How to Safely Use Extensionality in Liquid Haskell

Niki Vazou
niki.vazou@imdea.org
IMDEA Software Institute
Madrid, Spain

Michael Greenberg
michael.greenberg@pomona.edu
Stevens Institute of Technology
Hoboken, NJ, USA

Abstract

Refinement type checkers are a powerful way to reason about functional programs. For example, one can prove properties of a slow, specification implementation and port the proofs to an optimized pure implementation that behaves the same. But to reason about higher-order programs, we must reason about equalities between functions: we need a consistent encoding of functional extensionality.

A natural but naive phrasing of the functional extensionality axiom (`funext`) is inconsistent in refinement type systems with semantic subtyping and polymorphism: if we assume `funext`, then we can prove false. We demonstrate the inconsistency and develop a new approach to equality in Liquid Haskell: we define a propositional equality in a library we call `PEq`. Using `PEq` avoids the inconsistency while proving useful equalities at higher types; we demonstrate its use in several case studies. We validate `PEq` by building a model and developing its metatheory. Additionally, we prove metaproperties of `PEq` inside Liquid Haskell itself using an unnamed folklore technique, which we dub 'classy induction'.

CCS Concepts: **• Software and its engineering → Software verification**.

Keywords: refinement types, function equality, functional extensionality

ACM Reference Format:
Niki Vazou and Michael Greenberg. 2022. How to Safely Use Extensionality in Liquid Haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3546189.3549919

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '22, September 15–16, 2022, Ljubljana, Slovenia
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9438-3/22/09.
https://doi.org/10.1145/3546189.3549919

1 Introduction

Refinement types have been extensively used to reason about pure functional programs [12, 30, 31, 37, 47]. Equipped with SMT reasoning, and thus SMT equality, proving equalities between fully applied (first-order) expressions is straightforward for such systems. For example, Vazou et al. [41] prove function optimizations correct by equating the results of slow and fast program versions. Do these equalities hold in the context of higher order functions (e.g., maps and folds) or do the proofs need to be redone for each fully applied context? Without functional extensionality (a.k.a. `funext`), one must duplicate proofs for each higher-order function. Worse still, all reasoning about higher-order representations of data requires first-order observations.

Most verification systems allow for function equality by way of functional extensionality, either built-in (e.g., Lean) or as an axiom (e.g., Agda, Coq). Liquid Haskell and F*, two major, SMT-based verification systems using refinement types, are no exception: function equalities come up regularly. But, in both these systems, first attempts at axioms for functional extensionality were wrong[1]. A naive `funext` axiom proves equalities between unequal functions.

We begin by describing why a naive encoding of `funext` is inconsistent (§2). At first sight, function equality can be encoded as a refinement type stating that for functions `f` and `g`, if we can prove that `f x` equals `g x` for all `x`, then the functions `f` and `g` are equal:

```
funext :: ∀ a b. f:(a → b) → g:(a → b)
       → (x:a → {f x = g x}) → {f = g}
```

(The *refinement proposition* `{e}` is equivalent to `{_:() | e}`.) On closer inspection, `funext` encodes function equality improperly: its resulting domain equates `f` and `g` without reference to the domain `a`, seemingly applying on *any* domain. What if we instantiate the domain type parameter `a`'s refinement to an intersection of the domains of the input functions or, worse, to an uninhabited type? Would such an instantiation of `funext` still prove equality of the two input functions? It turns out that this naive extensionality axiom is inconsistent with refinement types: in §2 we assume this naive `funext` and prove `false`—disaster! We work in Liquid Haskell, but the problem generalizes to any refinement type

[1] See https://github.com/FStarLang/FStar/issues/1542 for F*'s initial, wrong encoding and §6 for F*'s different solution. We explain the situation in Liquid Haskell in §2.

system that allows for semantic subtyping along with polymorphism. F* used to have the same naive encoding until 2018 that F*'s developers reported the inconsistency in a (impenetrable to non-refinement type experts) github issue[1]. To be sound, proofs of function equality must carry information about the domain type on which the compared functions are equal.

Our second contribution is to define a type-indexed propositional equality as a Liquid Haskell library (§3), where the type indexing uses a refined Haskell data type generated by three axioms. We call the propositional equality `PEq` and find that it adequately reasons about function equality: we can prove the theorems we want and we can't prove the (non-) theorems we *don't* want. Further, we prove in Liquid Haskell *itself* that the implementation of `PEq` is an equivalence relation, i.e., it is reflexive, symmetric, and transitive. To conduct these proofs—which go by structural induction on the type index—we apply a heretofore-unnamed folklore proof methodology, which we dub *classy induction* (§3.3).

Our third contribution is to use `PEq` to prove a variety of equalities heretofore inaccessible in Liquid Haskell (§4), from equalities between functions up to monads. As simple examples, we prove optimizations correct as equalities between functions (i.e., `reverse`), work carefully with functions that only agree on certain domains and dependent codomains, lift equalities to higher-order contexts (i.e., `map`), prove equivalences with multi-argument higher-order functions (i.e., `fold`), showcase how higher-order, propositional equalities can co-exist with and speedup executable code, and prove monad laws for reader monads.

Our fourth and final contribution is to formalize λ^{RE}, a core calculus modeling `PEq`'s two important features: type-indexed, functionally extensional propositional equality and refinement types with semantic subtyping. We prove that λ^{RE} is sound and that propositional equality implies equality in a term model of equivalence (§5).

An extended (with proofs) version of this paper can be found in [42] and our implementation can be found at gihub in github.com/nikivazou/propositional-equality.

2 Functional Extensionality in Refinement Types

We start (§ 2.1) by concrete examples that showcase the inconsistencies of a naive encoding for function extensionality in refinement types and then (§ 2.2) introduce an alternative type-indexed encoding.

2.1 Inconsistency of the Naive Encoding

Functional extensionality states that two functions are equal, if their values are equal at every argument:

$$\forall f, g : A \to B, \forall x \in A, f(x) = g(x) \Rightarrow f = g \tag{1}$$

Most theorem provers consistently admit functional extensionality as an axiom, which we call `funext` throughout. Admitting `funext` is a convenient way to generate equalities on functions and reuse higher order proofs. But correctly encoding `funext` in a refinement typed language is not trivial. For example, in Liquid Haskell we naively admitted the `funext` axiom below (we highlight Liquid Haskell type signatures that in the implementation are written as comments):

```
assume funext :: ∀ a b. f:(a → b) → g:(a → b)
              → (x:a → {f x = g x}) → {f = g}
funext _f _g _pf = ()
```

The **assume** keyword introduces an axiom: Liquid Haskell will accept the refinement signature of `funext` wholesale and ignore its definition. Also, note that the = symbol in the refinements refers to SMT equality (see §3.4). Our encoding certainly *looks* like the mathematical encoding (1). But looks can be deceiving: in Liquid Haskell, we can use `funext` to prove `false`. Why?

Consider two functions on `Int`s: the `incrInt` function increases all integers by one; the `incrPos` function increases positive numbers by one, returning `0` otherwise:

```
incrInt, incrPos :: Int → Int
incrInt n = n + 1
incrPos n = if 0 < n then n + 1 else 0
```

Liquid Haskell easily proves that these two functions behave the same on positive numbers:

```
type Pos = {n:Int | 0 < n}
incrSamePos :: n:Pos → {incrPos n = incrInt n}
incrSamePos _n = ()
```

Using our proof `incrSamePos` on the domain of positive numbers, our `funext` axiom proves `incrPos` and `incrInt` equal.

```
incrExt :: {incrPos = incrInt}
incrExt = funext incrPos incrInt incrSamePos
```

Having `incrExt` to hand, it's easy to prove that every higher-order use of `incrPos` can be replaced with `incrInt`, which is much more efficient—it saves us a conditional branch! For example, `incrMap` shows that mapping over a list with `incrPos` is just the same as mapping over it with `incrInt`.

```
incrMap :: xs:[Pos]
        → {map incrPos xs = map incrInt xs}
incrMap xs = incrExt
```

We could prove `incrMap` without function equality, i.e., if we only knew `incrSamePos`. To do so, we would write an inductive proof—and we'd have to redo the proof for every context in which we would rewrite `incrPos` to `incrInt`. So `funext` is in part about *modularity* and *reuse* in theorem proving. But `funext` is critical to equate structures that are themselves higher order—like the reader monad (§ 4.6).

Unfortunately, `incrExt` proves too many equivalences… our system is inconsistent! We prove that 0 equals -4:

```
inconsistencyI :: {incrPos (-5) = incrInt (-5)}
inconsistencyI = incrExt -- proof of 0 = −4
```

What happened here? How can we have that equality… that `0 = -4`? Liquid Haskell looked at `incrExt` and saw the two functions were equal… without any regard to the domain on which `incrExt` proved `incrPos` and `incrInt` equal! We *forgot* the domain, and so `incrExt` generates a proof in SMT that those two functions are equal on *any* domain.

In short, `funext` is *inconsistent* in Liquid Haskell! Liquid Haskell forgets the domain on which the two functions are proved equal, remembering only the equality itself.

We can exploit `funext` to find equalities between *any* two functions that share the same Haskell type on the *empty* domain and Liquid Haskell will treat these functions as *universally* equal. For example, the `plus2` function adds `2` to its input; it does not equal `incrInt` on any nontrivial domain.

```
plus2 :: Int → Int
plus2 x = x + 2
```

But `plus2` *does* behave the same as `incrInt` on the empty domain, i.e., for all inputs `n` that satisfy `false`.

```
type Emp = {v:Int | false}
incrSameEmp :: n:Emp → {incrInt n = plus2 n}
incrSameEmp _n = ()
```

Now `incrSameEmp` provides enough evidence for `funext` to show that `incrInt` equals `plus2`—discarding the empty domain and proving another egregious inconsistency.

```
incrPlus2Ext :: {incrInt = plus2}
incrPlus2Ext = funext incrInt plus2 incrSameEmp

inconsistencyII :: {incrInt 0 = plus2 0}
inconsistencyII = incrPlus2Ext -- proof of 1 = 2
```

Refinement types, unlike type theories that are consistent with function equality, permit implicit subtyping. When calling `incrPlus2Ext`, the domains of the functions are implicitly strengthened to the empty domain of the `incrSameEmp` proof. Implicit subtyping is an extremely convenient feature when programming with refinement types that let us, as a trivial example, divide with a value of type `{v:Int | v > 0}`, without any proof that $v > 0 \Rightarrow v \neq 0$. But, when it comes to function equality, implicit subtyping renders a huge burden. Even though function equality has been extensively studied under various type theories (§ 6), none of these support implicit subtyping. But, in Liquid Haskell we still need to prove equalities between higher-order values! What can we do?

2.2 Refined, Type-Indexed, Extensional, Propositional Equality

When proving equalities, we must be careful about the domains of the functions involved. Proving f and g extensionally equal, we must reason about *four* domains. Let $\mathcal{D}_f$ and $\mathcal{D}_g$ be the domains on which the functions f and g are respectively defined. Let $\mathcal{D}_p$ be the domain on which the two functions are proved equal and $\mathcal{D}_e$ the domain on which the resulting equality between the two functions is found. In our `incrExt` example above: the function domains are `Int` ($\mathcal{D}_f$ = $\mathcal{D}_g$ = `Int`), as specified by the function definitions; the domain of the proof is positive numbers ($\mathcal{D}_p$ = `Pos`), as specified by `incrSamePos`; and, disastrously, the domain of the equality itself is unspecified in `funext`. Liquid Haskell chooses the most general domain possible ($\mathcal{D}_e$ = `Int`).

The `funext` axiom imposes no constraints between these domains, merely requiring that $\mathcal{D}_f$, $\mathcal{D}_g$, and $\mathcal{D}_p$ be supertypes of the empty domain, which trivially holds for all types, leaving $\mathcal{D}_e$ underconstrained.

To be consistent, it suffices for our functional extensionality axiom to (1) capture the domain of function equality $\mathcal{D}_e$ explicitly, (2) require that the domain of the equality, $\mathcal{D}_e$, is a subtype of the domain of the proof, $\mathcal{D}_p$, itself a subtype of the functions' domains, $\mathcal{D}_f$ and $\mathcal{D}_g$, and (3) restrict any use of the resulting equality between functions to subdomains of $\mathcal{D}_e$.

Our solution is to define a refined, type-indexed, extensional propositional equality. We do so in the Liquid Haskell library `PEq`, which defines a propositional equality also called `PEq`. We write `PEq a {e`$_l$`} {e`$_r$`}` to mean that the expressions `e`$_l$ and `e`$_r$ are propositionally equal at type `a`. We carefully axiomatize `PEq` (§3) to meet our three criteria.

1. PEq is Type-Indexed. The type index `a` in `PEq a {e`$_l$`} {e`$_r$`}` makes it easy to track types explicitly. We encode functional extensionality as the `xEq` axiom that keeps careful track of types:

```
assume xEq :: f:(a → b) → g:(a → b)
          → (x:a → PEq b {f x} {g x})
          → PEq (a → b) {f} {g}
```

The result type of `xEq` explicitly captures the equality domain as the domain of the function type in the returned equality (i.e., `a`). The standard variance and type checking rules of Liquid Haskell ensure that the domains $\mathcal{D}_f$, $\mathcal{D}_g$, and $\mathcal{D}_p$ are supertypes of $\mathcal{D}_e$.

2. Generating Function Equalities. The axiom `xEq` generates equalities at function types using functional extensionality. Liquid Haskell checks the domains, never proving equality between functions at an inappropriate domain.

Returning to `incrPos` and `incrInt`, we can use `xEq` to find these functions equal on the domain `Pos`, highlighting the Liquid Haskell signature and leaving the Haskell one plain:

```
incrExtGood :: PEq (Pos → Int) {incrPos} {incrInt}
incrExtGood :: PEq (Int → Int)
incrExtGood = xEq incrPos incrInt incrEq
```

`xEq` checks that the `incrPos` and `incrInt`'s domains are supertypes of `Pos`, i.e., `Pos <: Int`. Further it checks that the domain of the proof `incrEq` is a supertype of `Pos`.

What might we define for `incrEq`? Here are three alternatives. Each alternative is either accepted or rejected by `xEq` as appropriate for the `Pos → Int` type index; each alternative is also possible or impossible to prove. (See §3 for more on how `incrEq` can be defined.)

```
-- ACCEPTED and POSSIBLE:
incrEq :: n:Pos → PEq Int {incrPos n} {incrInt n}
-- ACCEPTED and IMPOSSIBLE:
incrEq :: n:Int → PEq Int {incrPos n} {incrInt n}
-- REJECTED and POSSIBLE:
incrEq :: n:Emp → PEq Int {incrPos n} {incrInt n}
```

The first two alternatives, `n:Pos` and `n:Int`, will be accepted by `xEq`, since both `Pos` and `Int` are supertypes of `Pos`... though it is impossible to actually construct a proof for the second alternative, i.e., a proof that `incrPos n` equals `incrInt n` for all integers `n`. On the other hand, the last proof on `n:Emp` is trivial, but `xEq` rejects it, because `Emp` is not a supertype of `Pos`. Liquid Haskell's checks on `xEq`'s type indices prevents inconsistencies like `inconsistencyII`.

3. Using Function Equalities. Just as the `xEq` axiom ensures that the right domains are checked and tracked for functional extensionality, we define an axiom to ensure these equalities are used appropriately. The axiom `sEq` characterizes equality as valid under substitution, i.e., if `x` and `y` are equal, they can be substituted in any context `f` and the results `f x` and `f y` will be equal:

```
assume sEq :: f:(a → b) → x:a → y:a
               → PEq a {x} {y} → PEq b {f x} {f y}
```

The `sEq` axiom applies equalities in higher-order contexts. For example, we show `map incrPos` equals `map incrInt`:

```
incrMapProp :: PEq ([Pos] → [Int]) {map incrPos}
                                  {map incrInt}
incrMapProp = sEq map incrPos incrInt incrExtGood
```

We can more generally show that propositionally equal functions produce equal results on equal inputs. The trick is to *flip* the context, defining a function `retract` that takes as input two functions `f` and `g`, a proof these functions are equal, and an argument `x`, returning a proof that `f x = g x`:

```
retract :: f:(a → b) → g:(a → b)
        → PEq (a → b) {f} {g}
        → x:a → PEq b {f x} {g x}
retract f g peq x = sEq (flip x) f g peq

flip x f = f x
```

The `retract` lemma makes it easy to use function equalities while still checking the domain on which the function is applied. These checks prevent inconsistencies like `inconsistencyI`. For instance, we can try to retract the functional equality `incrExtGood` to a bad and a good input.

```
-- REJECTED:
badFO :: PEq Int {incrPos 0} {incrInt 0}
badFO = retract incrPos incrInt incrExtGood 0

-- ACCEPTED
goodFO :: x:{Int | 42 < x }
       → PEq Int {incrPos x} {incrInt x}
goodFO x = retract incrPos incrInt incrExtGood x
```

Liquid Haskell rejects the bad input in `badFO`: the number 0 isn't in the `Pos` domain on which `incrExtGood` was proved. Liquid Haskell accepts the good input in `goodFO`, since any `x` greater than 42 is certainly positive. The `goodFO` proof yields a first-order equality on any such `x`, here on `Int`. Such first order equalities correspond neatly with the notion of equality used in the SMT solvers that buttress all of Liquid Haskell's reasoning. (For more information on how SMT equality relates to notions of equality in Liquid Haskell, see §3. For an example of how these first-order equalities can lead to runtime optimizations, see §4.5.)

3 PEq: a Type Indexed Equality Axiomatized with Extensional Equality

We define the `PEq` library in Liquid Haskell, implementing the type-indexed propositional equality, also called PEq. First, we axiomatize equality for base types in the `AEq` typeclass (§3.1). Next, we define propositional equality for base and function types with the `PEq` data type (§3.2). Axioms on `PEq` enforce the typing rules of our formal model (§5), but we also prove some of the metatheory in Liquid Haskell itself (§3.3). Finally, we discuss how `AEq` and `PEq` interact with Haskell's and SMT's equalities (§3.4).

3.1 The AEq Typeclass, for Axiomatized Equality

We begin by axiomatizing equality that can be ported to SMT: equivalence relations that imply SMT equality. We use refinements on typeclasses [22] to define a typeclass `AEq`, which contains the (operational, `Bool`-returning, SMT-coinciding) equality method ≡, three methods that encode the equality laws, and one method that encodes correspondence with SMT equality.

```
class AEq a where
  (≡)    :: x:a → y:a → Bool
  reflP  :: x:a → {x ≡ x}
```

```
-- (1) Plain Haskell Definitions
data PBEq a = PBEq
bEq :: AEq a ⇒ a → a → () → PBEq a
bEq = (const . const . const) PBEq
xEq :: (a → b) → (a → b) → (a → PEq b)
    → PBEq (a → b)
xEq = (const . const . const) PBEq
sEq :: (a → b) → a → a → PBEq a → PBEq b
sEq = (const . const . const . const) PBEq

-- (2) Uninterpreted equality between e1 and e2
type PEq a e1 e2 = {v:PBEq a | e1 ≍ e2}
measure (≍) :: a → a → Bool

-- (3) Axiomatization of PEq
assume bEq :: AEq a ⇒ x:a → y:a → {v:() | x ≡ y}
            → PEq a {x} {y}
assume xEq :: f:(a → b) → g:(a → b)
            → (x:a → PEq b {f x} {g x})
            → PEq (a → b) {f} {g}
assume sEq :: f:(a → b) → x:a → y:a
            → PEq a {x} {y} → PEq b {f x} {f y}
```

Figure 1. Implementation of the propositional equality PEq.

```
  symmP  :: x:a → y:a → { x ≡ y ⇒ y ≡ x }
  transP :: x:a → y:a → z:a
            → { (x ≡ y && y ≡ z) ⇒ x ≡ z }
  smtP   :: x:a → y:a → { x ≡ y } → { x = y }
```

An instance of AEq defines the method (≡) and provides explicit proofs that it is an equivalence relation (reflP, symmP, and transP resp.). The instance must also show (smtP) that (≡) implies SMT equality, namely, structural equality[2].

3.2 The PBEq Data Type and its PEq Axiomatization

We use AEq to define our type-indexed propositional equality `PEq a {e1} {e2}` in three steps (Figure 1): (1) structure as a Haskell definitions, (2) definition of the refined type PEq, and (3) axiomatization of equality using refinement types.

First, we define the structure of our proofs of equality as PBEq, an unrefined single constructor data type (Figure 1, (1)). The plain data type defines the structure of derivations in our propositional equality (i.e., which proofs are well formed), but none of the constraints on derivations (i.e., which proofs are valid). There are three functions that generate a propositional equality: using an AEq instance (bEq); using funext (xEq); and using substitutivity (sEq). All these functions are defined using `const _ x = x` to simply ignore their arguments and return the unique PBEq constructor.

Next, we define the refinement type PEq to be our propositional equality (Figure 1, (2)). Two terms e1 and e2 of type a are propositionally equal when (a) there is a well formed and valid PBEq proof and (b) we have `e1 ≍ e2`, where (≍) is an uninterpreted SMT function, i.e., a purely symbolic function about which SMT knows nothing. Liquid Haskell uses curly braces for expression arguments in type applications, e.g., in `PEq a {x} {y}`, x and y are expressions, but a is a type.

Finally, we use assumptions to axiomatize the uninterpreted (≍) and generate proofs of PEq (Figure 1, (3)). Each function from (1) is refined to return something of type PEq, where `PEq a {e1} {e2}` means that terms e1 and e2 are considered equal at type a. bEq constructs proofs that two terms, x and y of type a, are equal when $x \equiv y$ according to the AEq instance for a. xEq is the (type-indexed) funext axiom. Given functions f and g of type `a → b`, a proof of equality via extensionality also needs a PEq-proof that `f x` and `g x` are equal for all x of type a. Such a proof has refined type `x:a → PEq b {f x} {g x}`. Critically, we don't lose any type information about f or g! sEq implements substitutivity closure: for an arbitrary context with an a-shaped hole (`f :: a → b`) and for any x and y of type a that are equal—i.e., `PEq a {x} {y}`—filling the context with x and y yields equal results, i.e., `PEq b {f x} {g y}`.

Example. AEq and bEq suffice to prove incrEq from §2:

```
incrEq :: x:Pos → PEq Int {incrPos x} {incrInt x}
incrEq x = bEq (incrPos x) (incrInt x)
               (reflP (incrPos x))
```

We start from `reflP (incrPos x) :: {incrPos x ≡ incrPos x}`, since x is positive, the SMT derives `incrPos x = incrInt x`, generating the AEq proof term `{incrPos x ≡ incrInt x}`, which, in turn, is passed to the bEq axiom.

3.3 Equivalence Properties and Classy Induction

We prove the metaproperties of the actual implementation of PEq—reflexivity, symmetry, and transitivity—within Liquid Haskell itself, by induction on types. But "induction" in Liquid Haskell means writing a recursive function, which necessarily has a single, fixed type. To express that PEq is reflexive, we want a Liquid Haskell theorem `refl :: x:a → PEq a {x} {x}`, but its proof goes by induction on the type a, which is not possible in ordinary Haskell functions[3].

The essence of our proofs is a folklore method we call *classy induction* (see §6 for the history). To prove a theorem using classy induction on PEq, one must: (1) define a typeclass with a method whose refined type corresponds

[2]The three axioms of equality alone are not enough to ensure SMT's structural equality, e.g., one can define an instance `x ≡ y = True` which satisfies the equality laws, but does not correspond to SMT equality.

[3]A variety of GHC extensions allow case analysis on types (e.g., type families and generics), but, unfortunately, Liquid Haskell doesn't support such fancy type-level programming.

to the theorem; (2) prove the base case for types with `AEq` instances; and (3) prove the inductive case for function types, where typeclass constraints on smaller types generate inductive hypotheses. All three of our proofs follow this pattern. For example, for reflexivity (shown below): (1) the typeclass `Reflexivity` simply states the desired theorem type `refl :: x:a → PEq a {x} {x}`; (2) given an `AEq a` instance, `bEq` and the `reflP` method are combined to define the `refl` method; and (3) `xEq` can show that `f` is equal to itself by using the `refl` instance from the codomain constraint: the `Reflexivity b` constraint generates a method `refl :: x:b → PEq b {x} {x}`. The codomain constraint `Reflexivity b` corresponds exactly to the inductive hypothesis on the codomain: we are doing induction!

```
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE UndecidableInstances #-}
-- (1) Refined typeclass
class Reflexivity a where
  refl :: x:a → PEq a {x} {x}

-- (2) Base case (AEq types)
instance AEq a ⇒ Reflexivity a where
  refl a = bEq a a (reflP a)

-- (3) Inductive case (function types)
instance Reflexivity b ⇒ Reflexivity (a→b) where
  refl f = xEq f f (\a → refl (f a))
```

At compile time, any use of `refl x` when `x` has type `a` asks the compiler to find a `Reflexivity` instance for `a`. If `a` has an `AEq` instance, the proof of `refl x` will simply be `bEq x x (reflP a)`. If `a` is a function of type `b → c`, then the compiler will try to find a `Reflexivity` instance for the codomain `c`—and if it finds one, generate a proof using `xEq` and `c`'s proof. The compiler's constraint resolver does the constructive proof for us, assembling the 'inductive tower' to give us a `refl` for our chosen type. That is, even though Liquid Haskell can't mechanically check that our inductive proofs are in general complete (i.e., the base and inductive cases cover all types), our `refl` proofs will work for types where the codomain bottoms out with an `AEq` instance, i.e., any type consisting of functions and `AEq`-equable types.

Our proofs of symmetry and transitivity follow the same pattern and can be found in our implementation.

3.4 Interaction of the Different Equalities

We have four equalities in our system (Figure 2): SMT equality (=), the (≡) method of the `AEq` typeclass (§3.1), the refined `PEq` (§3.2), and the (==) method of Haskell's `Eq` typeclass.

SMT Equality. The single equal sign (=) represents SMT equality, which satisfies the three equality axioms and is syntactically defined for data types. The SMT-LIB standard [6] permits the equality symbol on functions but does not specify

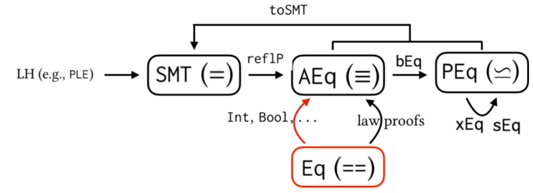

Figure 2. The four equalities and their interactions. Haskell equality in red to highlight its potential unsoundness.

its behavior. Implementations vary. CVC4 allows for functional extensionality and higher-order reasoning [5]. When Z3 compares functions for equality, it treats them as arrays, using the extensional array theory to incompletely perform the comparison. When asked if two functions are equal, Z3 typically answers `unknown`. To avoid this unpredictability, our system avoids SMT equality on functions.

Interactions of Equalities. SMT equalities are internally generated by Liquid Haskell using the reflection and PLE tactic of Vazou et al. [43] (see also §4.1). An $e_1 \equiv e_2$ equality can be generated in three ways: (1) If SMT can prove an equality $e_1 = e_2$, then the reflexivity `reflP` method can generate that equality, i.e., `reflP` e_1 proves $e_1 \equiv e_1$, which is enough to show $e_1 \equiv e_2$. (2) Our system provides `AEq` instances for the primitive Haskell types using the Haskell equality that we *assume* satisfies the four laws, e.g., the `instance AEq Int` is provided. (3) Using refinements in typeclasses [22] one can explicitly define instances of `AEq`, which may or may not coincide with Haskell `Eq` instances. Axioms generate `PEq` proofs, bottoming out at `AEq`, as in `bEq` combined with an `AEq` term and `xEq` or `sEq` combined with other `PEq` terms.

Finally, we define a mechanism to convert `PEq` into an SMT equality. This conversion is useful when we want to derive an equality $f\ e = g\ e$ from a function equality `PEq (a → b) {f} {g}` (see §4.5). The derivation requires that the domain `b` admits the axiomatized equality, `AEq`. To capture this requirement we define `toSMT` that converts `PEq` to SMT equality as a method of a class that requires an `AEq` constraint:

```
class AEq a ⇒ SMTEq a where
  toSMT :: x:a → y:a → PEq a {x} {y} → {x = y}
```

A Separate Equality. Liquid Haskell maps Haskell's (==) to SMT equality by default. It is surely unsound to do so, as users can define `Eq` instances with arbitrarily bad (==) methods. To avoid this built-in unsoundness, our implementation and case studies don't directly use Haskell's equality.

The interactions of the four equalities justify using `AEq` instances, instead of the standard reflexivity, as the base case of `PEq`. We only want to convert Haskell equalities to `PEq` when they are safe for SMT, i.e., equalities that satisfy `AEq`.

Equivalence Relation Axioms. Each of the four equalities has a different relationship to the equivalence relation

axioms (reflexivity, symmetry, transitivity). `AEq` comes with explicit proof methods that capture the axioms. For `PEq`, we prove them using classy induction (§3.3). For SMT equality, we simply trust implementation of the underlying solver. For Haskell's equality, there's no general way to enforce the equality axioms, though users can choose to prove them.

Computability. Finally, the `Eq` and `AEq` classes define the computable equalities used in programs, `(==)` and `(≡)` respectively. The `PEq` equality only contains proof terms, while the SMT equality lives entirely inside the refinements; neither can be meaningfully used in programs.

4 Examples and Case Study

To showcase the proposed propositional equality `PEq` for Liquid Haskell, we start by moving from first-order equalities to equalities between functions (`reverse`, §4.1). Next, we show how `PEq`'s type indices reason about refined domains and dependent codomains of functions (`incr`, §4.2). Proofs about higher-order functions demonstrate the contextual equivalence axiom in single- (`map`, §4.3) and multi-argument functions (`foldl`, §4.4). Finally, we show how a `PEq` proof can be used to reason about optimization correctness (`spec`, §4.5) and we prove associativity of the reader monad (§4.6).

4.1 Reverse: From First- to Higher-Order Equality

Consider three candidate list-reversal implementations (Figure 3, top): a 'slow' quadratic one that recursively appends to the right, a 'fast' one in accumulator-passing style, and a 'bad' one that returns the original list.

First-Order Proofs. The `reverseEq` theorem neatly relates the two correct list reversals (Figure 3, middle). It is a corollary of a `lemma` and `rightId`, which shows that `[]` is a right identity for the list concatenation operator, `(++)`. The `lemma` characterizes the core induction, relating the accumulating `fastGo` and the direct `slow`. The `lemma` itself uses the inductive lemma `assoc` to show associativity of `(++)`. All the equalities in the first order statements use the SMT equality, since they are automatically proved by Liquid Haskell's reflection and PLE tactic [43].

Higher-Order Proofs. Plain SMT equality isn't enough to prove that `fast` and `slow` are themselves equal. We need the functional extensionality axiom `xEq`.

```
reverseHO :: PEq ([a] → [a]) {fast} {slow}
reverseHO = xEq fast slow reversePf
```

The job of the `reversePf` lemma is to prove `fast xs` propositionally equal to `slow xs` for all `xs`:

```
reversePf :: xs:[a] → PEq [a] {fast xs} {slow xs}
```

We can use two different styles to construct such a proof.

Style 1: Lifting First-Order Proofs. Using the `bEq` axiom and the reflexivity property of `AEq`, we can lift the first order equality proof `reverseEq` into a propositional equality:

```
reversePf1 :: AEq [a] ⇒ xs:[a]
           → PEq [a] {fast xs} {slow xs}
reversePf1 xs = bEq (fast xs) (slow xs)
                  (reverseEq xs ? reflP (fast xs))
```

Such proofs rely on SMT equality, which the `reflP` call turns into axiomatized equality (`AEq`).

Style 2: Inductive Proofs. Alternatively, inductive proofs can be directly performed in the propositional setting, eliminating the `AEq` constraint (though any use of such proofs requires `AEq a`). To give a sense of what these proofs are like, we translate `lemma` into `lemmaP`:

```
lemmaP :: (Transitivity [a], Reflexivity [a])
       ⇒ l:[a] → xs:[a]
       → PEq [a] {fastGo l xs} {slow xs ++ l}
lemmaP l []     = refl l
lemmaP l (x:xs) =
  trans (fastGo l (x:xs)) (slow xs ++ (x:l))
        (slow (x:xs) ++ l)
        (lemmaP (x:l) xs) (assocP (slow xs) [x] l)
```

The proof goes by induction and uses the `Reflexivity` and `Transitivity` properties of PEq encoded as typeclasses (§3.3), along with `assocP` and `rightIdP`, the propositional versions of `assoc` and `rightId` (not shown). These typeclass constraints propagate to the `reverseHO` proof, via `reversePf2`.

```
reversePf2 :: (Transitivity [a]) ⇒ xs:[a]
           → PEq [a] {fast xs} {slow xs}
reversePf2 xs =
  trans (fast xs) (slow xs ++ []) (slow xs)
        (lemmaP [] xs) (rightIdP (slow xs))
```

We could not use any of these styles to generate a bad (non-)proof: neither `PEq ([a] → [a]) {fast} {bad}` nor `PEq ([a] → [a]) {slow} {bad}` are provable.

4.2 Succ: Refined Domains and Dependent Codomains

Our propositional equality `PEq` naturally reasons about functions with refined domains and dependent codomains. For example, recall the functions `incrInt` and `incrPos` from §2:

```
incrInt, incrPos :: Int → Int
incrInt n = n + 1
incrPos n = if 0 < n then n + 1 else 0
```

In §2 we proved that the two functions are equal on the domain of positive numbers:

```
type Pos = {x:Int | 0 < x }
posDom :: PEq (Pos → Int) {incrInt} {incrPos}
posDom = xEq incrInt incrPos $ \x →
```

Two correct and one wrong implementations of reverse

```
slow, bad, fast :: [a] → [a]
slow []     = []
slow (x:xs) = slow xs ++ [x]
bad xs = xs
```

```
fast xs = fastGo [] xs
fastGo :: [a] → [a] → [a]
fastGo acc []     = acc
fastGo acc (x:xs) = fastGo (x:acc) xs
```

First-Order Theorems relating `fast` *and* `slow`

```
reverseEq :: xs:[a] → { fast xs = slow xs }
lemma     :: xs:[a] → ys:[a] → {fastGo ys xs = slow xs ++ ys}
assoc     :: xs:[a] → ys:[a] → zs:[a] → { (xs ++ ys) ++ zs = xs ++ (ys ++ zs) }
rightId   :: xs:[a] → { xs ++ [] = xs }
```

Proofs of the First-Order Theorems

```
reverseEq x    = lemma x [] ? rightId (slow x)
lemma []    _  = ()
lemma (a:x) y  = lemma x (a:y) ? assoc (slow x) [a] y
x ? _pf        = x
```

```
rightId []       = ()
rightId (_:x)    = rightId x
assoc []     _ _ = ()
assoc (_:x) y z = assoc x y z
```

Figure 3. Reasoning about list reversal.

```
    bEq (incrInt x) (incrPos x) (reflP (incrInt x))
```

We can also reason about how each function's domain affects its codomain. For example, we can prove that these functions are equal *and* they take `Pos` inputs to natural numbers.

```
posRng :: PEq (Pos → {v:Int | 0 ≤ v})
              {incrInt} {incrPos}
posRng = xEq incrInt incrPos $ \x →
    bEq (incrInt x) (incrPos x) (reflP (incrInt x))
```

Finally, we can prove properties of the function's codomain that depend on the inputs. Below we show that on positive arguments, the result is always increased by one.

```
type SPos x = {v:Pos | v = x + 1}
depRng :: PEq (x:Pos → SPos {x})
              {incrInt} {incrPos}
depRng = xEq incrInt incrPos $ \x →
  bEq (incrInt x) (incrPos x) (reflP (incrInt x))
```

4.3 Map: Putting Equality in Context

Our propositional equality can be used in higher order settings; here, we prove that if two functions `f` and `g` are propositionally equal, then `map f` and `map g` are also equal. Our proofs use the substitutivity axiom `sEq`.

Equivalence on the Last Argument. Direct application of `sEq` ports a proof of equality to the first argument of the context (a function). For example, `mapEqP` below states that if two functions `f` and `g` are equal, then so are the partially applied functions `map f` and `map g`.

```
mapEqP :: f:(a → b) → g:(a → b)
        → PEq (a → b) {f} {g}
        → PEq ([a] → [b]) {map f} {map g}
mapEqP f g pf = sEq map f g pf
```

Equivalence on an Arbitrary Argument. To show that `map f xs` and `map g xs` are equal for all `xs`, we use `sEq` with `flipMap`, i.e., a context that puts `f` and `g` in a 'flipped' context.

```
mapEq :: f:(a → b) → g:(a → b)
       → PEq (a → b) {f} {g} → xs:[a]
       → PEq [b] {map f xs} {map g xs}
mapEq f g pf xs = sEq (flipMap xs) f g pf
                ? fMapEq f xs ? fMapEq g xs
```

```
fMapEq :: f:_ → xs:[a] → {map f xs = flipMap xs f}
fMapEq f xs  = ()
flipMap xs f = map f xs
```

The `mapEq` proof simply calls `sEq` with the flipped context and needs to know that `map f xs = flipMap xs f`. Liquid Haskell won't infer this fact on its own in the higher order setting of this proof; we explicitly provide this evidence with the calls to `fMapEq`. Finally, we use the `posDom` proof (§4.2) to show how to reuse proofs with `map`:

```
client :: xs:[Pos] →
        PEq [Int] {map incrInt xs} {map incrPos xs}
client = mapEq incrInt incrPos posDom
```

```
clientP :: PEq ([Pos] → [Int]) {map incrInt}
                                {map incrPos}
clientP = mapEqP incrInt incrPos posDom
```

`client` proves that `map incrInt xs` is equal to `map incrPos xs` for each list `xs` of positive numbers, while `clientP` proves that the partially applied functions `map incrInt` and `map incrPos` are equal on lists of positive numbers.

4.4 Fold: Equality of Multi-Argument Functions

As an example of equality proofs on multi-argument functions, we show that the directly tail-recursive `foldl` is equal to `foldl'`, a `foldr` encoding of a left-fold via CPS. The first-order equivalence theorem is expressed as follows:

```
thm :: f:(b → a → b) → b:b → xs:[a]
    → { foldl f b xs = foldl' f b xs }
```

We lift the first-order property into a multi-argument function equality by using `xEq` for all but the last arguments and `bEq` for the last, as below:

```
foldEq :: AEq b ⇒ PEq ((b → a → b) → b → [a]
       → b) {foldl} {foldl'}
foldEq = xEq foldl foldl' $ \f →
           xEq (foldl f) (foldl' f) $ \b →
             xEq (foldl f b) (foldl' f b) $ \xs →
               bEq (foldl f b xs) (foldl' f b xs)
                (thm f b xs ? reflP (foldl f b xs))
```

One can avoid the first-order proof and the `AEq` constraint, by using the second, typeclass-oriented style of §4.1 (see our implementation for details).

4.5 Spec: Function Equality for Program Efficiency

Function equality can be used to prove optimizations sound. For example, consider a `critical` function that, for safety, can only run on inputs that satisfy a specification `spec`, and `fastSpec`, a fast implementation to check `spec`.

```
spec, fastSpec :: a → Bool
critical :: x:{a | spec x} → a
```

A `client` function can soundly call `critical` for any input `x` by performing the runtime `fastSpec x` check, given a `PEq` proof that the functions `fastSpec` and `spec` are equal.

```
client :: PEq _ {fastSpec} {spec} → a → Maybe a
client pf x =
  if fastSpec x ? toSMT (fastSpec x) (spec x)
     (sEq (\x f → f x) fastSpec spec pf)
    then Just (critical x)
    else Nothing
```

The `toSMT` call generates the SMT equality that `fastSpec x = spec x`, which, combined with the branch condition check `fastSpec x`, lets the path-sensitive refinement type checker decide that the call to `critical x` is safe in the `then` branch.

Our propositional equality (1) co-exists with practical features of refinement types, e.g., path sensitivity, and (2) can help optimize executable code.

4.6 Associativity of Reader Monads

A *reader* is a function with a fixed domain `r`, i.e., the partially applied type `Reader r` (Figure 4, top left). Readers form a monad and are a popular way of defining and composing functions that take some fixed information, like command-line arguments or configuration files. We used propositional equality to prove that the `Reader` monad satisfies the standard functor, applicative, and monad laws. Here we present the associativity proof due to space constraints.

The monad instance for the reader type is defined using function composition (Figure 4, top). We also define Kleisli composition of monads as a convenience for specifying the monad. We express associativity using a refinement type and prove it using transitivity (Figure 4, bottom).

Proof by Associativity and Error Locality. In our associativity proof we use Haskell's let syntax to name the left (`el`) and right (`er`) expressions; our goal is to construct the proof `PEq _ {el} {er}`. To do so, we pick an intermediate term `em`; we attempt an equivalence proof as follows:

```
trans el em er
  (refl el)          -- proof of el  = em; local error
  (trans em emr er   -- proof of em  = er
    (refl em)        -- proof of em  = emr
    (refl emr))      -- proof of emr = er
```

This proving style comes with the great benefit of error locality. The `refl el` proof above will produce a type error; replacing that proof with an appropriate `trans` to connect `el` and `em` via `eml` completes the `monadAssociativity` proof (Figure 4, bottom). Writing proofs in this `trans`/`refl` style works well: start with `refl` and where the SMT solver can't figure things out, a local refinement type error tells you to expand with `trans` (or look for a counterexample).

5 A Refinement Calculus with Built-in Type-Indexed Equality

Because `funext` is inconsistent in Liquid Haskell (§2), we defined and axiomatized the type `PEq` to reason consistently about extensional equality (§3). We are able to prove interesting properties (§4) and Liquid Haskell's type checking seems to be doing the right thing. But how do we know that our definitions suffice? Formalizing *all* of Liquid Haskell is a challenge: we build a model to check our novel features. We formalize a core calculus λ^{RE} with refinement types, semantic subtyping, and type-indexed propositional equality.

λ^{RE} contains just enough to check the core interactions between refinement types and a type-indexed propositional equality resembling our `PEq` definition (§5.1). We omit plenty of important features from Liquid Haskell (e.g., algebraic data types, polymorphism): our purpose here is not to develop a complete formal model, but to check that our implementation holds together. Using λ^{RE}'s static semantics (§5.2), we prove several metatheorems (§5.3). Most importantly, a

Monad Instance for Readers

```
type Reader r a = r → a

kleisli :: (a → Reader r b)
        → (b → Reader r c)
        → a → Reader r c
kleisli f g x = bind (f x) g
```

```
pure :: a → Reader r a
pure a _r = a

bind :: Reader r a → (a → Reader r b) → Reader r b
bind fra farb = \r → farb (fra r) r
```

Associativity of Reader Monads

```
monadAssociativity :: (Transitivity c, Reflexivity c)
                   ⇒ m:(Reader r a) → f:(a → Reader r b) → g:(b → Reader r c)
                   → PEq (Reader r c) {bind (bind m f) g} {bind m (kleisli f g)}
monadAssociativity m f g = xEq (bind (bind m f) g) (bind m (kleisli f g)) $ \r →
  let { el  = bind (bind m f) g r ; eml = g (bind m f r) r ; em  = (bind (f (m r)) g) r
      ; emr = kleisli f g (m r) r ; er  = bind m (kleisli f g) r   }
  in trans el em er (trans el eml em (refl el) (refl eml)) (trans em emr er (refl em) (refl emr))
```

Figure 4. Case study: Reader Monad Proofs.

$$
\begin{array}{rcl}
c & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid (==_b) \mid (==_{(c,b)}) \\
e & ::= & c \mid x \mid e\ e \mid \lambda x{:}\tau.\ e \mid \mathsf{bEq}_b\ e\ e\ e \mid \mathsf{xEq}_{x:\tau\to\tau}\ e\ e\ e \\
v & ::= & c \mid \lambda x{:}\tau.\ e \mid \mathsf{bEq}_b\ e\ e\ v \mid \mathsf{xEq}_{x:\tau\to\tau}\ e\ e\ v \\
r & ::= & e \\
b & ::= & \mathsf{Bool} \mid () \\
\tau & ::= & \{x{:}b \mid r\} \mid x{:}\tau \to \tau \mid \mathsf{PEq}_\tau\ \{e\}\ \{e\} \\
\mathcal{E} & ::= & \bullet \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathsf{bEq}_b\ e\ e\ \mathcal{E} \mid \mathsf{xEq}_{x:\tau\to\tau}\ e\ e\ \mathcal{E} \\
\Gamma & ::= & \emptyset \mid \Gamma, x:\tau \\
\theta & ::= & \emptyset \mid \theta, x \mapsto v \\
\delta & ::= & \emptyset \mid \delta, (v,v)/x
\end{array}
$$

Reduction $\boxed{e \hookrightarrow e}$

$$
\begin{array}{rcll}
\mathcal{E}[e] & \hookrightarrow & \mathcal{E}[e'], & \text{if } e \hookrightarrow e' \\
(\lambda x{:}\tau.\ e)\ v & \hookrightarrow & e[v/x] & \\
(==_b)\ c_1 & \hookrightarrow & (==_{(c_1,b)}) & \\
(==_{(c_1,b)})\ c_2 & \hookrightarrow & c_1 = c_2, & \textit{syntactic equality on constants}
\end{array}
$$

Figure 5. Syntax and Dynamic Semantics of λ^{RE}.

logical relation characterizes λ^{RE} equivalence and reflects λ^{RE}'s propositional equality. Propositional equivalence in λ^{RE} implies equivalence in the logical relation (Theorem 5.3); both are reflexive, symmetric, and transitive (Theorems 5.1 and 5.2). Full details are in the extended version [42].

5.1 Syntax and Operational Semantics of λ^{RE}

We present λ^{RE}, a core calculus with *R*efinement types and type-indexed *E*quality (Figure 5). The core is standard CBV lambda calculus, extended with booleans and units. We add two primitives for proofs of propositional equality: bEq_b and $\mathsf{xEq}_{x:\tau_x\to\tau}$ construct proofs of equality at base and function types, respectively. Equality proofs take three arguments: the two expressions equated and a proof of their equality; proofs at base type are trivial, of type $()$, but higher types use functional extensionality. These two primitives correspond to bEq and xEq axioms of §3. We did not encode substitutivity, since it could be derived in our metatheory by induction on expressions. We have the axiom in Haskell, though, where we can only do (classy) induction on types.

λ^{RE} only refines *basic types*, booleans and unit; it uses *dependent function types* $x{:}\tau_x \to \tau$ with arguments of type τ_x and result type τ, where τ can refer back to the argument x. We add the *propositional equality* type $\mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}$, which denotes a proof of equality between the two expressions e_l and e_r of type τ. We write b to mean the trivial refinement type $\{x{:}b \mid \mathsf{true}\}$. We omit polymorphic types to simplify the metatheory [32]. We define function extensionality as a family of primitives xEq, one for each refinement function type, capturing the essence of polymorphic function equality.

The relation $\cdot \hookrightarrow \cdot$ evaluates λ^{RE} expressions using small step, call-by-value semantics (Figure 5, bottom; $\cdot \hookrightarrow^* \cdot$ is its reflexive, transitive closure). The semantics are standard; bEq_b and $\mathsf{xEq}_{x:\tau_x\to\tau}$ evaluate proofs but not equated terms.

5.2 Static Semantics of λ^{RE}

λ^{RE}'s static semantics has two parts: typing judgments (§5.2.1) and a binary logical relation capturing equivalence (§5.2.2).

5.2.1 Typing of λ^{RE}. Type checking in λ^{RE} uses three mutually recursive judgments (Figure 6): *type checking*, $\Gamma \vdash e :: \tau$, for when e has type τ in Γ; *well formedness*, $\Gamma \vdash \tau$, for when τ is well formed in Γ; and *subtyping*, $\Gamma \vdash \tau_l \preceq \tau_r$, for when τ_l is a subtype of τ_r in Γ.

Beyond the conventional rules for refinement type systems [18, 29, 30], the interesting rules are concerned with

Type checking (equality rules) $\boxed{\Gamma \vdash e :: \tau}$

$$\frac{\Gamma \vdash e_l :: \tau_l \quad \Gamma \vdash \tau_l \preceq \{x{:}b \mid \mathtt{true}\} \quad \Gamma \vdash e_r :: \tau_r \quad \Gamma \vdash \tau_r \preceq \{x{:}b \mid \mathtt{true}\} \quad \Gamma, l : \tau_l, r : \tau_r \vdash e :: \{x{:}() \mid l ==_b r\}}{\Gamma \vdash \mathsf{bEq}_b\ e_l\ e_r\ e :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}}\textsc{TEqBase}$$

$$\frac{\Gamma \vdash \tau_r \preceq x{:}\tau_x \to \tau \quad \Gamma \vdash \tau_l \preceq x{:}\tau_x \to \tau \quad \Gamma \vdash e_l :: \tau_l \quad \Gamma \vdash e_r :: \tau_r \quad \Gamma \vdash x{:}\tau_x \to \tau \quad \Gamma, l : \tau_l, r : \tau_r \vdash e :: (x{:}\tau_x \to \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\})}{\Gamma \vdash \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e :: \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}}\textsc{TEqFun}$$

Subtyping (all rules) $\boxed{\Gamma \vdash \tau \preceq \tau}$

$$\frac{\forall \theta \in [\![\Gamma]\!],\ [\![\theta \cdot \{x{:}b \mid r\}]\!] \subseteq [\![\theta \cdot \{x'{:}b \mid r'\}]\!]}{\Gamma \vdash \{x{:}b \mid r\} \preceq \{x'{:}b \mid r'\}}\textsc{SBase}$$

$$\frac{\Gamma \vdash \tau_x' \preceq \tau_x \quad \Gamma, x : \tau_x' \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau_x' \to \tau'}\textsc{SFun}$$

$$\frac{\Gamma \vdash \tau \preceq \tau' \quad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\} \preceq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}}\textsc{SEq}$$

Figure 6. Typing of λ^{RE} (selected rules about equality).

equality (TEqBase, TEqFun). We assign selfified types to `true`, `false`, and `unit` (e.g., $\{x{:}\mathsf{Bool} \mid x ==_{\mathsf{Bool}} \mathtt{true}\}$) [29]. Equality is given a similarly reflective type, with $\mathsf{TyCon}(==_b)$ defined as $x{:}b \to y{:}b \to \{z{:}\mathsf{Bool} \mid z ==_{\mathsf{Bool}} (x ==_b y)\}$. The rule TEqBase assigns to the expression $\mathsf{bEq}_b\ e_l\ e_r\ e$ the type $\mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$. To do so, we *guess* types τ_l and τ_r that fit e_l and e_r, respectively. Both these types should be subtypes of b that are *strong* enough to derive that if $l : \tau_l$ and $r : \tau_r$, then the proof argument e has type $\{_{:}() \mid l ==_b r\}$. Our formal model allows checking of strong, selfified types, but does not define an algorithmic procedure to generate them. In Liquid Haskell, type inference [30] automatically and algorithmically derives such strong types. We don't bother with inference: formally, we can guess any type that inference can derive.

The rule TEqFun gives the expression $\mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e$ type $\mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}$. As in TEqBase, we guess strong types τ_l and τ_r to stand for e_l and e_r such that with $l : \tau_l$ and $r : \tau_r$, the proof argument e should have type $x{:}\tau_x \to \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\}$, i.e., it should prove that l and r are extensionally equal. We require that the index $x{:}\tau_x \to \tau$ is well formed as technical bookkeeping.

As is common in refinement type systems, we use subtyping instead of conversion. SEq reduces subtyping of equality types to subtyping of the type indices, while the expressions to be equated remain unchanged. Covariant treatment of the type index would suffice for our metatheory, but we treat the type index invariantly to be consistent with the implementation, since the type index of PEq is not used in its definition and thus treated invariantly by Liquid Haskell.

Value equivalence relation $\boxed{v \sim v :: \tau; \delta}$

$$c \sim c :: \{x{:}b \mid r\}; \delta \iff \vdash_B c :: b \wedge \delta_1 \cdot r[c/x] \hookrightarrow^* \mathtt{true} \wedge \delta_2 \cdot r[c/x] \hookrightarrow^* \mathtt{true}$$
$$v_1 \sim v_2 :: x{:}\tau_x \to \tau; \delta \iff \forall v_3 \sim v_4 :: \tau_x; \delta.\ v_1\ v_3 \sim v_2\ v_4 :: \tau; \delta, (v_3, v_4)/x$$
$$v_1 \sim v_2 :: \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}; \delta \iff \delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: \tau; \delta$$

Expression equivalence relation $\boxed{e \sim e :: \tau; \delta}$

$$e_1 \sim e_2 :: \tau; \delta \iff \exists v_1 v_2,\ e_1 \hookrightarrow^* v_1 \wedge e_2 \hookrightarrow^* v_2 \wedge v_1 \sim v_2 :: \tau; \delta$$

Open expression equivalence relation $\boxed{\delta \in \Gamma}$ $\boxed{\Gamma \vdash e \sim e :: \tau}$

$$\delta \in \Gamma \doteq \forall x : \tau \in \Gamma,\ \delta_1(x) \sim \delta_2(x) :: \tau; \delta$$
$$\Gamma \vdash e_1 \sim e_2 :: \tau \doteq \forall \delta \in \Gamma,\ \delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau; \delta$$

Figure 7. The binary logical relation defining equivalence.

Our subtyping rule allows equality proofs between functions with convertible types (§4.2). The subtyping rule for refinements depends on a unary logical relation, i.e., a semantic typing relation [16][4]. The interpretation of base-type equalities $\mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$ includes all proofs whose first arguments reduce to equal b-constants. The interpretation of the function equality type $\mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}$ includes all expressions that satisfy the basic typing and whose proof argument produces appropriate proofs.

5.2.2 Equivalence Logical Relation for λ^{RE}. We characterize equivalence with a term-model binary logical relation. We lift a relation on closed values to closed and then open expressions (Figure 7). Instead of directly substituting in type indices, all three relations use *pending substitutions* δ, which map variables to pairs of equivalent values. The value relation $v_1 \sim v_2 :: \tau; \delta$ is defined as a fixpoint on types, noting that the propositional equality on a type, $\mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$, is structurally larger than the type τ. Two proofs of equality are equivalent when the two equated expressions are equivalent in the logical relation at type-index τ—equality proofs 'reflect' the logical relation.

Since the equated expressions appear in the type itself, they may be open, referring to variables in the pending substitution δ. Thus we use δ to close these expressions, using the logical relation on $\delta_1 \cdot e_l$ and $\delta_2 \cdot e_r$. Proofs aren't computationally relevant in refinement typing, so the logical relation doesn't even *inspect* the proofs v_1 and v_2 themselves. Two open expressions, with variables from Γ are equivalent on type τ, written $\Gamma \vdash e_1 \sim e_2 :: \tau$, *iff* for each δ that satisfies Γ, we have $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau; \delta$. Here e_1, e_2, and τ might

[4]We could probably get away with having just the binary logical relation we use for equivalence (§5.2.2). We found it easier to do one metatheoretical thing at a time.

refer to variables in the environment Γ. We use δ to close the expressions eagerly, while we close the type lazily.

5.3 Metaproperties: PEq is an Equivalence Relation

We show various metaproperties of λ^{RE}. All the proofs, along with type soundness, are in the extended version [42].

Theorem 5.1 (LR is an Equivalence). *$\Gamma \vdash e_1 \sim e_2 :: \tau$ is reflexive, symmetric, and transitive.*

Our relation is an equivalence. Transitivity requires reflexivity on e_2, so we assume that $\Gamma \vdash e_2 :: \tau$.

Theorem 5.2 (PEq is an Equivalence). *For all τ that do not contain equalities themselves:*

- *Reflexivity: If $\Gamma \vdash e :: \tau$, then there exists v such that $\Gamma \vdash v :: \mathsf{PEq}_\tau\ \{e\}\ \{e\}$.*
- *Symmetry: If $\Gamma \vdash v_{12} :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$, then there exists v_{21} such that $\Gamma \vdash v_{21} :: \mathsf{PEq}_\tau\ \{e_2\}\ \{e_1\}$.*
- *Transitivity: If $\Gamma \vdash v_{12} :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ and $\Gamma \vdash v_{23} :: \mathsf{PEq}_\tau\ \{e_2\}\ \{e_3\}$ and $\Gamma \vdash e_2 :: \tau$, then there exists v_{13} such that $\Gamma \vdash v_{13} :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_3\}$.*

The proofs go by induction on τ. Reflexivity requires a generalized the IH to find appropriate τ_l and τ_r for the PEq proofs.

Theorem 5.3 (PEq is Sound). *If $\Gamma \vdash e :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$, then $\Gamma \vdash e_1 \sim e_2 :: \tau$.*

The proof is a corollary of the fundamental property of the logical relation, i.e., reflexivity.

6 Related Work

Functional Extensionality and SMT-Based Subtyping. F*, like Liquid Haskell, combines semantic subtyping with SMT equality and in 2018 its developer team reported in an (impenetrable by non refinement type experts) github issue[1] exactly the same inconsistency of the naive `funext` encoding. We hope that our detailed explanation of the inconsistency (§ 2) will prevent developers of similar systems rediscovering it. As a solution [14], F* defines an extensionality axiom that makes a more roundabout connection with SMT: function equality uses ==, which is a proof-irrelevant, propositional Leibniz equality, which they assume coincides with SMT equality. F*'s approach requires that refined types appear as program terms, and it is not viable in Liquid Haskell, where the Haskell compiler defines program terms knowing nothing about refinement types. Our PEq definition does not have the full power of F*'s inductive definitions, but can be implemented without changing Liquid Haskell itself. Dafny's SMT encoding axiomatizes extensionality for data, but not for functions [21]. Function equality is neither provable nor disprovable in their encoding into Z3. Ou et al. [29] introduce *selfification*, which assigns singleton types using equality. SAGE assigns selfified types to *all* variables, including functions [19]. Dminor lacks first-class functions [7].

Equality in Dependent Type Theories. Equality in general and functional extensionality in particular have a rich history of study. Our work shares a common goal with OTT [1] and cubical type theories [3, 10, 36]: we want to assign different meanings to equality at different types. We also use common tools, namely, `funext`. There the similarities stop: OTT and cubical type theories are Martin-Löf type theories, while Liquid Haskell is based on subtyping, implications resolved by SMT, and refinement of a simply typed core (NuPRL [11] and RedPRL [2] are untyped languages with equivalence, so they bear some similarity, too). Because of all three of these issues, but especially because of implicit subtyping (i.e., without proof terms/coercions), type theoretic approaches do not directly address our problem.

XTT [36] is perhaps the most closely related. They define a single notion of (cubical) equality which admits extensionality. To use equality, they close the universe of types and do typecase, not unlike our PEq. The details are quite different: XTT is a cubical type theory with a notion of proof; in XTT, equality between M and N is a function that maps an interval to a path starting in M and ending at N. But their typecase strategy morally resembles our PEq and the typeclass-based approach: both dissect the universe of types by structure, singling out functions for special treatment. Their type-directed notion of coercion is different from how we use equality. XTT's coercions take a proof that $M = N$ and compute, transforming M into N. Our equalities are purely static, computationally irrelevant, and typically bottom out by handing off to the SMT solver. Approaches like XTT's may play better with F*'s approach using dependent, inductive types than the 'flatter' approach that we propose here. Univalent parametricity [40] is also closely related: they use a type-indexed notion of equivalence; their implementation uses classy induction with the `IsEquiv` typeclass. We focus on SMT interactions, while they are interested in univalence.

Dependent type theories often care about equalities between equalities, with axioms like UIP (all identity proofs are the same), K (all identity proofs are the same, namely `refl`), and univalence (identity proofs are isomorphisms, and so not the same). F*'s squashed equality is proof-irrelevant, with at most one proof equating any given pair of terms. If we allowed equalities between equalities, we could add UIP. Our propositional equality doesn't come with a reflexivity constructor, so axiom K would be less natural to encode, as it's not obvious what to choose as the canonical proof of equality.

Depending on the details of the theory, a `refl` constructor with an extensionality axiom generates the same equality as our type-indexed PEq. While we haven't investigated the metatheory, we conjecture that our proofs work when we replace `bEq` with a notion of reflexivity: we need to shuffle around `AEq` constraints. In F* and type theory, reflexivity is a natural choice for defining equality, since dependent pattern matching turns a `refl` constructor into Leibniz equality.

But Liquid Haskell, by design, lacks true dependent pattern matching: in both cases we are using an axiom. Since it's also important in Liquid Haskell to distinguish SMT equality from language equality (via `AEq`), on balance, we prefer the type-indexed equality: it brings that distinction to the fore, while laying the groundwork for other, more nuanced approaches to equality (e.g., non-syntactic set equality).

Classy Induction: Inductive Proofs Using Typeclasses. We used 'classy induction' to prove metaproperties of `PEq` inside Liquid Haskell (§3.3), using ad-hoc polymorphism and general instances to generate proofs that 'cover' some class of types. We did not *invent* classy induction—it is a folklore technique that we named. We have seen six independent uses of "classy induction" in the literature [8, 13, 17, 20, 40, 45]. Any typeclass system that accommodates ad-hoc polymorphism and a notion of proof can use classy induction. Sozeau [34] generates proofs of nonzeroness using something akin to classy induction, though it goes by induction on the operations used to build up arithmetic expressions in the (dependent!) host language (§6.3.2); he calls this the 'programmation logique' aspect of typeclasses. Instance resolution is characterized as proof search over lemmas (§7.1.3). Sozeau and Oury [35] introduce typeclasses to Coq; their system can do classy induction, but they don't show it in the paper. Earlier work on typeclasses focused on overloading [27, 28, 44], with no notion of classy induction even with proofs [46].

Monads. Much prior work tests, reasons about, and verifies monads [4, 9, 15, 23–26, 33, 38, 39, 48]. We have only verified the correctness of a monad instance; scaling to monadic computations is a good next step.

7 Conclusion

In a refinement type system with subtyping a naive encoding of `funext` is inconsistent. We explained the inconsistency by examples (that proved `false`), we implemented a type-indexed propositional equality that avoids this inconsistency, and validated it with a model calculus. Several case studies demonstrate the range, effectiveness, and power of our work.

Acknowledgments

We thank Conal Elliott for his help in exposing the inadequacy of the naive functional extensionality encoding. Éric Tanter, Stephanie Weirich, and Nicolas Tabareau offered valuable insights into the folklore of classy induction. This work is partially founded by the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), the US Office of Naval Research HACKCRYPT (Ref. N00014-19-1-2292), the Atracción de Talento grant (Ref. 2019-T2/TIC-13455), and the Juan de la Cierva grant (IJC2019-041599-I).

References

[1] Thorsten Altenkirch and Conor McBride. 2006. Towards Observational Type Theory. http://www.strictlypositive.org/ott.pdf Unpublished manuscript.

[2] Carlo Angiuli, Evan Cavallo, Kuen-Bang Hou (Favonia), Robert Harper, and Jonathan Sterling. 2018. The RedPRL Proof Assistant (Invited Paper). In *Theoretical Computer Science*. https://doi.org/10.4204/eptcs.274.1

[3] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. 2018. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In *Computer Science Logic*. http://drops.dagstuhl.de/opus/volltexte/2018/9673

[4] Robert Atkey and Patricia Johann. 2015. Interleaving Data and Effects. In *Journal of Functional Programming*. https://doi.org/10.1017/S0956796815000209

[5] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. 2019. Extending SMT Solvers to Higher-Order Logic. In *CADE*. https://doi.org/10.1007/978-3-030-29436-6_3

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. *The SMT-LIB Standard: Version 2.0.* Technical Report. Department of Computer Science, The University of Iowa. www.SMT-LIB.org

[7] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. 2012. Semantic Subtyping with an SMT Solver. In *Journal of Functional Programming*. https://doi.org/10.1017/S0956796812000032

[8] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Certified Programs and Proofs*. https://doi.org/10.1145/3018610.3018620

[9] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. In *GPCE*. https://doi.org/10.1145/636517.636527

[10] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2015. Cubical Type Theory: a Constructive Interpretation of the Univalence Axiom. In *Types for Proofs and Programs*. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

[11] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

[12] Robert L Constable and Scott Fraser Smith. 1987. *Partial objects in constructive type theory*. Technical Report. Cornell University. https://dl.acm.org/doi/10.5555/866226

[13] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of Dependent Interoperability. In *Journal of Functional Programming*. https://doi.org/10.1017/S0956796818000011

[14] Github FStarLang. 2018. Functional Equality Discussions in F*. https://github.com/FStarLang/FStar/blob/cba5383bd0e84140a00422875de21a8a77bae116/ulib/FStar.FunctionalExtensionality.fsti#L133-L134 and https://github.com/FStarLang/FStar/issues/1542 and https://github.com/FStarLang/FStar/wiki/SMT-Equality-and-Extensionality-in-F%2A.

[15] Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. In *ICFP*. https://doi.org/10.1145/2034773.2034777

[16] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2012. Contracts Made Manifest. In *Journal of Functional Programming*. https://doi.org/10.1017/S0956796812000135

[17] Louis-Julien Guillemette and Stefan Monnier. 2008. A Type-Preserving Compiler in Haskell. In *ICFP*. https://doi.org/10.1145/1411204.1411218

[18] Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. In *ACM Transactions on Programming Languages and Systems*. https://doi.org/10.1145/1667048.1667051

[19] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*. http://www.cs.williams.edu/~freund/papers/06-sfp.pdf

[20] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. In *POPL*. https://doi.org/10.1145/3158133

[21] K. Rustan M. Leino. 2012. Developing verified programs with Dafny. In *High Integrity Language Technology*. https://doi.org/10.1145/2402676.2402682

[22] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell. In *OOPSLA*. https://doi.org/10.1145/3428284

[23] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. In *ICFP*. https://doi.org/10.1145/3341708

[24] E. Moggi. 1989. Computational Lambda-Calculus and Monads. In *LICS*. https://doi.org/10.1109/LICS.1989.39155

[25] Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *ESOP*, Nobuko Yoshida (Ed.). https://doi.org/10.1007/978-3-030-72019-3_17

[26] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. In *ICFP*. https://doi.org/10.1145/1159803.1159812

[27] Tobias Nipkow and Christian Prehofer. 1993. Type Checking Type Classes. In *POPL*. https://doi.org/10.1145/158511.158698

[28] Tobias Nipkow and Gregor Snelting. 1991. Type Classes and Overloading Resolution via Order-Sorted Unification. In *Functional Programming Languages and Computer Architecture*. https://doi.org/10.1007/3540543961_1

[29] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *International Conference on Theoretical Computer Science*. https://doi.org/10.1007/1-4020-8141-3_34

[30] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *PLDI*. https://doi.org/10.1145/1375581.1375602

[31] John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering* (1998). https://doi.org/10.1109/32.713327

[32] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. In *ACM Transactions on Programming Languages and Systems*. https://doi.org/10.1145/2994594

[33] Lucas Silver and Steve Zdancewic. 2021. Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees. In *POPL*. https://doi.org/10.1145/3434307

[34] Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants.* Ph. D. Dissertation. Université Paris 11. https://tel.archives-ouvertes.fr/tel-00640052

[35] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*. https://doi.org/10.1007/978-3-540-71067-7_23

[36] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2019. Cubical Syntax for Reflection-Free Extensional Equality. In *LIPIcs*. https://doi.org/10.4230/LIPIcs.FSCD.2019.31

[37] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL*. http://dx.doi.org/10.1145/2837614.2837655

[38] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In *PLDI*. https://doi.org/10.1145/2491956.2491978

[39] Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *TPHOLs*. https://doi.org/10.1007/978-3-642-03359-9_30

[40] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The Marriage of Univalence and Parametricity. *J. ACM*. https://doi.org/10.1145/3429979

[41] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Haskell*. https://doi.org/10.1145/3242744.3242756

[42] Niki Vazou and Michael Greenberg. 2021. How to Safely Use Extensionality in Liquid Haskell (extended version). In *CoRR*. https://doi.org/10.48550/ARXIV.2103.02177

[43] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement Reflection: Complete Verification with SMT. In *POPL*. https://doi.org/10.1145/3158141

[44] P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *POPL*. https://doi.org/10.1145/75277.75283

[45] Stephanie Weirich. 2017. The Influence of Dependent Types (Keynote). In *POPL*. https://doi.org/10.1145/3009837.3009923

[46] Markus Wenzel. 1997. Type Classes and Overloading in HigherOorder Logic. In *Theorem Proving in Higher Order Logics*. https://doi.org/10.1007/BFb0028402

[47] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *PLDI*. https://doi.org/10.1145/277650.277732

[48] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. In *POPL*. https://doi.org/10.1145/3371119