Functional Extensionality for Refinement Types

Niki Vazou
IMDEA

Michael Greenberg
Pomona College

WG 2.8, 2021

Functional Extensionality for Refinement Types

"Two functions are equal,
if their values are equal at every argument."

$$\text{funExt} :: \forall a\ b.\ f{:}(a \to b) \to g{:}(a \to b) \to (x{:}a \to \underline{\{f\ x = g\ x\}}) \to \{f = g\}$$

short for
$\{v{:}()\ |\ f\ x = g\ x\}$

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

incrNat :: Int → Int                              incrInt :: Int → Int
incrNat x = if 0 ≤ x then 0 else x + 1            incrInt x = x + 1

type Nat = {v:Int | 0 ≤ v}
incrEq :: x:Nat → {incrNat x = incrInt x}
incrEq _ = ()

incrFEq :: () → { incrNat = incrInt }
incrFEq _ = funExt incrNat incrInt incrEq

incrEqMap :: xs:[Nat] → {map incrNat xs = map incrInt xs}
incrEqMap _ = incrFEq ()
```

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

Problem 1: Result type forgets the domain

```
incrEq :: x:Nat → {incrNat x = incrInt x}
incrEq _ = ()

incrFEq :: () → { incrNat = incrInt }
incrFEq _ = funExt incrNat incrInt incrEq

incrEqMap :: xs:[Nat] → {map incrNat xs = map incrInt xs}
incrEqMap _ = incrFEq ()
```

$$\text{funExt} :: \forall a\ b.\ f{:}(a \to b) \to g{:}(a \to b) \to (x{:}a \to \{f\ x = g\ x\}) \to \{f = g\}$$

Problem 1: Result type forgets the domain

```
incrEq :: x:Nat → {incrNat x = incrInt x}
incrEq _ = ()
incrFEq :: () → { incrNat = incrInt }
incrFEq _ = funExt incrNat incrInt incrEq
incrEqMap :: xs:[Nat] → {map incrNat xs = map incrInt xs}
incrEqMap _ = incrFEq ()
```

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

Problem 1: Result type forgets the domain

```
incrEq :: x:Nat → {incrNat x = incrInt x}
incrEq _ = ()

incrFEq :: () → { incrNat = incrInt }
incrFEq _ = funExt incrNat incrInt incrEq

incrEqMap :: xs:[Int] → {map incrNat xs = map incrInt xs}
incrEqMap _ = incrFEq ()

incrBad :: { incrNat (-2) = incrInt (-2) } —  i.e., 0 = -1
incrBad _ = incrFEq ()
```

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

Problem 2: Domain only appears positively

At instantiation we can pick **any** domain, e.g.,

```
a := {v:Int | false}
```

Under the empty domain, we can trivially prove anything

```
incrFEq :: () → { incrInt = plus2 }
incrFEq _ = funExt incrInt plus2
                 (\_ → ()) — x:{v:Int | false) → {incrInt x = plus2 x}
eqBad :: () → { incrInt 0 = plus2 0 }    — i.e., 1 = 2
eqBad _ = incrFEq ()
```

Problem 1: Result type forgets the domain

Problem 2: Domain only appears positively

Solution: Type-indexed equality

$$\text{PEq}\ a\ \{e_l\}\ \{e_r\}$$

"e_l is equal to e_r on type a"

Problem 1: Result type forgets the domain

Problem 2: Domain only appears positively

Solution: Type-indexed equality

$$\mathrm{PEq}\ a\ \{e_l\}\ \{e_r\}$$

$$\begin{aligned} & f : (a \to b) \to g : (a \to b) \\ & \to (x : a \to \mathrm{PEq}\ b\ \{f\ x\}\ \{g\ x\}) \\ & \to \mathrm{PEq}\ (a \to b)\ \{f\}\ \{g\} \end{aligned}$$

Domain appears

negatively in the result!

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq ::  f:(a → b) → g:(a → b)
        → (x:a → PEq b {f x} {g x})
        → PEq (a → b) {f} {g}
```

e.g.,

```
XEq incrNat incrInt ...
:: PEq (Nat → Int) {incrNat} {incrInt}
```

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq ::  f:(a → b) → g:(a → b)
        → (x:a → PEq b {f x} {g x})
        → PEq (a → b) {f} {g}

  BEq :: x:a → y:a → { x = y }
       → PEq a {x} {y}
```

e.g.,

```
XEq incrNat incrInt ...
:: PEq (Nat → Int) {incrNat} {incrInt}
```

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq ::  f:(a → b) → g:(a → b)
        → (x:a → PEq b {f x} {g x})
        → PEq (a → b) {f} {g}

  BEq :: x:a → y:a → { x = y }
       → PEq a {x} {y}
```

What if a is function???

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq ::  f:(a → b) → g:(a → b)
        → (x:a → PEq b {f x} {g x})
        → PEq (a → b) {f} {g}
  BEq :: AEq a ⇒ x:a → y:a → { x ≡ y }
       → PEq a {x} {y}
```

What if a is function???

AEq: axiomatised equality

AEq: axiomatised equality

$$
\begin{array}{l}
\textbf{class}\ \mathrm{AEq}\ a\ \textbf{where} \\
\quad (\equiv) :: a \rightarrow a \rightarrow \mathrm{Bool} \\
\quad \mathrm{reflP} :: x{:}a \rightarrow \{x \equiv x\} \\
\quad \mathrm{symmP} :: x{:}a \rightarrow y{:}a \rightarrow \{x \equiv y \Rightarrow y \equiv x\} \\
\quad \mathrm{transP} :: x{:}a \rightarrow y{:}a \rightarrow z{:}a \rightarrow \{x \equiv y \wedge y \equiv z \Rightarrow x \equiv z\}
\end{array}
$$

e.g., $x :: \mathrm{Nat} \vdash \mathrm{reflP}\ (\mathrm{incrInt}\ x) :: \{\mathrm{incrInt}\ x \equiv \mathrm{incrInt}\ x\}$

$$:: \{\mathrm{incrNat}\ x \equiv \mathrm{incrInt}\ x\}$$

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq ::   f:(a → b) → g:(a → b)
         → (x:a → PEq b {f x} {g x})
         → PEq (a → b) {f} {g}

  BEq :: AEq a ⇒ x:a → y:a → { x ≡ y }
        → PEq a {x} {y}
```

e.g.,

```
XEq incrNat incrInt $ \x →
      BEq (incrNat x) (incrInt x) $ reflP (incrInt x)
:: PEq (Nat → Int) {incrNat} {incrInt}
```

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq :: f:(a → b) → g:(a → b)
         → (x:a → PEq b {f x} {g x})
         → PEq (a → b) {f} {g}

  BEq :: AEq a ⇒ x:a → y:a → { x ≡ y }
        → PEq a {x} {y}

  CEq :: x:a → y:a → ctx:(a → b) → PEq a {x} {y}
        → PEq b {ctx x} {ctx y}
```

e.g.,

```
XEq incrNat incrInt $ \x →
     BEq (incrNat x) (incrInt x) $ reflP (incrInt x)
:: PEq (Nat → Int) {incrNat} {incrInt}
```

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq :: f:(a → b) → g:(a → b)
         → (x:a → PEq b {f x} {g x})
         → PEq (a → b) {f} {g}

  BEq :: AEq a ⇒ x:a → y:a → { x ≡ y }
       → PEq a {x} {y}

  CEq :: x:a → y:a → ctx:(a → b) → PEq a {x} {y}
       → PEq b {ctx x} {ctx y}
```

e.g.,

```
CEq incrNat incrInt map $
    XEq incrNat incrInt $ \x →
          BEq (incrNat x) (incrInt x) $ reflP (incrInt x)
:: PEq ([Nat] → [Int]) {map incrNat} {map incrInt}
```

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq :: f:(a → b) → g:(a → b)
         → (x:a → PEq b {f x} {g x})
         → PEq (a → b) {f} {g}

  BEq :: AEq a ⇒ x:a → y:a → { x ≡ y }
       → PEq a {x} {y}

  CEq :: x:a → y:a → ctx:(a → b) → PEq a {x} {y}
       → PEq b {ctx x} {ctx y}
```

e.g.,

```
CEq incrNat incrInt map $
    XEq incrNat incrInt $ \x →
        BEq (incrNat x) (incrInt x) $ reflP (incrInt x)
:: PEq ([Nat] → [Int]) {map incrNat} {map incrInt}
```

PEq: Type-indexed equality

```
data PEq :: * → * where
  XEq :: f:(a → b) → g:(a → b)
        → (x:a → PEq b {f x} {g x})
        → PEq (a → b) {f} {g}

  BEq :: AEq a ⇒ x:a → y:a → { x ≡ y }
       → PEq a {x} {y}

  CEq :: x:a → y:a → ctx:(a → b) → PEq a {x} {y}
       → PEq b {ctx x} {ctx y}

\xs → CEq (map incrNat) (map incrInt) (\f → f xs) $
         CEq incrNat incrInt map $
            XEq incrNat incrInt $ \x →
                BEq (incrNat x) (incrInt x) $ reflP (incrInt x)
:: xs:[Nat] → PEq ([Int]) {map incrNat xs} {map incrInt xs}
```

Can I convert PEq back to SMT equality?

```
\xs → CEq (map incrNat) (map incrInt) (\f → f xs) $
          CEq incrNat incrInt map $
             XEq incrNat incrInt $ \x →
                 BEq (incrNat x) (incrInt x) $ reflP (incrInt x)
:: xs:[Nat] → PEq ([Int]) {map incrNat xs} {map incrInt xs}
```

Can I convert PEq back to SMT equality?

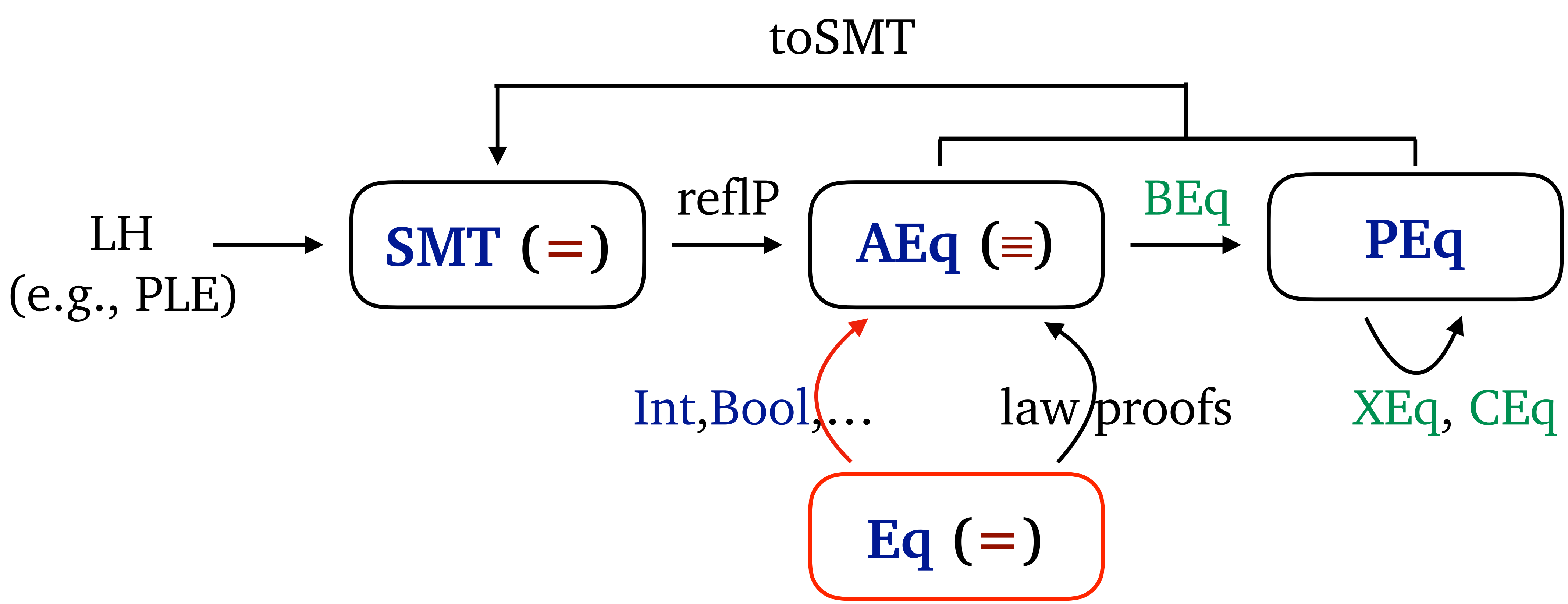
We defined a class for this conversion

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

which comes with a default instance

```
assume instance AEq a ⇒ SMTEq a where
  toSMT _ _ _ = ()
```

Can I convert PEq back to SMT equality?



* the more equalities, the more consistent the system

Is PEq expressive?

We used PEq to prove

Monoid laws for Endofuctors (64 LoC)

Monad & Monoid laws for Reader (243 LoC)

```
monadLeftIdentity :: a:a → f:(a → Reader r b) → PEq (Reader r b) {bind (pure a) f} {f a}
monadRightIdentity :: m:(Reader r a) → PEq (Reader r a) {bind m pure} {m}
monadAssociativity :: 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)}
```

Is PEq expressive?

We used PEq to prove
Monoid laws for Endofuctors (64 LoC)
Monad & Monoid laws for Reader (243 LoC)

Is PEq an equivalence?

Paper-and-pencil proofs for toy core calculus
Equivalence properties proofs within Liquid Haskell

PEq is reflexive by “classy induction”

```
-- (1) Property Definition by 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))
```

Functional Extensionality for Refinement Types

Problem: Naive axiom forgets the domain type
which leads to inconsistencies, i.e., proves false

Solution: Type Indexed Equality
which is verbose, but expressive and consistent

Draft: https://nikivazou.github.io/static/equality-PLDI21.pdf **Thanks!**