verifies that `div` is called safely:

```
div :: Int → {v:Int | v /= 0} → Int

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

Verification succeeds as the post-condition of `nonZero` is instantiated to `λ_ b w →b ⇔ select w ctr /= 0` and the precondition of `divX`'s is instantiated to `λw →select w 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, Inv p g b, Exit p g q)
       ⇒ RIO <p, g> Bool       -- cond
       → RIO <pTrue, b> ()     -- body
       → RIO <p, q> ()
```

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`:

```
bound Inv p g b = λw w’ w’’ →
  p w ⇒ g w True w’ ⇒ b w’ () w’’ ⇒ p w’’
```

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

```
bound Exit p g q = λw w’ →
  p w ⇒ g w False w’ ⇒ q w () 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):

```
bound OneState q = λw w’ w’’ →
   q w () w’’ ⇒ q w’ () w’’
```

***We can use* `whileM`** to implement a loop that repeatedly decrements a counter while it is positive, and to then verify that if were initially non-negative, then at the end the counter is exactly equal to `0`.

```
whileTest   :: RIO <posCtr, zeroCtr> ()
whileTest = whileM gtZeroX decr
  where gtZeroX = getCtr >>= return . (> 0)

posCtr  = λw → 0 ≤ select w ctr
zeroCtr = λ_ _ w’ → 0 = select w ctr
```

Where the decrement is implemented by `decr` with type:

```
decr :: RIO <pTrue, decCtr> ()

decCtr = λw _ w’ →
  w’ = update w ctr ((select ctr w) - 1)
```

LIQUIDHASKELL verifies that at the end of `whileTest` the counter is zero (*i.e.,* the post-condition `zeroCtr`) by instantiating suitable (*i.e.,* inductive) refinements for this particular use of `whileM`.

```
pread, pwrite, plookup, pcontents,
pcreateD, pcreateF, pcreateFP :: Priv → Bool

active   :: World → Set FH
caps     :: World → Map FH Priv

pset p h = λw → p (select (caps w) h) &&
                h ∈ active w
```

Figure 5. Privilege Specification

6. Capability Safe Scripting via `RIO`

Next, we describe how we use the `RIO` monad to reason about shell scripting, inspired by the `Shill` [14] programming language.

`Shill` is a scripting language that restricts the privileges with which a script may execute by using *capabilities* and *dynamic contract checking* [14] . 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` runtime 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 [14], how can we use Bounded Refinement Types to encode scripting privileges and reason about them *statically?*

We use* `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 (`World`) of the `RIO` monad with a *privilege specification* denoting the set of privileges that a program may use (§ 6.1); specify the *file system API* in terms of this abstraction (§ 6.2); and use the above to specify and verify the particular privileges that a *client* of the API uses (§ 6.3).

6.1 Privilege Specification

Figure 5 summarizes how we specify privileges inside `RIO`. We use the type `FH` to denote a file handles, analogous to `Shill`'s capabilities. An abstract type `Priv` denotes the sets of privileges that may be associated with a particular `FH`.

To connect* `World`*s with Privileges we assume a set of uninterpreted functions of type `Priv → Bool` that act as predicates on values of type `Priv`, each denoting a particular privilege. For example, given a value `p :: Priv`, the proposition `pread p` denotes that `p` includes the "read" privilege. The function `caps` associates each `World` with a `Map FH Priv`, a table that associates each `FH` with its privileges. The function `active` maps each `World` to the `Set` of allocated `FH`s. Given `x:FH` and `w:World`, `pwrite (select (caps w)x)` denotes that in the state `w`, the file `x` may be written. This pattern is generalized by the predicate `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 `eqP w w’` that says that the set of privileges and active handles in worlds `w` and `w’` are *equivalent.*

```
eqP = λw _ w’ →
  caps w == caps w’ && active w == active w’
```

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

```
read :: {- Read the contents of h -}
  h:FH → RIO<pset pread h, eqp> String

write :: {- Write to the file h -}
  h:FH → String → RIO<pset pwrite h, eqp> ()

contents :: {- List the children of h -}
  h:FH → RIO<pset pcontents h, eqp> [Path]
```

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

```
alloc w' w x =
  (x ∉ active w) && active w' = {x} ∪ active w
```

which says that the active handles in (the new `World`) `w'` are those of (the old `World`) `w` extended with the hitherto *inactive* handle `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`:

```
derivP h = λw x w' →
  alloc w' w x &&
  caps w' = store (caps w) x
                  (pcreateFP (select (caps w)) h)

create :: {- Create a file -}
  h:FH→Path→RIO<pset pcreateF h, derivP h> 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'` (`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:

```
copyP h = λw x w' →
  alloc w' w x &&
  caps w' = store (caps w) x (select (caps w) y)

lookup :: {- Open a child of h -}
  h:FH→Path→RIO<pset plookup h, copyP h> FH

createDir :: {- Create a directory -}
  h:FH→Path→RIO<pset pcreateD h, copyP h> 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`.

```
copyRec recur s d =
  do cs <- contents s
     forM_ cs $ λ p → do
       x <- flookup s p
       when (isFile x) $ do
         y <- create d p
         s <- fread x
         write y s
       when (recur && (isDir x)) $ do
         y <- createDir d p
         copyRec recur x y
```

`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:

```
copyRec :: Bool → s:FH → d:FH →
           RIO<copySpec s d,
               λ_ _ w → copySpec s d w> ()

copySpec h d = λw →
  pset pcontents h w && pset plookup h     w &&
  pset pread h     w && pset pcreateFile d w &&
  pset pwrite d    w && pset pcreateF d    w &&
  pwrite (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` (`pcreateF`), 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 priviliges 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:

```
bound Stable i = λx y w w' →
  i w ⇒ ( eqP w () w' || copyP y w x w'
         || derivP y w x w'
        ) ⇒ i 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:

```
copyFrame i s d = λw → i w && copySpec s d w

copyRec :: (Stable i) ⇒
           Bool → s:FH → d:FH →
           RIO<copyFrame i s d,
               λ_ _ w → copyFrame i s d w> ()
```

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 `λw →copySpec s d w`.

7. Related Work

Higher order Logics and Dependent Type Systems including Coq [3], Agda [17], and even Haskell [13, 19], are highly expressive. However, in these settings, checking requires explicit user-provided proofs. 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 undecidable (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 [21] and *indexed types* (DML [29]) where types are constrained by predicates drawn from a logic. To ensure decidable checking several refinement type systems including [5, 28, 29] restrict refinements to decidable, quantifier free logics. While this ensures predictable checking and inference [20] 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 [10] 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 [26], a form of refinement polymorphism analogous to parametric polymorphism. While [26] 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 [12] showed how phantom types could be used to eliminate certain classes of errors. Haskell's HList library [11] 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-classes for new operations (*e.g.,* appending lists); [18] shows how a dependently typed language greatly simplifies the implementation. Much of this simplicity can be recovered in Haskell using the `singleton` library [6]. Our goal is to show that bounded refinements are expressive enough to permit 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 [7], which was later enriched with separation logic in Ynot [15]. In future work it would be interesting 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 [22] 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.

References

[1] C. Barrett, A. Stump, and C. Tinelli. `http://smt-lib.org`.

[2] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 2011.

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

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs. In *POPL*, 1977.

[5] J. Dunfield. Refined typechecking with Stardust. In *PLPV*, 2007.

[6] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell*, 2012.

[7] J.C. Filliâtre. Proof of imperative programs in type theory. In *TYPES*, 1998.

[8] C. Fournet, M. Kohlweiss, and P-Y. Strub. Modular code-based cryptographic verification. In *CCS*, 2011.

[9] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE S & P*, 2011.

[10] G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. In *ICFP*, 2014.

[11] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell*, 2004.

[12] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, 1999.

[13] C. McBride. Simulating dependent types in haskell. In *JFP*, 2002.

[14] S. Moore, C. Dimoulas, D. King, and S. Chong. SHILL: A secure shell scripting language. In *OSDI*, 2014.

[15] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.

[16] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[17] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.

[18] N. Oury and W. Swierstra. The power of Pi. In *ICFP*, 2008.

[19] S. L. Peyton-Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, 2006.

[20] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[21] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.

[22] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, 2013.

[23] UCSD Programming Systems. `github.com/ucsd-progsys/liquidhaskell/tree/master/benchmarks/icfp15`.

[24] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[25] H. Unno, T. Terauchi, and N. Kobayashi. Relatively complete verification of higher-order functional programs. In *POPL*, 2013.

[26] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

[27] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell*, 2014.

[28] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones. Refinement types for haskell. In *ICFP*, 2014.

[29] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.