Functional Extensionality for Refinement Types

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

Michael Greenberg
michael.greenberg@pomona.edu
Pomona College
Claremont, CA, 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, porting the proofs to an optimized implementation that behaves the same. Without functional extensionality, proofs must relate functions that are fully applied. When data itself has a higher-order representation, fully applied proofs face serious impediments! When working with first-order data, fully applied proofs lead to noisome duplication when using higher-order functions.

While dependent type theories are typically consistent with functional extensionality axioms, SMT-backed refinement type systems treat naïve phrasings of functional extensionality inadequately, leading to *unsoundness*. We extend a refinement type theory with a type-indexed propositional equality that is adequate for SMT. We implement our theory in `PEq`, a Liquid Haskell library that defines propositional equality and apply `PEq` to several case studies. We prove metaproperties of `PEq` inside Liquid Haskell using an unnamed folklore technique, which we dub 'classy induction'.

1 Introduction

Refinement types have been extensively used to reason about functional programs [5, 22, 23, 30, 41]. Higher-order functions are a key ingredient of functional programming, so reasoning about function equality within refinement type systems is unavoidable. For example, Vazou et al. [33] prove function optimizations correct by specifying equalities between fully applied functions. Do these equalities hold in the context of higher order function (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.

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 that allow for refinement types, are no exception: function equalities come up regularly. But, in both these systems, the first attempt to give an axiom for functional extensionality was wrong[1]. A naïve funext axiom proves equalities between unequal functions.

Our first contribution is to expose why a naïve function equality encoding is unutterble (§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` does not encode function equality, since it is not reasoning about equality on the domains of the functions. What if type inference instantiates 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? Turns out that this naïve extensionality axiom belongs to a syntactic category of axioms that cannot be uttered using refinement types (§2). We work in Liquid Haskell, but the problem generalizes to any refinement type system that allows for polymorphism, semantic subtyping, and refinement type inference. Proofs of function equality must carry information about the domain type on which the compared functions are equal.

Our second contribution is to formalize λ^{RE}, a core calculus that circumvents the inadequacy of the naïve encoding (§3). We prove that λ^{RE}'s refinement types and type-indexed, functionally extensional propositional equality is sound; propositional equality implies equality in a term model.

Our third contribution is to implement λ^{RE} as a Liquid Haskell library (§4). We implement λ^{RE}'s type-indexed propositional equality using Haskell's GADTs and Liquid Haskell's refinement types. We call the propositional equality `PEq` and find that it adequately reasons about function equality. 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 induction on the structure of the type index—we applied an heretofore-unnamed folklore proof methodology, which we dub *classy induction* (§4.3).

Our final contribution is to use `PEq` to prove equalities between functions (§5). As simple examples, we prove optimizations correct as equalities between functions (i.e., `reverse`),

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/1122445.1122456

[1] See https://github.com/FStarLang/FStar/issues/1542 for F*'s initial, wrong encoding and §6 for F*'s different solution. The LH case is elaborated in §2.

$$\texttt{axiom ::}\ \overbrace{t_1 \to \cdots \to (x:a \to \dots) \to \cdots \to t_n}^{\text{Premises}} \to t$$

Positive only variable Empty Premise

Figure 1. Unutterable Axioms and Empty Premise.

work carefully with functions that only agree on certain domains and dependent ranges, lift equalities to higher-order contexts (i.e., `map`), prove equivalences with multi-argument higher-order functions (i.e., `fold`), and showcase how higher-order, propositional equalities can co-exist with and speedup executable code. Finally, we provide a more substantial case study, proving the monad laws for reader monads.

2 The Problem: Naive Function Extensionality is Unutterable

Function extensionality cannot be soundly expressed as an axiom in refinement type systems. In this section, we identify a syntactic class of logical axioms that cannot be uttered in polymorphic refinement types (§ 2.1) and we explain why by type checking the simplest such axiom (§ 2.2). Finally, we show that function extensionality is an unutterable axiom and provide a type level interpretation of our claim (§ 2.3).

2.1 Unutterable Axioms

Logical formulas can be expressed as types [36], by encoding foralls and existentials as dependent functions and dependent pairs, respectively. In theory, any logical axiom can be encoded as an assumption in a refinement type setting. In practice, there is a syntactic class of axioms, we call *unutterable*, that cannot be soundly assumed in refinement type systems with polymorphism and type inference.

Figure 1 presents the general syntactic structure of unutterable axioms. The `axiom` has premises, encoded as the arguments $t_1 \dots t_n$, and the conclusion t. In this axiom, the type variable a appears only on positive positions, e.g., as a higher-order argument in a premise. We call the premise that contains positive type variables as arguments an *empty premise*, because these premises are not checked when the axiom is invoked (§ 2.2). Since the empty premises are not checked, unutterable axioms are "unsound": the conclusion can be established without actually checking the premises.

Example: No Empty Types is Unutterable An example of an unutterable axiom is the proposition that "every type has an inhabitant", i.e., there are no empty types. We express this proposition as a logical formula that if `false` holds for all x (i.e., x can have an empty type), then `false` itself holds:

$$\mathsf{hab} \doteq (\forall x.\texttt{false}) \Rightarrow \texttt{false}$$

The proposition `hab` can be encoded as an assumed refinement type, where forall and implication are encoded as (dependent) function types:

```
assume hab :: (x:a → {false}) → {false}
```

The premise of the axiom `hab` is empty: when invoking `hab` the premise $\forall x.\texttt{false}$ will not actually be checked. That is, the client below *soundly* type checks in Liquid Haskell.

```
clientH :: {false}          evidH :: x:b → ()
clientH = hab evidH         evidH _ = ()
```

The evidence the client provides is simply `x:b → ()`, i.e., $\forall\texttt{b.true}$ in logic, which does not imply the premise $\forall\texttt{b.false}$. Yet, the client does soundly type check (as explained in § 2.2).

Schematic for Unutterable Axioms We generalize our example to a minimal schema for unutterable axioms, by abstracting the false predicates in the premise and conclusion of `hab` to, respectively p_x and p. The generalized proposition now states that "if forall x, p_x holds, then p holds" and its encoding as formula and refinement type are shown below:

$$\mathsf{unA} \doteq (\forall x.p_x) \Rightarrow p$$

```
assume unA :: (x:a → {px}) → {p}
```

Similarly, we generalize the client of `unA` and the evidence:

```
client :: {p}          evid :: x:b → {q}
client = unA evid      evid _ = ...
```

The above code *soundly type checks*, even though it does not adequately match the interpretation that `unA`'s (empty) premise is checked: The conclusion of the unutterable axiom `unA` (here p) can be derived even when the evidence (here $\forall x.q$) is not sufficient to ensure the premise (here $\forall x.p_x$).

2.2 Checking the Empty Premise

Why does the above `client` "soundly" type check? Here we construct `client`'s refinement type checking derivation tree.

Type checking Environment Type checking occurs in the below environment that contains the axiom `unA` and the evidence `evid` with types generalized over p_x, p, and q.

$$\Gamma \doteq \left\{ \begin{array}{lll} \mathsf{unA} & :: & \forall a.(x:a \to \{p_x\}) \to \{p\} \\ , \ \mathsf{evid} & :: & x:b \to \{q\} \end{array} \right\}$$

Type Instantiation Before type checking, `unA evid` is explicitly type instantiated. In Hindley–Milner type inference, the polymorphic type variable instantiates to `b`. In a refinement type system, instantiation uses a refined type `{v:b | κ}` where κ is an *unknown* refinement variable, to be inferred.

After explicit type instantiation the call to `unA` will be elaborated to `unA @{v:b | κ} evid`.

The refinement type instantiation rule below checks the instantiated expression by substituting the polymorphic type variable with the explicitly applied type.

$$\frac{\Gamma \vdash \mathsf{unA} :: \forall a.(x:a \to \{p_x\}) \to \{p\}}{\Gamma \vdash \mathsf{unA}\ @\{v{:}\mathsf{b} \mid \kappa\} :: (x:\{v{:}\mathsf{b} \mid \kappa\} \to \{p_x\}) \to \{p\}}\ \textsc{TI}$$

Function Application Next, the instantiated axiom is applied to the `evid` argument using the function application:

$$\frac{\begin{array}{c}(1)\ \Gamma \vdash \texttt{evid} :: x : \mathsf{b} \to \{q\} \\ (2)\ \Gamma \vdash \mathsf{unA}\ @\{v{:}\mathsf{b} \mid \kappa\} :: (x : \{v{:}\mathsf{b} \mid \kappa\} \to \{p_x\}) \to \{p\} \\ (3)\ \Gamma \vdash x : \mathsf{b} \to \{q\} \ \preceq\ x : \{v{:}\mathsf{b} \mid \kappa\} \to \{p_x\}\end{array}}{\Gamma \vdash (\mathsf{unA}\ @\{v{:}a \mid \kappa\})\ \texttt{evid} :: \{p\}}$$

The argument premise (1) gets the type of `evid` (here, from the typing environment). The function premise (2) uses the derivation `TI` from above. Finally, the subtyping premise (3) checks that the type of the actual argument is a subtype of the argument of the function and is explained below.

Subtyping Below the premise (3) reduces to implications:

$$\frac{\dfrac{(4)\ \kappa \Rightarrow \texttt{true}}{\Gamma \vdash \{v{:}\mathsf{b} \mid \kappa\} \ \preceq\ \mathsf{b}} \qquad \dfrac{(5)\ \kappa \Rightarrow q \Rightarrow p_x}{\Gamma, x : \{\mathsf{b} \mid \kappa\} \vdash \{q\} \ \preceq\ \{p_x\}}}{\Gamma \vdash x{:}\mathsf{b} \to \{q\} \ \preceq\ x{:}\{\mathsf{b} \mid \kappa\} \to \{p_x\}}$$

The rule checks subtyping of the argument and result in the usual contra- and co-variant ways, respectively. In both cases, subtyping on basic types reduces to implication checking.

Implication Checking Type checking of the `client` code reduces to the validity of the following two implications

$$(4) \quad \kappa \Rightarrow \texttt{true} \qquad\qquad (5) \quad \kappa \Rightarrow q \Rightarrow p_x$$

On the surface, the implication system seems like a good encoding. Implication (4) ensures κ implies the refinement of the evidence domain (i.e., true). Assuming κ, implication (5) ensures q is sufficient to prove p_x. So far, so good: we've correctly implemented contravariance of functions. If the refinement variable κ *unified* to `true` (per (4)), the implication system adequately *checks* the `unA`'s premise. Unfortunately, the implication system won't choose `true`—inference is free to choose any valid solution for κ.

Here's the bad news. The implication system has a trivial, very specific solution: set κ to `false`. Such a solution is valid: type checking succeeds. Liquid type inference [22] always returns the strongest solution for the refinement variables, and so it will set κ to `false`. This choice is natural enough in light of `unA`'s type. The type variable a only appears in a positive position. Since functions are contravariant, `unA` never actually touches a value of type a—so Liquid Haskell (soundly!) infers the strongest possible refinement, setting κ to `false`, since no value of the type is never actually used.

2.3 Function Extensionality is Unutterable

One might be tempted to naively encode function extensionality an axiom that states that for each argument function `f` and `g`, if you have a proof that forall x, `f x = g x`, then you get a proof that the two functions are equal.

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

Notice though, that the axiom is polymorphic with respect to both the function domain `a` and the function range `b`. Critically, the domain variable `a` appears in positive positions only, thus the premise `x:a → {f x = g x}` is empty, i.e., it is not going to get checked when we use `funext`.

For example, suppose we had two concrete functions `h` and `k`, with refined domains and ranges, and a lemma proving `q`:

```
h :: x:{v:a | dh} → {v:b | rh}
k :: x:{v:a | dk} → {v:b | rk}
lemma :: x:a → {q}
```

A call to `funext` that equates the functions `h` and `k` will, as in the `client` previous example, explicitly instantiate the type variables using two refinement variables, as below

```
thmEq :: { h = k }
thmEq = funext @{v : α | κα} @{v : β | κβ} h k lemma
```

Type checking the above call reduces to solving the below set of logical implications (the derivation is presented in [29])

(1)	$\kappa_\alpha \Rightarrow d_{\mathsf{h}}$	(2)	$\kappa_\alpha \Rightarrow d_{\mathsf{k}}$
(3)	$\kappa_\alpha \Rightarrow \texttt{true}$	(4)	$\kappa_\alpha \Rightarrow r_{\mathsf{h}} \Rightarrow \kappa_\beta$
(5)	$\kappa_\alpha \Rightarrow r_{\mathsf{k}} \Rightarrow \kappa_\beta$	(6)	$\kappa_\alpha \Rightarrow q \Rightarrow h\ x == k\ x$

As in § 2.2, the implication system seems like a good encoding. Implications (1) and (2) ensure κ_α is at least as restrictive as the two functions' domains. Assuming κ_α, implications (4) and (5) ensure κ_β is at least as inclusive as the two functions' ranges. Finally, implication (6) ensures that κ_α and the property p jointly imply first order equality of the two applications, $h\ x = k\ x$. To sum up: if we can find a common domain, the implication system will check that every application of the two functions on that domain yield equal results. If the domains d_{k} and d_{h} unify to κ_α, the implication system adequately *checks* function extensionality.

Unfortunately, type inference will choose a meaningless domain, and set κ_α to `false`. Later, we will forget that choice of trivial domain and apply the equality at any domain.

Type Level Interpretation of Trivial Domains Our naïve extensionality axiom is *unutterable*: it relates all functions and doesn't mean much, since we're finding equality on a *trivial*, empty domain. The axiom doesn't generate any inconsistency or unsoundness itself: arbitrary functions `h` and `k` really *are* equal on the empty domain. Rather, when we *use* `thmEq`, unsoundness strikes: we have `h = k` with nothing to remark on the (trivial!) types at which they're equal. Any use of `thmEq` will freely substitute `h` for `k` at any domain.

To address this problem, the type variable α representing the unified domain of the functions to be checked equal should appear in a negative position to exclude trivial domains. In other words, function equality cannot be expressed as a mere refinement, but must be expressed as a type that also records the domains on which the functions are equal.

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

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 2. Syntax and Dynamic Semantics of λ^{RE}.

3 The Solution: Explicitly Typed Equality

We formalize a core calculus λ^{RE} with refinement types and type-indexed propositional equality. First, we define the syntax and dynamic semantics of λ^{RE} (§3.1). Next, we define the typing judgement and a logical relation characterizing equivalence of λ^{RE} expressions (§3.2). Finally, we prove that λ^{RE} is sound, and that both the logical relation and the propositional equality satisfy the three equality axioms (§3.3).

3.1 Syntax and Semantics of λ^{RE}

Figure 2 presents λ^{RE}, a core calculus with *R*efinement types extended with typed *E*quality primitives.

Expressions λ^{RE} expressions include constants (booleans, unit, and equality operations on base types), variables, lambda abstraction, and application. There are also two primitives to prove propositional equality: $\mathtt{bEq}_b$ and $\mathtt{xEq}_{x:\tau_x\to\tau}$ construct proofs of equality at base and function types, resp.. 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.

Values The values of λ^{RE} are constants, functions, and equality proofs with converged proofs.

Types λ^{RE}'s *basic types* are booleans and unit. Basic types are refined with boolean expressions r in *refinement types* $\{x{:}b \mid r\}$, which denote all expressions of base type b that satisfy the refinement r. In addition to refinements, λ^{RE}'s types also include *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. Finally, types include our *propositional equality* $\mathtt{PEq}_\tau\ \{e_1\}\ \{e_2\}$, which denotes a proof of equality between the two expressions e_1 and e_2 of type τ. We write b to mean the trivial refinement type $\{x{:}b \mid \mathtt{true}\}$. To keep our formalism and metatheory simple, we omit polymorphic types; we could add them following Sekiyama et al. [24].

Environments The typing environment Γ binds variables to types, the (semantic typing) closing substitutions θ binds variables to values, and the (logical relation) pending substitutions δ binds variables to pairs of equivalent values.

Runtime Semantics The relation $\cdot \hookrightarrow \cdot$ evaluates λ^{RE} expressions using contextual, small step, call-by-value semantics (Figure 2, bottom). The semantics are standard with $\mathtt{bEq}_b$ and $\mathtt{xEq}_{x:\tau_x\to\tau}$ evaluating proofs but not the equated terms. Let $\cdot \hookrightarrow^* \cdot$ be the reflexive, transitive closure of $\cdot \hookrightarrow \cdot$.

Type Interpretations Semantic typing uses a unary logical relation to interpret types in a syntactic term model (closed terms, Figure 3; open terms, Figure 4).

The interpretation of the base type $\{x{:}b \mid r\}$ includes all expressions which yield b-value v that satisfy the refinement, i.e., r evaluates to true on v. To decide the unrefined type of an expression we use $\vdash_B e :: b$ (defined in supplementary material). The interpretation of function types $x{:}\tau_x \to \tau$ is *logical*: it includes all expressions that yield τ-results when applied to τ_x arguments. The interpretation of base-type equalities $\mathtt{PEq}_b\ \{e_l\}\ \{e_r\}$ includes all expressions that satisfy the basic typing ($\mathtt{PBEq}_\tau$ is the unrefined version of $\mathtt{PEq}_\tau\ \{e_l\}\ \{e_r\}$) and reduce to a basic equality proof whose first arguments reduce to equal b-constants. Finally, the interpretation of the function equality type $\mathtt{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}$ includes all expressions that satisfy the basic typing (based on the $\lfloor\cdot\rfloor$ operator; supplementary material). These expressions reduce to a proof (noted as `xEq_`, since the type index does not need to be syntactically equal to the index of the type) whose first two arguments are functions of type $x{:}\tau_x \to \tau$ and the third proof argument takes τ_x arguments to equality proofs of type $\mathtt{PEq}_{\tau[e_x/x]}\ \{e_l\ e_x\}\ \{e_r\ e_x\}$.

Constants For simplicity, λ^{RE} constants are only the two boolean values, unit, and equality operators for basic types. For each b, we define the type indexed "computational" equality $==_b$. For two constants c_1 and c_2 of basic type b, $c_1\ ==_b\ c_2$ evaluates in one step to $(==_{(c_1,b)})\ c_2$, which then steps to `true` when c_1 and c_2 are the same and `false` otherwise.

Each constant c has the type $\mathtt{TyCon}(c)$. We assign selfified types to `true`, `false`, and `unit` (e.g., $\{x{:}\mathtt{Bool} \mid x ==_{\mathtt{Bool}} \mathtt{true}\}$) [20]. Equality is given a similarly reflective type:

$$\mathtt{TyCon}(==_b) \doteq x{:}b \to y{:}b \to \{z{:}\mathtt{Bool} \mid z ==_{\mathtt{Bool}} (x ==_b y)\}.$$

Our system can be extended with any constant $c \in [\![\mathtt{TyCon}(c)]\!]$.

3.2 Static Semantics of λ^{RE}

Next, we define the static semantics of λ^{RE} as given by typing judgements (§3.2.1) and a binary logical relation (§3.2.2).

$$
\begin{array}{rcl}
\llbracket\{x{:}b \mid r\}\rrbracket & \doteq & \{e \mid \ e \hookrightarrow^* v \land \vdash_B e :: b \land r[e/x] \hookrightarrow^* \texttt{true}\} \\
\llbracket x{:}\tau_x \to \tau\rrbracket & \doteq & \{e \mid \forall e_x \in \llbracket\tau_x\rrbracket.\ e\ e_x \in \llbracket\tau[e_x/x]\rrbracket\} \\
\llbracket\mathsf{PEq}_b\ \{e_l\}\ \{e_r\}\rrbracket & \doteq & \{e \mid \ \vdash_B e :: \mathsf{PBEq}_b \land e \hookrightarrow^* \mathsf{bEq}_b\ e_l\ e_r\ e_{pf} \land e_l ==_b e_r \hookrightarrow^* \texttt{true}\} \\
\left\llbracket\mathsf{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}\right\rrbracket & \doteq & \{e \mid \ \vdash_B e :: \mathsf{PBEq}_{\lfloor x:\tau_x\to\tau\rfloor} \land e \hookrightarrow^* \mathsf{xEq\_}\ e_l\ e_r\ e_{pf} \\
 & & \land\ e_l, e_r \in \llbracket x{:}\tau_x \to \tau\rrbracket \land \forall e_x \in \llbracket\tau_x\rrbracket.e_{pf}\ e_x \in \left\llbracket\mathsf{PEq}_{\tau[e_x/x]}\ \{e_l\ e_x\}\ \{e_r\ e_x\}\right\rrbracket\}
\end{array}
$$

Figure 3. Semantic typing: a unary syntactic logical relation interprets types.

3.2.1 Typing of λ^{RE}

Figure 4 defines three mutually recursive judgements:

Typing Checking: $\Gamma \vdash e :: \tau$ when e has type τ in Γ.
Well formedness: $\Gamma \vdash \tau$ when τ is well formed in Γ.
Subtyping: $\Gamma \vdash \tau_l \preceq \tau_r$ when τ_l is a subtype of τ_r in Γ.

Type Checking rules are mostly standard [11, 20, 22]; the interest lies in equality (TEqBase, TEqFun).

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, there must be *invariant 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 $\{\_{:}() \mid l ==_b r\}$. While we allow selfified types (rule TSelf), our formal model leaves it to the programmer to give strong, meaningful types that prove equality. In Liquid Haskell, type inference [22] automatically derives such strong types.

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 use invariant 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.

Well Formedness is uneventful: refinements should be booleans (WFBase); functions are treated in the usual way (WFFun); and the propositional equality $\mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}$ is well formed when the expressions e_l and e_r are typed at the index τ, which is also well formed (WFEq).

Subtyping of basic types reduces to set inclusion on the interpretation of these types (SBase, and Figure 3). Concretely, for all closing substitutions (CEmp, CSub) the interpretation of the left hand side type should be a subset of the right hand side type. The rule SFun implements the usual (dependent) function subtyping. Finally, SEq reduces subtyping of equality types to subtyping of the type indexes, while the expressions to be equated remain unchanged. Even though covariant treatment of the type index would suffice for our metatheory, we treat the type index bivariantly to be consistent with the implementation (§4) where the GADT encoding of PEq is bivariant. Our subtyping rule allows equality proofs between functions with convertible types (§5.2).

3.2.2 Equivalence Logical Relation for λ^{RE}

We characterize equivalence with a term-model binary logical, lifting relations on closed values and expressions to an open relation (Figure 5). Instead of directly substituting in type indices, all three relations use *pending substitutions* δ, which map variables to pairs of equivalent values.

Closed Values and Expressions The relation $v_1 \sim v_2{::}\tau;\delta$ states that the values v_1 and v_2 are related under the type τ with pending substitutions δ. The relation 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.

For refinement types $\{x{:}b \mid r\}$, related values must be the same constant c. Further, this constant should actually be a b-constant and it should actually satisfy the refinement r, i.e., substituting c for x in r should evaluate to `true` under either pending substitution (δ_1 or δ_2). Two values of function type are equivalent when applying them to equivalent arguments yield equivalent results. Since we have dependent types, we record the arguments in the pending substitution for later substitution in the codomain. Two proofs of equality are equivalent when the two equated expressions are equivalent in the logical relation at type-index τ. 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, checking equivalent between $\delta_1 \cdot e_l$ and $\delta_2 \cdot e_r$. Following the proof irrelevance notion of refinement typing, the equivalence of equality proofs does not relate the proof terms—in fact, it doesn't even *inspect* the proofs v_1 and v_2.

Two closed expressions e_1 and e_2 are equivalent on type τ with equivalence environment δ, written $e_1 \sim e_2{::}\tau;\delta$, *iff* they respectively evaluate to equivalent values v_1 and v_2.

Open Expressions A pending substitution δ satisfies a typing environment Γ when its bindings are related pairs of values. 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, $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2{::}\tau;\delta$ holds. The expressions e_1 and e_2 and the type τ might refer to variables in the environment Γ. We use δ to close the expressions eagerly, while we close the type lazily: we apply δ in the refinement and equality cases of the closed value equivalence relation.

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

$$\frac{\Gamma \vdash e :: \tau \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e :: \tau'}\textsc{TSub} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau}\textsc{TVar} \qquad \frac{\Gamma \vdash e :: \{z{:}b \mid r\}}{\Gamma \vdash e :: \{z{:}b \mid z ==_b e\}}\textsc{TSelf} \qquad \frac{}{\Gamma \vdash c :: \mathsf{TyCon}(c)}\textsc{TCon} \qquad \frac{\Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash e :: \tau}{\Gamma \vdash \lambda x{:}\tau_x.\ e :: x{:}\tau_x \to \tau}\textsc{TLam}$$

$$\frac{\Gamma \vdash e_x :: \tau_x \quad \Gamma \vdash e :: x{:}\tau_x \to \tau}{\Gamma \vdash e\ e_x :: \tau[e_x/x]}\textsc{TApp} \qquad \frac{\Gamma \vdash e_l :: \tau_l \quad \Gamma \vdash \tau_l \preceq \{x{:}b \mid \mathtt{true}\} \quad \Gamma \vdash e_r :: \tau_r \quad \Gamma \vdash \tau_r \preceq \{x{:}b \mid \mathtt{true}\} \quad \Gamma, l : \tau_l, r : \tau_r \vdash e :: \{x{:}() \mid l ==_b r\}}{\Gamma \vdash \mathsf{bEq}_b\ e_l\ e_r\ e :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}}\textsc{TEqBase}$$

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

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

$$\frac{\lfloor \Gamma \rfloor, x : b \vdash_B r :: \mathtt{Bool}}{\Gamma \vdash \{x{:}b \mid r\}}\textsc{WFBase} \qquad \frac{\Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau}\textsc{WFFun} \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash e_l :: \tau \quad \Gamma \vdash e_r :: \tau}{\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}}\textsc{WFEq} \qquad \frac{}{\vdash \emptyset}\textsc{WFEmp} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}\textsc{WFBind}$$

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

$$\frac{\forall \theta \in [\![\Gamma]\!],\ [\![\theta \cdot \{x{:}b \mid r\}]\!] \subseteq [\![\theta \cdot \{x'{:}b \mid r'\}]\!]}{\Gamma \vdash \{x{:}b \mid r\} \preceq \{x'{:}b \mid r'\}}\textsc{SBase} \qquad \frac{\Gamma \vdash \tau_x' \preceq \tau_x \quad \Gamma, x : \tau_x' \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau_x' \to \tau'}\textsc{SFun} \qquad \frac{\Gamma \vdash \tau \preceq \tau' \quad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\} \preceq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}}\textsc{SEq}$$

Semantic typing and closing substitutions $\boxed{\theta \in [\![\Gamma]\!]}$ $\boxed{\Gamma \models e \in \tau}$

$$\frac{}{\emptyset \in [\![\emptyset]\!]}\textsc{CEmp} \qquad \frac{v \in [\![\tau]\!] \quad \theta \in [\![\Gamma[v/x]]\!]}{x \mapsto v, \theta \in [\![x : \tau, \Gamma]\!]}\textsc{CSub} \qquad \Gamma \models e \in \tau \Leftrightarrow \forall \theta \in [\![\Gamma]\!],\ \theta \cdot e \in [\![\theta \cdot \tau]\!]$$

Figure 4. Typing of λ^{RE}.

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

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

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

$$e_1 \sim e_2{::}\tau; \delta \doteq e_1 \hookrightarrow^* v_1,\ e_2 \hookrightarrow^* v_2,\ v_1 \sim v_2{::}\tau; \delta$$

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 5. Definition of equivalence logical relation.

3.3 Metaproperties: PEq is an Equivalence Relation

Finally, we show various metaproperties of λ^{RE}. Theorem 3.1 proves soundness of syntactic typing with respect to semantic typing. Theorem 3.2 proves that propositional equality implies equivalence in the term model. Theorems 3.3 and 3.4 prove that both the equivalence relation and propositional equality define equivalences i.e., satisfy the three equality axioms. All the proofs are in supplementary material [29].

λ^{RE} is semantically sound: syntactically well typed programs are also semantically well typed.

Theorem 3.1 (Typing is Sound)**.** *If* $\Gamma \vdash e :: \tau$*, then* $\Gamma \models e \in \tau$.

The proof goes by induction on the derivation tree. Our system could not be proved sound using purely syntactic techniques, like progress and preservation [40]: SBase needs to quantify over all closing substitutions, and purely syntactic approaches flirt with non-monotonicity (but see [42]).

Theorem 3.2 (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., if $\Gamma \vdash e :: \tau$ then $\Gamma \vdash e \sim e :: \tau$, which is proved in turn by induction on the typing derivation.

Theorem 3.3 (The logical relation is an Equivalence)**.** $\Gamma \vdash e_1 \sim e_2 :: \tau$ *is reflexive, symmetric, and transitive.*

Reflexivity is essentially the fundamental property. The other proofs proceed by structural induction on the type τ. Transitivity requires reflexivity on e_2, so we assume that $\Gamma \vdash e_2 :: \tau$.

Theorem 3.4 (PEq is an Equivalence)**.** $\mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ *is reflexive, symmetric, and transitive on equable types. That is, for all* τ *that contain only basic refined types and functions:*

- *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* $\Gamma \vdash v_{23} :: \mathsf{PEq}_\tau\ \{e_2\}\ \{e_3\}$, *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 us to generalize the inductive hypothesis to generate appropriate τ_l and τ_r for the PEq proofs.

4 Implementation: a GADT for Typed Propositional Equality

First (§ 4.1), we define the AEq typeclass as axiomatized equality for base types. Next (§ 4.2), we define the PEq GADT, as propositional equality for base and function types. Refinements on the GADT enforce the typing rules of our formal model (§3) while the metatheory is established in Liquid Haskell itself (§4.3). Finally (§ 4.4), we discuss how AEq and PEq interact with Haskell's and SMT's equalities.

4.1 The AEq typeclass, for axiomatized equality

We use refinements in typeclasses [14] to define AEq as a typeclass that contains the (operational) equality method ≡ and three methods that encode the equality laws.

```
{-@ class AEq a where
     (≡)    :: x:a → y:a → Bool
     reflP  :: x:a → {x ≡ x}
     symmP  :: x:a → y:a → { x ≡ y ⇒ y ≡ x }
     transP :: x:a → y:a → z:a
            → { (x ≡ y && y ≡ z) ⇒ x ≡ z } @-}
```

To define an instance of AEq one has to define the method (≡) and provide explicit proofs that it is reflexive, symmetric, and transitive (reflP, symmP, and transP resp.); thus ≡ is, by construction, an equality.

4.2 The PBEq GADT and its PEq Refinement

In Figure 6, we use AEq to define our type-indexed propositional equality PEq a {e1} {e2} in three steps: (1) structure (à la λ^{RE}) as a GADT, (2) definition of the refined type PEq, and (3) proof construction via a refinement of the GADT.

First, we define the structure of our proofs of equality as PBEq, an unrefined, i.e., Haskell, GADT (Figure 6, (1)). The plain GADT 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 ways to prove our propositional equality, each corresponding to a constructor of PBEq: using an AEq instance (constructor BEq); using funext (constructor XEq); and by congruence closure (constructor CEq).

Next, we define the refinement type PEq to be our propositional equality (Figure 6, (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. 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.

```
-- (1) Plain GADT
data PBEq :: * → * where
     BEq  :: AEq a ⇒ a → a → () → PBEq a
     XEq  :: (a → b) → (a → b) → (a → PEq b)
          → PBEq (a → b)
     CEq  :: a → a → PBEq a → (a → b) → PBEq b

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

-- (3) Type refinement of the GADT
{-@ data PBEq  :: * → *  where
     BEq :: AEq a ⇒ x:a → y:a → {v:() | x ≡ y}
        → PEq a {x} {y}
     XEq :: f:(a → b) → g:(a → b)
        → (x:a → PEq b {f x} {g x})
        → PEq (a → b) {f} {g}
     CEq :: x:a → y:a → PEq a {x} {y}
        → ctx:(a → b) → PEq b {ctx x} {ctx y} @-}
```

Figure 6. Implementation of the propositional equality PEq as a refinement of Haskell's GADT PBEq.

Finally, we refine the type constructors of PBEq to axiomatize the uninterpreted (≍) and generate proofs of PEq (Figure 6, (3)). Each constructor of PBEq 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 funext axiom. Given functions f and g of type a → b, a proof of equality via extensionality also needs an 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! CEq implements congruence closure (§ 4.3): for x and y of type a that are equal—i.e., PEq a {x} {y}—and an arbitrary context with an a-shaped hole (ctx :: a → b), filling the context with x and y yields equal results, i.e., PEq b {ctx x} {ctx y}.

4.3 Equivalence Properties and Classy Induction

Interestingly, we prove metaproperties of the actual implementation of PEq —reflexivity, symmetry, and transitivity (as in Theorem 3.4)—within Liquid Haskell itself.

Just as our paper metatheory, our proofs in Liquid Haskell go by induction on types. But "induction" in Liquid Haskell means writing a recursive function, which necessarily has a single, fixed type. We want a Liquid Haskell theorem refl :: x:a → PEq a {x} {x} that corresponds to Theorem 3.4

(a), but the proof goes by induction on the type a, which is not a thing an ordinary Haskell function could do.[2]

The essence of our proofs is a folklore method we name *classy induction* (see §6 for the history). To prove a theorem using classy induction on the `PEq` GADT, one must: (1) define a typeclass with a method whose refined type corresponds 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. Below we present the proof of reflexivity (the proofs of symmetry and transitivity follow the same pattern and can be found in supplementary material [29]).

```
-- (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))
```

For (1), the typeclass `Reflexivity` simply states the desired theorem type, `refl :: x:a → PEq a {x} {x}`. For (2), given an `AEq a` instance, `BEq` and the `reflP` method are combined to define the `refl` method. To define such a general instance, we enabled the GHC extensions `FlexibleInstances` and `UndecidableInstances`. For (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!

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 a `refl` for our chosen type. Just as our paper metatheory works only for a fixed model, our `refl` proofs only work for types where the codomain bottoms out with an `AEq` instance.

Congruence Closure Congruence closure is proved typically by induction on expressions, i.e., following the cases of the definition of the logical relation. While classy induction allows us to perform induction on types to prove metaproperties within the language, we have no way to perform

[2] A variety of GHC extensions allow case analysis on types (e.g., type families and generics), but unfortunately, are not supported by Liquid Haskell.

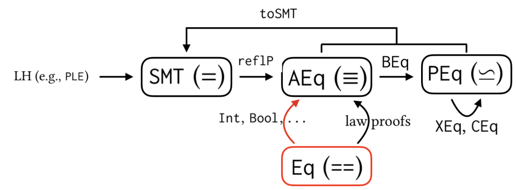

Figure 7. The four different equalities and their interactions.

induction on terms in Liquid Haskell (Coq can; see discussion of Sozeau's work in §6). Instead, we axiomatize congruence closure with the `CEq` data constructor and use it in our proofs about function equalities (e.g., the `map` function in §5.3)

4.4 Interaction of the different equalities.

Figure 7 presents the four equalities that are present in our system: SMT equality (=), the class `AEq` method (≡) (§ 4.1), the GADT `PEq` (§ 4.2), and the Haskell's `Eq` method (==).

Interactions of Equalities SMT equalities are internally generated by Liquid Haskell using the reflection and PLE tactic of [34] (see § 5.1). An $e_1 \equiv e_2$ equality can be generated with three ways: 1/ Given an SMT equality $e_1 = e_2$ and the reflexivity `reflP` method, i.e., calling `reflP` e_1 to prove $e_1 \equiv e_1$, the proof of $e_1 \equiv e_2$ is generated. 2/ Our system provides `AEq` instances for the primitive Haskell types using the Haskell equality that we *assume* satisfies the equality laws, e.g., the `instance AEq Int` is provided. 3/ Finally, using refinements in typeclasses [14] one can explicitly define instances of `AEq`, following or not the Haskell equality definitions.

Data constructors generate `PEq` proofs: `BEq` combined with an `AEq` term and `XEq` or `CEq` combined with `PEq` terms.

Finally, we define a mechanism to convert `PEq` into an SMT equality. This conversion is useful when (see §5.5) we want to derive an SMT equality $f\ e = g\ e$ from a function equality `PEq (a → b) {f} {g}`. The derivation requires that the domain `b` admits axiomatized equality. 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}
```

Non-interaction Liquid Haskell, by default and *unsoundly* maps Haskell's `(==)` to SMT equality. To avoid this build-in unsoundness in our implementation and case studies, we do not direclty use Haskell's equality.

Satisfication of Equality Axioms Comparing the four equalities we note that `AEq` comes with explicit proof terms as the three methods that capture the equality axioms and in `PEq` we proved the equality axioms using classy induction. For the SMT equality, we trust that the implementation of the underlying solver satisfies the axioms, while Haskell's equality has no way to enforce the equality axioms.

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

Figure 8. Reasoning about list reversal.

Computable Equalities As a final comparison of the equalities, we summarize that the `Eq` and `AEq` classes define the computational equalities, `(==)` and `(≡)` respectively. On the contrary, the `PEq` equality only contains proof terms, while the SMT equality only lives inside the refinements.

5 Case Studies

We demonstrate our propositional equality in six case studies. We start by moving from first-order equalities to equalities between functions (`reverse`, §5.1). Next, we show how `PEq`'s type indices reason about refined domains and dependent ranges of functions (`succ`, §5.2). Proofs about higher-order functions exhibit the contextual equivalence axiom (`map`, §5.3). Then, we see that `PEq` plays well with multi-argument functions (`foldl`, §5.4). Next, we present how a `PEq` proof can speedup code (`spec`, §5.5). Finally, we present a bigger case study that proves the monad laws for reader monads (§5.6). In [29] we also prove the monoid laws for endofunctions.

5.1 Reverse: from First- to Higher-Order Equality

Consider three candidate definitions of the list-reverse function (Figure 8, top): a 'fast' one in accumulator-passing style, a 'slow' one in direct style, and a 'bad' one that is the identity.

First-Order Proofs Figure 8 proves, quite easily, a theorem relating the two list reversals. The final theorem `reverseEq` is a corollary of a `lemma` and `rightId`, which shows that `[]` is a right identity for list append, `(++)`. The `lemma` is 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 [34].

Higher-Order Proofs Plain SMT equality isn't enough to prove that `fast` and `slow` are themselves equal. We need functional extensionality: the `XEq` constructor of the `PEq` GADT.

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

The inner `reversePf` shows `fast xs` is propositionally equal to `slow xs` for all `xs`:

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

There are several different styles to construct such a proof.

Style 1: Lifting First-Order Proofs The first order equality proof `reverseEq` lifts directly into propositional equality, using the `BEq` constructor and the reflexivity property of `AEq`.

```
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 that, via the `reflP` call, is turned into axiomatized equality, imposing an `AEq` constraint.

Style 2: Inductive Proofs Alternatively, inductive proofs can be directly performed in the propositional setting, eliminating the `AEq` constraint. To give a sense of the inductive propositional proofs, we translate `lemma` into `lemmaP`:

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

The proof goes by induction and uses the `Reflexivity` and `Transitivity` properties of `PEq` encoded as typeclasses (§4.3) along with `assocP` and `rightIdP`, the propositional versions of `assoc` and `rightId`. These typeclass constraints propagate to the `reverseHO` proof, via `reversePf2`.

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

Style 3: Combinations One can combine the easy first order inductive proofs with the typeclass-encoded properties. For instance below, `refl` sets up the propositional context; `lemma` and `rightId` complete the proof.

```
reversePf3 :: (Reflexivity [a])
           ⇒ xs:[a] → PEq [a] {fast xs} {slow xs}
reversePf3 xs = refl (fast xs)
              ? lemma xs [] ? rightId (slow xs)
```

Bad Proofs Soundly, we could not use any of these styles to generate a (bad) proof of neither `PEq ([a] → [a]) {fast} {bad}` nor `PEq ([a] → [a]) {slow} {bad}`.

5.2 Succ: Refined Domains and Dependent Ranges

Our propositional equality `PEq` naturally reasons about functions with refined domains and dependent ranges. For example, consider the functions `sNat` and `sInt` that respectively return the successor of a natural and integer number.

```
sNat, sInt :: Int → Int
sNat x = if x >= 0 then x + 1 else 0
sInt x = x + 1
```

First, we prove that the two functions are equal on the domain of natural numbers:

```
type Nat = {x:Int | 0 <= x }
natDom :: PEq (Nat → Int) {sInt} {sNat}
natDom = XEq sInt sNat $ \x →
         BEq (sInt x) (sNat x) (reflP (sInt x))
```

We can also reason about how each function's domain affects its range. For example, we can prove that both functions take `Nat` inputs to `Nat` outputs.

```
natRng :: PEq (Nat → Nat) {sInt} {sNat}
natRng = XEq sInt sNat $ \x →
         BEq (sInt x) (sNat x) (reflP (sInt x))
```

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

```
type SNat x = {v:Nat | v = x + 1}
depRng :: PEq (x:Nat → SNat {x}) {sInt} {sNat}
depRng = XEq sInt sNat $ \x →
         BEq (sInt x) (sNat x) (reflP (sInt x))
```

Equalities Rejected by Our System Liquid Haskell correctly rejects various wrong proofs of equality between the functions `sInt` and `sNat`. We highlight three:

```
badDom  :: PEq (  Int → Int)          {sInt} {sNat}
badRng  :: PEq (  Nat → {v:Int|v<0})  {sInt} {sNat}
badDRng :: PEq (x:Nat → {v:Int|v=x+2}) {sInt} {sNat}
```

`badDom` expresses that `sInt` and `sNat` are equal for any `Int` input, which is wrong, e.g., `sInt (-2)` yields `-1`, but `sNat (-2)` yields 0. Correctly constrained to natural domains, `badRng` specifies a negative range (wrong) while `badDRng` specifies that the result is increased by 2 (also wrong). Our system rejects both with a refinement type error.

5.3 Map: Putting Equality in Context

Our propositional equality can be used in higher order settings: we prove that if `f` and `g` are propositionally equal, then `map f` and `map g` are also equal. Our proofs use the congruence closure equality constructor/axiom `CEq`.

Equivalence on the Last Argument Direct application of `CEq` ports a proof of equality to the last 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 = CEq f g pf map
```

Equivalence on an Arbitrary Argument To show that `map f xs` and `map g xs` are equal for all `xs`, we use `CEq` 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 = CEq f g pf (flipMap xs)
                ? 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 relies on `CEq` with the flipped context and needs to know that `map f xs = flipMap xs f`. This fact cannot be inferred by Liquid Haskell, in the higher order setting of the proof, and is explicitly provided by the `fMapEq` calls.

Proof Reuse in Context Finally, we use the `natDom` proof (§5.2) to show how existing proofs can be reused with `map`.

```
client :: xs:[Nat]
       → PEq [Int] {map sInt xs} {map sNat xs}
client = mapEq sInt sNat natDom

clientP :: PEq ([Nat] → [Int]) {map sInt} {map sNat}
clientP = mapEqP sInt sNat natDom
```

`client` proves that `map sInt xs` is equivalent to `map sNat xs` for each list `xs` of natural numbers, while `clientP` proves that the partially applied functions `map sInt` and `map sNat` are equivalent on the domain of lists of natural numbers.

5.4 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 lifted 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 Proving Style 2 of §5.1, i.e., the typeclass-encoded properties (see supplementary material [29]).

5.5 Spec: Function Equality for Program Efficiency

Function equality can be used to soundly optimize runtimes. 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 (a → Bool) {fastSpec} {spec}
       → a → Maybe a
client pf x =
  if fastSpec x
     ? toSMT (fastSpec x) (spec x)
             (EqCtx fastSpec spec pf (\x f → f x))
  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.

This example showcases how our propositional equality 1/ co-exists with practical features of refinement types, e.g., path sensitivity, and 2/ is used to optimize executable code.

5.6 Monad Laws for Reader Monads

A *reader* is a function with a fixed domain `r`, i.e., the partially applied type `Reader r` (Figure 9, top left). We use our propositional equality to prove the monad laws for readers.

The monad instance for the reader type is defined using function composition (Figure 9, top). We also define Kleisli composition of monads as a convenience for specifying the monad. We prove that readers are in fact monads, i.e., their operations satisfy the monad laws (Figure 9, middle). Along the way, we also prove that they satisfy the functor and applicative laws in supplementary material [29]. The reader monad laws are expressed as refinement type specifications using `PEq`. We prove the left and right identities in three steps: `XEq` to take an input of type `a`; `refl` to generate an equality proof; and `(=~=)` to give unfolding hints to the SMT solver. The `(=~=)` operator is defined as `_ =~= y = y` (unrefined) and is used to unfold the function definitions of its arguments and thus, give proof hints to the SMT solver.

Proof by Associativity and Error Locality The use of `(=~=)` in proofs by reflexivity is not checking intermediate equational steps. So, the proof either succeeds or fails without explanation. To address this problem, during proof construction, we employed transitivity. For instance, in the `monadAssociativity` proof, our goal is to construct the proof `PEq _ {el} {er}`. To do so, we pick an intermediate term `em`; we might 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
```

The `refl el` proof will produce a type error; replacing that proof with an appropriate `trans` completes the proof (Figure 9, bottom). Such an approach to writing proofs in this 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).

Our reader proofs use the `Reflexivity` and `Transitivity` typeclasses to ensure that readers are monads whatever the return type may be. Having generic monad laws is critical: readers are typically used to compose functions that take configuration information, but such functions usually have other arguments, too! For example, an interpreter might run `readFile >>= parse >>= eval`, where `readFile :: Config → String` and `parse :: String → Config → Expr` and `eval :: Expr → Config → Value`. With our associativity proof, we can rewrite the above to `readFile >>= (kleisli parse eval)` even though `parse` and `eval` are higher-order terms. Doing so could, in theory, trigger inlining/fusion rules that would combine the parser and the interpreter.

6 Related Work

Functional Extensionality and Subtyping with an SMT Solver F*also uses a type-indexed `funext` axiom after having run into similar unsoundness issues [8]. Their extensionality axiom makes a more roundabout connection with SMT: function equality uses ==, a proof-irrelevant, propositional

Monad Instance for Readers

```
type Reader r a = r → a

pure :: a → Reader r a
pure a _r = a

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

Reader Monad Laws

```
monadLeftIdentity :: Reflexivity b ⇒ a:a → f:(a → Reader r b) → PEq (Reader r b) {bind (pure a) f} {f a}
monadRightIdentity :: Reflexivity a ⇒ m:(Reader r a) → PEq (Reader r a) {bind m pure} {m}
monadAssociativity :: (Reflexivity c, Transitivity 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)}
```

Identity Proofs By Reflexivity

```
monadLeftIdentity a f =
  XEq (bind (pure a) f) (f a) $ \r →
    refl (bind (pure a) f r) ?
     (bind (pure a) f r =~= f (pure a r) r
                        =~= f a r *** QED)

monadRightIdentity m =
  XEq (bind m pure) m $ \r →
    refl (bind m pure r) ?
     (bind m pure r =~= pure (m r) r
                    =~= m r *** QED)
```

Associativity Proof By Transitivity and Reflexivity

```
  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 9. Case study: Reader Monad Proofs.

Leibniz equality. They assume that their Leibniz equality coincides with SMT equality. Liquid Haskell can't just copy F*: there are no dependent, inductive type definitions, nor a dedicated notion of propositions. Our `PEq` GADT approximates F*'s approach, with different compromises.

Dafny's SMT encoding axiomatizes extensionality for data, but not for functions [13]. Function equality is utterable but neither provable nor disprovable in their encoding into Z3.

Ou et al. [20] introduce *selfification*, which assigns singleton types using equality (as in our TSelf rule). SAGE assigns selfified types to *all* variables, implying equality on functions [12]. Dminor avoids function equality by not having first-class functions [2].

Extensionality in Dependent Type Theories Functional extensionality (`funext`) has a rich history of study. Martin-Löf type theory comes in a decidable, intensional flavor (ITT) [15] as well as an undecidable, extensional one (ETT) [16]. NuPRL implements ETT [4], while Coq and Agda implement ITT [2008, 2020]. Lean's quotient-based reasoning can *prove* `funext` [7]. Extensionality axioms are independent of the rules of ITT; `funext` is a common axiom, but is not consistent in every model of type theory [35]. Hofmann [10] shows that ETT is a conservative but less computational extension of ITT with `funext` and UIP. Pfenning [21] and Altenkirch and McBride [1] try to reconcile ITT and ETT.

Dependent type theories often care about equalities between equalities, with axioms like UIP (all identity proofs are the same), K (all identity proofs are `refl`), and univalence (identity proofs are isomorphisms, and so not the same). If we had a way to prove equalities between equalities, we could add UIP. Since our propositional equality isn't exactly Leibniz equality, axiom K would be harder to encode.

Zombie's type theory uses an adaptation of a congruence closure algorithm to automatically reason about equality [25]. Zombie can do some reasoning about equalities on functions but cannot show equalities based on bound variables. Zombie is careful to omit a λ-congruence rule, which could be used to prove `funext`, "which is not compatible with [their] 'very heterogeneous' treatment of equality" [Ibid., §9].

Cubical type theory offer alternatives to our propositional equality [28]. Such approaches may play better with F*'s approach using dependent, inductive types than the 'flatter' approach we used for Liquid Haskell. Univalent systems like cubical type theory get `funext` 'for free'—that is, for the price of the univalence axiom or of cubical foundations.

Classy Induction: Inductive Proofs Using Typeclasses We used 'classy induction' to prove metaproperties of `PEq` inside Liquid Haskell (§4.3), using ad-hoc polymorphism and general instances to generate proofs that 'cover' all types.

We did not *invent* classy induction—it is a folklore technique that we named. We have seen five independent uses of "classy induction" in the literature [3, 6, 9, 31, 38].

Any typeclass system that accommodates ad-hoc polymorphism and a notion of proof can use classy induction.

Sozeau [26] 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 [27] introduce typeclasses to Coq; their system can do induction by typeclasses, but they do not demonstrate the idea in the paper. Earlier work on typeclasses focused on overloading [17, 18, 37], with no notion of classy induction even in settings with proofs [39].

7 Conclusion

Refinement type checking uses SMT solvers to support automated and assisted reasoning about programs. Functional programs make frequent use of higher-order functions and higher-order representations with data. Our type-indexed propositional equality avoids unsoundness in the naïve framing of `funext`; we reason about function equality in both our formal model and its Liquid Haskell implementation. Several case studies demonstrate the range and power of our work.

Acknowledgments

We thank Conal Elliott for his help in exposing the inadequacy of the naïve function extensionality encoding. Stephanie Weirich, Éric Tanter, and Nicolas Tabareau offered valuable insights into the folklore of classy induction.

References

[1] Thorsten Altenkirch and Conor McBride. 2006. Towards observational type theory. Unpublished manuscript.

[2] Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. 2012. Semantic subtyping with an SMT solver. *J. Funct. Program.* 22, 1 (2012), 31–105. https://doi.org/10.1017/S0956796812000032

[3] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*. Paris, France, 182 – 194. https://doi.org/10.1145/3018610.3018620

[4] 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

[5] Robert L Constable and Scott Fraser Smith. 1987. *Partial objects in constructive type theory.* Technical Report. Cornell University.

[6] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of dependent interoperability. *J. Funct. Program.* 28 (2018), e9. https://doi.org/10.1017/S0956796818000011

[7] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[8] 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.

[9] Louis-Julien Guillemette and Stefan Monnier. 2008. A Type-Preserving Compiler in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP ’08)*. Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/1411204.1411218

[10] Martin Hofmann. 1996. Conservativity of equality reflection over intensional type theory. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–164.

[11] Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article Article 6 (Feb. 2010), 34 pages. https://doi.org/10.1145/1667048.1667051

[12] 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*.

[13] K. Rustan M. Leino. 2012. Developing verified programs with Dafny. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology, HILT ’12, December 2-6, 2012, Boston, Massachusetts, USA*, Ben Brosgol, Jeff Boleng, and S. Tucker Taft (Eds.). ACM, 9–10. https://doi.org/10.1145/2402676.2402682

[14] 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 *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*.

[15] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium ’73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73 – 118. https://doi.org/10.1016/S0049-237X(08)71945-1

[16] Per Martin-Löf. 1984. *Intuitionistic Type Theory.* Bibliopolis. https://books.google.com/books?id=_D0ZAQAAIAAJ As recorded by Giovanni Sambin.

[17] Tobias Nipkow and Christian Prehofer. 1993. Type Checking Type Classes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’93)*. Association for Computing Machinery, New York, NY, USA, 409–418. https://doi.org/10.1145/158511.158698

[18] Tobias Nipkow and Gregor Snelting. 1991. Type Classes and Overloading Resolution via Order-Sorted Unification. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, Berlin, Heidelberg, 1–14.

[19] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP’08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

[20] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France (IFIP)*, Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell (Eds.), Vol. 155. Kluwer/Springer, 437–450. https://doi.org/10.1007/1-4020-8141-3_34

[21] F. Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. 221–230.

[22] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[23] John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on*

Software Engineering 24, 9 (1998), 709–720.

[24] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. Syst.* 39, 1, Article 3 (Feb. 2017), 36 pages. https://doi.org/10.1145/2994594

[25] Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to Congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 369–382. https://doi.org/10.1145/2676726.2676974

[26] Matthieu Sozeau. 2008. *Un environnement pour la programmation avec types dépendants.* Ph.D. Dissertation. Université Paris 11, Orsay, France.

[27] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293.

[28] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2019. Cubical Syntax for Reflection-Free Extensional Equality. *CoRR* abs/1904.08562 (2019). arXiv:1904.08562 http://arxiv.org/abs/1904.08562

[29] supplementary material. 2020. Supplementary Material for Functional Extensionality for Refinement Types.

[30] 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 *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[31] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2019. The Marriage of Univalence and Parametricity. *arXiv e-prints*, Article arXiv:1909.05027 (Sept. 2019), arXiv:1909.05027 pages. arXiv:cs.PL/1909.05027

[32] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/zenodo.3744225

[33] 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 *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

[34] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2018. Refinement reflection: complete verification with SMT. *PACMPL* 2, POPL (2018), 53:1–53:31. https://doi.org/10.1145/3158141

[35] Tamara von Glehn. 2014. *Polynomials and Models of Type Theory.* Ph.D. Dissertation. Magdalene College, University of Cambridge.

[36] Philip Wadler. 2015. Propositions as Types. *Commun. ACM* 58, 12 (Nov. 2015), 75–84. https://doi.org/10.1145/2699407

[37] P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[38] Stephanie Weirich. 2017. The Influence of Dependent Types (Keynote). In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3009837.3009923

[39] Markus Wenzel. 1997. Type classes and overloading in higher-order logic. In *Theorem Proving in Higher Order Logics*, Elsa L. Gunter and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–322.

[40] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115 (1994), 38–94. Issue 1.

[41] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 249–257.

[42] Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. 2020. Blame tracking at higher fidelity. In *Workshop on Gradual Typing (WGT)*.