

Bounded Refinement Types Extended Version

Niki Vazou Alexander Bakst Ranjit Jhala
UC San Diego

Abstract

We present *Bounded Refinement Types* which enable *bounded quantification* over refinements. We extend Abstract Refinement Types with bounds *i.e.*, Horn implications that allow *dependencies between* abstract refinements. We translate bounds into “ghost” functions, preserving the decidable type checking and inference of the “unbounded” setting of Abstract Refinements [27]. We implemented Bounded Refinement Types in LIQUIDHASKELL and proved safety of a relational algebra and stateful computations.

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 [22, 30], 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 [21], termination and totality checking [28], protocol validation [1, 9], and securing web applications [10]. 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 [27]. We preserved decidability of checking and inference by encoding abstractly refined types with uninterpreted functions obeying the decidable axioms of congruence [17]. 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 [27]. Consequently, checking remains decidable and, 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-granularity, ensuring that functions only access specified resources (§ 6).

We have implemented Bounded Refinement Types in LIQUIDHASKELL [29]. The source code of the examples, with more verbose concrete syntax is at [24]. While the construction of these verified abstractions is possible with full dependent types our encoding makes it possible, for the first time, to simultaneously use SMT solvers and 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 current environment:

```

a:Int, b:IntGE a, cmp:{Bool | cmp ⇔ a ≤ b}
⊢ {v : Bool | v = cmp} ≤ TRUE

```

Intuitively, the relation $G \vdash T \leq U$ may be understood as “assuming the type bindings in G , T is a subtype of U ” [27]. Subtyping between refinement types reduces to checking the validity of the *logical implication* between the sub- and supertype’s refinements in the current environment. The above subtyping check reduces to the following *verification condition* (VC)

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

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

```

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

negMax :: List Neg → Neg
negMax = 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, $\text{Int}<p>$ stands for $\{v:\text{Int} \mid p \ v\}$, 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 .

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, [27] 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:

$$\forall x \ y. \ x = y \Rightarrow p \ x \Leftrightarrow p \ y$$

thereby preserving decidable checking [17].

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` and `negMax` by instantiating:

$$\begin{aligned} p &\mapsto \lambda v \rightarrow 0 < v && \text{-- at posMax} \\ p &\mapsto \lambda v \rightarrow v < 0 && \text{-- at negMax} \end{aligned}$$

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. `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:

$$(\text{Int} \rightarrow \text{Bool}) \rightarrow (\text{Int}<p> \rightarrow a) \rightarrow \text{Int}<p> \rightarrow a$$

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

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

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, $\{v:\text{Int} \mid v = i + 1\}$ is in general *not* a subtype of $\text{Int}<p>$, as the associated VC

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

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 $\forall x. p\ x \Rightarrow p\ (x+1)$, 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 ($n \leq x \Rightarrow n \leq x + 1$) 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 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 specifying 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 $\{v: c \mid r\ x\ v\}$.)

```
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 [25] 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:

```
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 [29]:

```
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, [27] 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 [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 `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 [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 `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 quantifier 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 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 `compose`, the elaborated lambda $(\lambda_ _ _ \rightarrow \text{true})$ is constrained to be a subtype of $\text{ChainTy } p \ q \ r$. 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:

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

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 [27]. 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 [27]. 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

Expressions	$e ::= x \mid c \mid \lambda x:t.e \mid e\ x$ $\quad \quad \quad \mid \text{let } x:t = e \text{ in } e$ $\quad \quad \quad \mid \Lambda\alpha.e \mid e[t]$ $\quad \quad \quad \mid \Lambda\pi:t.e \mid e[\phi]$
Constants	$c ::= \text{true} \mid \text{false} \mid \text{crash}$ $\quad \quad \quad \mid 0 \mid 1 \mid -1 \mid \dots$
Parametric Refinements	$\phi ::= r \mid \lambda x:b.\phi$
Predicates	$p ::= c \mid \neg p \mid p = p \mid \dots$
Atomic Refinements	$a ::= p \mid \pi \bar{x}$
Refinements	$r ::= a \mid a \wedge r \mid a \Rightarrow r$
Basic Types	$b ::= \text{Int} \mid \text{Bool} \mid \alpha$
Types	$t ::= \{v : b \mid r\}$ $\quad \quad \quad \mid \{v : (x : t) \rightarrow t \mid r\}$
Bounded Types	$\rho ::= t$
Schemata	$\sigma ::= \rho \mid \forall\alpha.\sigma \mid \forall\pi:t.\sigma$

Figure 1. Syntax of λ_P

Bounded Types	$\rho ::= \tau \mid \{\phi\} \Rightarrow \rho$
Expressions	$e ::= \dots \mid \Lambda\{\phi\}.e \mid e\{\phi\}$

Figure 2. Extending Syntax of λ_P to λ_B

let $x:t = e$ in e , type abstraction $\Lambda\alpha.e$, and type application $e[t]$. (We add let-binders to λ_P from [27] 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.

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 [29]. An *abstract refinement* $\pi \bar{x}$ 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.

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 $\{v : (x : t) \rightarrow t \mid r\}$ 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 $\lambda x:\rho.\text{true}$.

Notation. We write b , $b\langle\pi \bar{x}\rangle$, $\{v : b\langle\pi \bar{x}\rangle \mid r\}$ to abbreviate $\{v : b \mid \text{true}\}$, $\{v : b \mid \pi \bar{x} v\}$, $\{v : b \mid r \wedge \pi \bar{x} v\}$ 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 $\text{Shape}(t)$ defined:

$$\text{Shape}(\{v : b \mid r\}) \doteq b$$

$$\text{Shape}(\{v : (x : t_1) \rightarrow t_2 \mid r\}) \doteq \text{Shape}(t_1) \rightarrow \text{Shape}(t_2)$$

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, \dots, 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 $\Gamma; \Phi \vdash e \rightsquigarrow e'$, 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:

$$\langle\lambda x:b.\phi\rangle \doteq (x : b) \rightarrow \langle\phi\rangle$$

$$\langle r \rangle \doteq \{v : \text{Bool} \mid r\}$$

Variable Environment	$\Gamma ::= \emptyset \mid \Gamma, x:\tau$
Bound Environment	$\Phi ::= \emptyset \mid \Phi, x:\tau$
Translation	$\boxed{\Gamma; \Phi \vdash e \rightsquigarrow e}$
$\frac{}{\Gamma; \Phi \vdash x \rightsquigarrow x}$	VAR
$\frac{}{\Gamma; \Phi \vdash c \rightsquigarrow c}$	CON
$\frac{\Gamma' = \Gamma, x:\text{Shape}(t) \quad \Gamma'; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \lambda x:t.e \rightsquigarrow \lambda x:t.\text{Inst}(\Gamma', \Phi, e')}$	FUN
$\frac{\Gamma; \Phi \vdash e_x \rightsquigarrow e'_x \quad \Gamma' = \Gamma, x:\text{Shape}(t) \quad \Gamma'; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \text{let } x:t = e_x \text{ in } e \rightsquigarrow \text{let } x:\tau = e'_x \text{ in } \text{Inst}(\Gamma', \Phi, e')}$	LET
$\frac{\Gamma; \Phi \vdash e_1 \rightsquigarrow e'_1 \quad \Gamma; \Phi \vdash e_2 \rightsquigarrow e'_2}{\Gamma; \Phi \vdash e_1 e_2 \rightsquigarrow e'_1 e'_2}$	APP
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda \alpha.e \rightsquigarrow \Lambda \alpha.e'}$	TABS
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash e[t] \rightsquigarrow e'[t]}$	TAPP
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash \Lambda \pi : t.e \rightsquigarrow \Lambda \pi : t.e'}$	PABS
$\frac{\Gamma; \Phi \vdash e_1 \rightsquigarrow e'_1 \quad \Gamma; \Phi \vdash e_2 \rightsquigarrow e'_2}{\Gamma; \Phi \vdash e_1[e_2] \rightsquigarrow e'_1[e'_2]}$	PAPP
$\frac{\text{fresh } f \quad \Gamma; \Phi, f:\text{Shape}(\langle \phi \rangle) \vdash e \rightsquigarrow e' \quad \Gamma; \Phi \vdash \Lambda \{ \phi \}.e \rightsquigarrow \lambda f:\langle \phi \rangle.e'}$	CABS
$\frac{\Gamma; \Phi \vdash e \rightsquigarrow e'}{\Gamma; \Phi \vdash e\{ \phi \} \rightsquigarrow e' \text{Const}(\phi)}$	CAPP

Figure 3. Translation Rules from λ_B to $\lambda_{P+1\text{et}}$

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 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 $\text{Inst}(\Gamma, \Phi, e)$

$$\begin{aligned}
\text{Inst}(\Gamma, \Phi, e) &\doteq \text{Wrap}(e, \text{Instances}(\Gamma, \Phi)) \\
\text{Wrap}(e, \{e_1, \dots, e_n\}) &\doteq \text{let } t_1 = e_1 \text{ in } \dots \text{let } t_n = e_n \text{ in } e \\
&\quad (\text{where } t_i \text{ are fresh Bool binders}) \\
\text{Instances}(\Gamma, \Phi) &\doteq \{ f \bar{x} \mid f:\tau \leftarrow \Phi, \bar{x}:\tau \leftarrow \Gamma \\
&\quad, \Gamma, f:\tau \vdash_B f \bar{x}:\text{Bool} \}
\end{aligned}$$

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 $\text{Const}(\phi)$ that depends only on the *shape* of the bound, *i.e.*, the non-refined types of its parameters (and not the actual constraint itself).

$$\begin{aligned}
\text{Const}(r) &\doteq \text{true} \\
\text{Const}(\lambda x:b.\phi) &\doteq \lambda x:b.\text{Const}(\phi)
\end{aligned}$$

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 [27]. Let \hookrightarrow_P denote the transition relation defining the operational semantics of λ_P and \hookrightarrow_P^* 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 λ_B reduces to a constant then the translated variant of e' also reduces to the same constant:

Lemma 1 (Semantics Preservation). *If $\emptyset; \emptyset \vdash e \rightsquigarrow e'$ and $e \hookrightarrow_B^* c$ then $e' \hookrightarrow_P^* c$.*

Proof. By assumption, there exists a sequence $e \equiv e_1 \hookrightarrow_B e_2 \hookrightarrow_B \dots \hookrightarrow_B e_n \equiv c$. Let i be the largest index in which rule O-BND was applied. Then, for some ϕ and ϕ' , e_i contains a sub-expression of the form $(\Lambda\{ \phi \}.e_i^0)\{ \phi' \}$. Let e_i^1 be the expression we get if we replace $(\Lambda\{ \phi \}.e_i^0)\{ \phi' \}$ with e_i^0 in e_i . By the way we choose i , there exist a sequence $e_i^1 \hookrightarrow_P^* c$.

Let e_i^2 be the expression we get if we replace $(\Lambda\{ \phi \}.e_i^0)\{ \phi' \}$ with $(\lambda f:\langle \phi \rangle.e_i^0) (\text{Const}(\phi'))$ in e_i . Then, since f does not appear in e_i^0 , $e_i^2 \hookrightarrow_P^* c$. Finally, by the definition of $\text{Const}(\cdot)$ and the way we choose i , we have that there exists a sequence $\langle e_i \rangle \hookrightarrow \dots \hookrightarrow c$.

Since n is finite, we iteratively apply the above procedure to $e \equiv e_1 \hookrightarrow_B \dots \hookrightarrow_B \langle e_i \rangle \hookrightarrow \dots \hookrightarrow c$, until we get the sequence $\langle e \rangle \hookrightarrow_P^* c$. \square

The Soundness of λ_B follows by combining the above Semantics Preservation Lemma with the soundness of λ_P as stated in [27]:

Lemma (Soundness of λ_P). *If $\emptyset \vdash e:\sigma$ then $e \not\hookrightarrow_P^* \text{crash}$.*

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 $e \not\hookrightarrow_B^* \text{crash}$.*

3.5 Inference

Abstract Refinement Types [27] are practical mostly because they rely on the framework of Liquid Types to infer the appropriate instantiations of the abstracted refinement variables. Termination of the inference algorithm depends on the monotonicity of the intermediate solution for each π instantiation. Ultimately, this constraint means that abstract refinement variables only appear in positive positions in the right hand side r_2 of sub-typing constraints $\Gamma \vdash \{v:b \mid r_1\} \preceq \{v:b \mid r_2\}$. Our syntax does not, in general, enforce this restriction as it allows atomic refinements, and thus π variables, to appear in negative positions, which is crucial for expressiveness. Thus, we can get sub-typing constraints of the form

$$\Gamma \vdash \{v:b \mid r_1\} \preceq \{v:b \mid a_1 \Rightarrow \dots \Rightarrow a_n \Rightarrow a_q\}$$

Taking into account the semantics of subtyping, we can always do an appropriate α -renaming and get an equivalent constraint by pushing negative atomic constraints into the environment.

$$\Gamma, \{x_1:b_1 \mid a'_1\}, \dots, \{x_n:b_n \mid a'_n\} \vdash \{v:b \mid r_1\} \preceq \{v:b \mid a'_q\}$$

With this transformation we have restored monotonicity of the π variables and thus, termination of the inference algorithm.

4. Case Study: The Refined Relational Database

Next, we use Bounded Refinement Types to type relational algebra queries in relational databases. A relational database stores data in *tables*, which in turn organized are composed of *rows*, which are *records* that represents a unit of data stored in the table. For example, the table below organizes information about movies, storing a movie's title, directory, release year, and IMDB rating.

Title	Director	Year	Stars
"Birdman"	"Inárritu"	2014	8.1
"Persepolis"	"Paronnaud"	2007	8.0

The tables's *schema* describes the types of the values in each row of the table. The schema of this table is

(Title :: String, Director :: String, Year :: Int, Stars :: Double)

stating that the title and the director fields contain strings, the year is an integer and the stars is a double.

In the rest of this section we start (§ 4.1) by describing refined and type safe extensible records and use them to represent rows of databases. Then (§ 4.2), we group rows together to define database tables and we specify type safe relational operations. At this step we rely on bounds on refinement types to precisely type the relational product operator. Finally (§ 4.3), we discuss how to safely express queries in the refined database.

4.1 Rows

We represent the rows of a database with dictionaries, which are maps from some 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** $\text{Dict } \langle r \rangle \text{ k v}$ maps a key x of type k to a value of type v that satisfies the property $r \ x$

type $\text{Range } k \ v = k \rightarrow v \rightarrow \text{Prop}$

data $\text{Dict } k \ v \ \langle r :: \text{Range } k \ v \rangle$
 $= \text{D } \{dkeys :: [k]\}$
 $, dfun :: x:\{k \mid x \in \text{elts } dkeys\} \rightarrow v \langle r \ x \rangle$

Each dictionary d has a domain $dkeys$ *i.e.*, the list of ks (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 $r \ x$.

Off-the-shelf SMT solvers such as Z3 [5] include decision procedures for the theory of sets, and propositions about sets may be embedded in refinement types [28]. We define a measure keys denoting the set of keys in a dictionary:

$\text{elts} :: [a] \rightarrow \text{Set } a$
 $\text{elts } [] = \emptyset$
 $\text{elts } (x:xs) = \{x\} \cup \text{elts } xs$

$\text{keys} :: \text{Dict } k \ v \rightarrow \text{Set } k$
 $\text{keys} = \text{elts} \cdot dkeys$

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 $\text{RD } k \ v \ \langle \text{dom} :: \text{Domain } k \ v, \text{rng} :: \text{Range } k \ v \rangle$
 $= \{v:\text{Dict } \langle \text{rng} \rangle \text{ k } v \mid \text{dom } v\}$

type $\text{Domain } k \ v = \text{Dict } k \ v \rightarrow \text{Prop}$

We may instantiate dom and rng with predicates that precisely describe the values contained with the dictionary. For example, instantiating dom with the predicate $\lambda d \rightarrow \text{keys } d == \{\text{"x"}\}$ and rng with the predicate $\lambda k \ v \rightarrow (k == \text{"x"}) \Rightarrow (v > 0)$ describes dictionaries with a single field "x" whose value (as determined by $dfun$) is strictly greater than 0. We use the appropriate instantiations of these predicates to define database schemas.

An **empty dictionary** is a dictionary with empty domain and a function that will never be called

$\text{empty} :: \text{RD } \langle \text{emptyRD}, \text{rfalse} \rangle \text{ k } v$
 $\text{empty} = \text{D } [] \ (\lambda x \rightarrow \text{error "calling empty"})$

$\text{emptyRD } z = \text{keys } z == \emptyset$
 $\text{rfalse } k \ v = \text{false}$

empty 's domain is the empty set. Thus, empty 's range satisfies *any* predicate – that is, it satisfies false .

Extending dictionaries A dictionary d may be extended by adding a mapping for a particular key to a particular value. We denote singleton maps as dependent pairs, such that the expression $x := y$ denotes the mapping from x to y .

data $P \ k \ v \ \langle r :: \text{Range } k \ v \rangle$
 $= (:=) \{pk :: k, pv :: v \langle r \ pk \rangle\}$

Thus, $x := y$ has type $P \ \langle r \rangle \text{ k } v$ only when y satisfies $r \ x \ y$.

Next, we define the extend operator in terms of these dependent pairs. The expression $(k := v) += d$ evaluates to a dictionary d' that extends d with the mapping from k to v , *i.e.*, k is added to d' 's domain and when queried by k , d' returns v :

$(+=) :: p:P \ \langle r \rangle \text{ k } v \rightarrow x:\text{RD } \langle \text{dtrue}, r \rangle \text{ k } v$
 $\rightarrow \text{RD } \langle \text{cupRD } \{pk \ p\} \ x, r \rangle \text{ k } v$

$(k := v) += (\text{D } ks \ f)$
 $= \text{D } (k:ks) \ (\lambda i \rightarrow \text{if } i == k \text{ then } v \text{ else } f \ i)$

$\text{cupRD } x \ y \ z = \text{keys } z == \{x\} \cup \text{keys } y$
 $\text{dtrue } z = \text{true}$

The type of $(+=)$ constrains p , x and the returned value to have the same range r . The return type reflects the fact that the returned dictionary's domain is precisely equal to the domain of x extended by the singleton $(pk \ p)$, *i.e.*, the new key.

Database Schemata Now we are ready to model a row in a database table as a refined dictionary. We start by defining the unrefined (Haskell) type Schema , which is a dictionary mapping Strings , *i.e.*, the names of the fields of the row, to elements of some universe U containing integers, strings, and doubles.

```
data U = I Int | S String | D Double
type Schema = RD String U
```

Drawing elements from a predefined universe is not a major restriction, as most databases support limited number of types.

The type `Schema` can be used to describe *any* schema in a database, but `Schema` can be refined in such ways as to give very precise information about *some* concrete schema.

Refining the Schema. Below we define `MovieSchema`, a refined version of `Schema` that describes the schema of our `Movie` table.

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

md z = keys z == {"year", "stars", "dir", "title"}
mr k v
  = k == "year"  => isI v && 1888 < toInt v
  && k == "stars" => isD v && 0 <= toDouble v <= 10
  && k == "dir"   => isS v && k == "title" => isS v

isI (I _) = True
isI _     = False

toInt      :: {v: U | isI v} -> Int
toInt (I n) = n
...
```

The predicate `md` describes the *domain* of the movie schema, restricting the keys to "year", "stars", "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`, "stars" 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.

Populating the Movie Schema Finally, we can create the two movie entries described in our movie Table by extending the empty dictionary with the appropriate dependent pairs of fields and values

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

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

Typing `movie1` (and `movie2`) as `MovieSchema` boils down to proving: That `keys movie1 == {"year", "stars", "dir", "title"}`; and that each key is mapped to an appropriate value as determined by `mr`. For example, replacing the field "dir" with "Director" or declaring `movies year` to be `I 0` will turn cause the movie entry to become ill-typed. However, note that key ordering does *not* impact the typechecker: in `movie1` the stars field is added before the director, while `movie2` follows the reverse order. Since the (sub)typing relation depends on logical implication, ordering is not a concern.

4.2 Relational Algebra

Database tables are collections of rows, *i.e.*, collections of refined dictionaries. We define the type alias `Refined Table RT s` for the list of refined dictionaries from the field type `String` to the Universe. To save space we pack both the domain and the range refinements in the record `s` that alone describes the schema refinement, thus each row dictionary has domain `s.dom` and range `s.rng`.

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

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

The Movies Table is simply a Refined Table with domain `md` and range `mr` as previously (§ 4.1) defined.

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

The list with elements `movie1` and `movie2` is a `MoviesTable`:

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

Relational Algebra Operators. Having expressed the type of Refined Tables we want to specify properties of table transformers, *i.e.*, relational algebra operators. Below we describe the types of the basic relational operators. The safe implementation of all these operators in Haskell can be found in our benchmarks [24].

```
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
```

The relational union (*resp.* diff) is the set union (*resp.* diff) of the two tables. The type of union and diff is simple: both operators unify the schema `s` of their two arguments and their result. `select p t` filters the table `t` to those elements that satisfy the predicate `p`. The type of `select` ensures that `p` will not reference columns (fields) that are not mapped in `t`, since it is constrained to the same Refined Dictionary type. `project ks t` 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 some 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`.

Finally, `product` is the cross product of the two tables. It takes two Refined Tables with schemata `s1` and `s2` and returns a Refined Table with schema `s`. The three schemata are bounded by four bounds: (1) `Disjoint s1 s2`: the domains of `s1` and `s2` should be disjoint, (2) `Union s1 s2 s`: the domain of `s` is the union of the domains of `s1` and `s2`, (3) `Range s1 s`: the range of `s1` should imply the result range `s` for all elements of the domain of `s1`, and symmetrically, (4) `Range s2 s`: the range of `s2` should imply the result range of `s` for all elements of the domain of `s2`.

```
bound Disjoint s1 s2
  = λx y v -> s1.dom x => s2.dom y
  => v = keys x ∩ keys y => v = ∅

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
```

We note that none of these restrictions can be expressed using unbounded Abstract Refinement Types. Thus, with Bounded Refinement Types we type relational algebra operations in such a way as

to catch errors regarding domain disjointness on product, projection on invalid fields, union or difference on incompatible schemata *etc.*

4.3 Data Base Query

Consider the relational query that returns the “good movies”

```
good_movies = project ["title"] $ select (
  λd → toDouble (dfun d $ "stars") > 8
) movies
```

The above query selects the movies that have more than 8 stars and projects only their “title” field. More sophisticated queries can be found in our benchmarks [24]. Still, we note that `good_movies`, like every query in our algebra only uses haskell values: no type level computation or lifting is required. All schema reasoning happens at the granularity of the logical refinements in types.

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 (§ 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 (cf. § ??)

```
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 $\text{World} \rightarrow (a, \text{World})$, 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 [8] (also used in [16]).

Our Post-conditions are Two-State Predicates that relate the input- and output- world (as in [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) $p :: \text{Pre}$, which is a predicate over the *input* world, *i.e.*, the input world w satisfies the refinement $p \ w$; and (2) $q :: \text{Post}$, 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.

The return operator yields a pair of the supplied value x 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 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''
```

These 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 [?].

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

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 `if` statements. 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 can specify that `div` should not be called with a zero divisor, and LIQUID-HASKELL can verify that it 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 $\lambda_ b w \rightarrow b \Leftrightarrow \text{select } w \text{ ctr} \neq 0$ and the pre-condition of `divX`'s is instantiated to $\lambda w \rightarrow \text{select } w \text{ ctr} \neq 0$, which suffices to prove that `div` is only called with non-zero values.

The `whileM` combinator formalizes “while-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 trivial 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'
```

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

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

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

Figure 4. Privilege Specification

```
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)))

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
```

Figure 5. Specification and Source of `copyRec`

Third, to avoid having to transitively connect guard and the body, we require that the loop post-condition be a one-state predicate, independent of the input world (as in classical 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 at the end, the counter is exactly equal to 0.

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

6. Safe Scripting via RIO

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

Shill is a scripting language that restricts the privileges with which a script may execute by using *capabilities* and *contracts* [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` runtime guarantees that system resources are accessed in the manner described by its contract.

In the remainder of 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 [15], how can we use Bounded Refinement Types to encode scripting privileges and reason about them *statically*?

We encode Shill contracts as RIO types thereby providing *compile-time* guarantees about privilege and resource usage. To achieve this, we: connect the state (World) of the RIO monad to an *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 4 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 Worlds with Privileges we assume a set of uninterpreted functions of type $\text{Priv} \rightarrow \text{Bool}$ that act as predicates on values of type Priv, each denoting a particular privilege. For example, given a value $p :: \text{Priv}$, the proposition $\text{pread } p$ 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 FHs. Given $h:\text{FH}$ and $w:\text{World}$, $\text{pwrite } (\text{select } (\text{caps } w)x)$ denotes that in the state w , the file x may be written. This pattern is generalized by the predicate $\text{pset } \text{pwrite } h$ 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 $\text{eqP } w \ w'$ says that the set of privileges and active handles in worlds w and w' are *equivalent*.

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

We can now specify the preserving operations that read, write files and list the contents of a gdirectory, 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 $\text{pcreateFP } (\text{select } (\text{caps } w)h)$ (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 ($\text{pwrite } (\text{pcreateFP } (\text{select } (\text{caps } w)h))$) and x is derived from h ($\text{derivP } w' \ w \ x$ h both hold), then we know that x is writable in the new World w' ($\text{pwrite } (\text{select } (\text{caps } w')x)$).

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
```

RJ:♣.HEREHEREHERE♣

6.3 Client Script Verification

We now turn to the program copyRec in Figure 5 that copies the contents of the directory f to the directory d . copyRec executes by first listing the contents of f . copyRec opens each child path, p in f . If the result is a file, then it is copied to the directory d . Otherwise, copyRec recurses on the child if recur is **True**.

A good type for copyRec would specify the minimal set of privileges that copyRec requires. 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), and (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.

We define $\text{copySpec } s \ d \ w$ that encode these privileges. This specification is sufficient to verify the various API calls in copyRec , as the types for >>= and return track updates to the current state. However, merely specify that $\text{copySpec } s \ d \ w$ hold as a postcondition of copyRec is insufficient: the output world w is only known to satisfy $\text{copySpec } x \ y \ w$ (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. 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 note that if a 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 . We thus parameterize copyRec by a predicate i , bounded by $\text{stable } i$, which precisely describes the possible world transformations under which i should be stable. 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 type `copyRec`:

```
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, at the recursive call, `i` is instantiated with `λw → copySpec s d w`, allowing `copyRec` to typecheck.

7. Related Work

Higher order Logics and Dependent Type Systems including NuPRL [4], Coq [2], Agda [18], and even Haskell [14, 20], 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 [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 [3, 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 HaskellBB) statically reject accesses of “missing” fields. The HList implementation is non-trivial, requiring new type-classes 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 [7]. 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 [8], which was later enriched with separation logic in Ynot [16]. 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. Second, consequently, we automate inference of pre- and post-conditions and loop invariants by using the machinery of refinement instantiation via the abstract interpretation of Liquid Types.

References

- [1] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. Refinement types for secure implementations. *ACM TOPLAS*, 2011.
- [2] Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
- [3] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin. Student paper: HaskellDB improved. In *Haskell*, 2004.
- [4] R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [5] L. de Moura and N. Björner. Z3: An efficient SMT solver. 2008.
- [6] J. Dunfield. Refined typechecking with Stardust. In *PLPV*, 2007.
- [7] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell*, 2012.
- [8] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *TYPES*, 1998.
- [9] C. Fournet, M. Kohlweiss, and P-Y. Strub. Modular code-based cryptographic verification. In *CCS*, 2011.
- [10] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE S & P*, 2011.
- [11] Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. In *ICFP*, 2014.
- [12] Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Strongly typed heterogeneous collections. In *Haskell*, 2004.
- [13] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, 1999.
- [14] Conor McBride. Faking it: Simulating dependent types in Haskell. In *JFP*, 2002.
- [15] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A secure shell scripting language. In *OSDI*, 2014.
- [16] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.
- [17] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1979.
- [18] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, 2007.
- [19] Nicolas Oury and Wouter Swierstra. The power of Pi. In *ICFP*, 2008.
- [20] S. L. Peyton-Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, 2006.

- [21] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
- [22] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. *IEEE TSE*, 1998.
- [23] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, 2013.
- [24] UCSD Programming Systems. github.com/ucsd-progsys/liquidhaskell/tree/master/benchmarks/icfp15.
- [25] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, 2010.
- [26] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, 2013.
- [27] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.
- [28] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell*, 2014.
- [29] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In *ICFP*, 2014.
- [30] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.