Functional Extensionality for Refinement Types

NIKI VAZOU, IMDEA Software Institute, Spain
MICHAEL GREENBERG, Pomona College, USA

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, refinement type systems with semantic subtyping treat naive phrasings of functional extensionality inconsistently, leading to *unsoundness*. We demonstrate this unsoundness 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 unsoundness while still proving useful equalities at higher types; we demonstrate its use in several case studies. We validate `PEq` by building a small 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'.

1 INTRODUCTION

Refinement types have been extensively used to reason about functional programs [Constable and Smith 1987; Rondon et al. 2008; Rushby et al. 1998; Swamy et al. 2016; Xi and Pfenning 1998]. 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. [2018a] prove function optimizations correct by specifying equalities between fully applied functions. 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 built on 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 naive `funext` axiom proves equalities between unequal functions.

Our first contribution is to expose why a naive encoding of `unfext` 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` does not encode function equality, since it is not reasoning about equality on the domains of the functions. 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

[1] See https://github.com/FStarLang/FStar/issues/1542 for F*'s initial, wrong encoding and §7 for F*'s different solution. We explain the situation in Liquid Haskell in §2.

Authors' addresses: Niki Vazou, niki.vazou@imdea.org, IMDEA Software Institute, Madrid, Spain; Michael Greenberg, michael.greenberg@pomona.edu, Pomona College, Claremont, CA, USA.

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 system that allows for semantic subtyping along with refinement polymorphism, i.e., refinements inferred from constraints [Rondon et al. 2008]. 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 Haskell's GADTs and Liquid Haskell's refinement types. 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 induction on the structure of the type index—we applied a heretofore-unnamed folklore proof methodology, which we dub *classy induction* (§3.3).

Our third contribution is to use `PEq` to prove equalities between functions (§4). 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`), and showcase how higher-order, propositional equalities can co-exist with and speedup executable code. We also provide a more substantial case study, proving the 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 (§5). We prove that λ^{RE} is sound and that propositional equality implies equality in a term model of equivalence (§6).

2 FUNCTIONAL EXTENSIONALITY IS INCONSISTENT IN REFINEMENT TYPES

Functional extensionality states that two functions are equal, if their values are equal at every argument: $\forall f, g : A \to B, \forall x \in A, f(x) = g(x) \Rightarrow f = g$. 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. For example, Agda defines functional extensionality as below in the standard library:

```
Extensionality : (a b : Level) → Set _ -- Axiom.Extensionality.Propositional
Extensionality a b =
  {A : Set a} {B : A → Set b} {f g : (x : A) → B x} → (∀ x → f x ≡ g x) → f ≡ g
```

Having seen `funext`'s success in other dependently typed languages, we naively admitted the `funext` axiom below in Liquid Haskell:

```
{-@ assume funext :: ∀ a b. f:(a→b) → g:(a→b) → (x:a → {f x = g x}) → {f = g} @-}
funext :: (a → b) → (a → b) → (a → ()) → ()
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 Agda's `Extensionality` axiom. But looks can be deceiving: in Liquid Haskell, we can use `funext` to prove `false`. Why?

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

```
incrInt, incrPos :: Integer → Integer
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:Integer | 0 < n } @-}
{-@ incrSamePos :: n:Pos → {incrPos n = incrInt n} @-}
incrSamePos :: Integer → ()
incrSamePos _n = ()
```

We can use `funext` to show that `incrPos` and `incrInt` are equal, using our proof `incrSamePos` on the domain of positive numbers.

```
{-@ incrExt :: {incrPos = incrInt} @-}
incrExt :: ()
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 :: [Integer] → ()
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. We don't give a full example here, but `funext` is particularly critical when trying to equate structures that are themselves higher order, like difference lists or streams.

Unfortunately, `incrExt` makes it *too* easy to prove equivalences... our system is inconsistent! Here's a proof that 0 is equal to -4:

```
{-@ inconsistencyI :: {incrPos (-5) = incrInt (-5)} @-} -- 0 = -4
inconsistencyI :: ()
inconsistencyI = incrExt
```

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.

So **`funext` is inconsistent in Liquid Haskell**! The problem is that **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. Ouch!

For example, `plus2` below defines a function that increases its input by `2` and is obviously not equal to `incrInt` on any nontrivial domain.

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

Even so, we can use `funext` to prove that `plus2` behaves the same as `incrInt` on the empty domain, i.e., for all inputs `n` that satisfy `false`.

```
{-@ type Empty = {v:Integer | false } @-}
{-@ incrSameEmpty :: n:Empty → {incrInt n = plus2 n} @-}
incrSameEmpty :: Integer → ()
incrSameEmpty _n = ()
```

Now `incrSameEmpty` provides enough evidence for `funext` to show that `incrInt` equals `plus2`, which we use to prove another egregious inconsistency.

```
{-@ incrPlus2Ext :: {incrInt = plus2} @-}
incrPlus2Ext ::  ()
incrPlus2Ext = funext incrInt plus2 incrSameEmpty

{-@ inconsistencyII :: {incrInt 0 = plus2 0} @-} -- 1 = 2
inconsistencyII :: ()
inconsistencyII = incrPlus2Ext
```

Liquid Haskell isn't like most other dependent type theories: we can't just admit `funext` as phrased. But we still want to prove equalities between higher-order values! What can we do?

2.1 Refined, Type-Indexed, Extensional, Propositional Equality

If we're going to reason using functional extensionality in Liquid Haskell, we'll need to be careful to remember the type at which we show the functions produce equal results. What domains are involved when we use functional extensionality?

To prove two functions 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 `Integer` ($\mathcal{D}_f$ = $\mathcal{D}_g$ = `Integer`), 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 will implicitly set the domain on which the functions are equal to the most general one where both functions can be called ($\mathcal{D}_e$ = `Integer`).

Our `funext` encoding naively imposes no real constraints between these domains. In fact, `funext` only requires that $\mathcal{D}_f$, $\mathcal{D}_g$, and $\mathcal{D}_p$ are supertypes of the empty domain (§5), which trivially holds for all types, leaving $\mathcal{D}_e$ underconstrained.

To be consistent, we need a functional extensionality axiom that (1) captures the domain of function equality $\mathcal{D}_e$ explicitly, (2) requires that the domain of the equality, $\mathcal{D}_e$, is a subtype of the domain of the proof, $\mathcal{D}_p$, which should be a subtype of the functions domains, $\mathcal{D}_f$ and $\mathcal{D}_g$, and (3) ensures that the resulting equality between functions is only used on 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 $\mathtt{e}_l$ and $\mathtt{e}_r$ are propositionally equal and of type `a`. We carefully crafted `PEq`'s definition as a refined GADT (§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. `PEq`'s constructor axiomatizing functional extensionality keeps careful track of types:

```
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 return type (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 $\mathcal{D}_e$. (See §5 for more detail on type checking.)

2. Generating Function Equalities. The `XEq` case of `PEq` generates equalities at function types using functional extensionality. Liquid Haskell will check the domains appropriately: it won't prove equality between functions at an inappropriate domain.

Returning to our concrete example of `incrPos` and `incrInt`, we can use `XEq` to find these functions equal on the domain `Pos`:

```
{-@ incrExtGood :: PEq (Pos → Integer) {incrPos} {incrInt} @-}
incrExtGood :: PEq (Integer → Integer)
incrExtGood = XEq incrPos incrInt incrEq
```

`XEq` checks that the domains of the functions `incrPos` and `incrInt` are supertypes of `Pos`, i.e., `Pos <: Integer`. Further it checks that the domain of the proof `incrEq` is 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 → Integer` type index; each alternative is also possible or impossible to prove. (See §3 for more on how `incrEq` can be defined.)

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

The first two alternatives, `n:Pos` and `n:Integer`, will be accepted by `XEq`, since both `Pos` and `Integer` 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:Empty` is trivial, but `XEq` rejects it, because `Empty` 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 `PEq`'s `XEq` constructor ensures that the right domains are checked and tracked for functional extensionality, we have a constructor for ensuring these equalities are used appropriately. The constructor `CEq` characterizes equality as valid in all contexts, i.e., if `x` and `y` are equal, they can be substituted in any context `ctx` and the results `ctx x` and `ctx y` will be equal:

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

It is easy to use `CEq` to apply functional equalities in higher order contexts. For example, we can prove that `map incrPos` equals `map incrInt`:

```
{-@ incrMapProp :: PEq ([Pos] → [Integer]) {map incrPos} {map incrInt} @-}
incrMapProp :: PEq ([Integer] → [Integer])
incrMapProp = CEq incrPos incrInt incrExtGood (map)
```

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 `app` 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`:

```
{-@ app :: f:(a → b) → g:(a → b) → PEq (a → b) {f} {g}
        → x:a → PEq b {f x} {g x} @-}
app :: (a → b) → (a → b) → PEq (a → b) → a → PEq b
app f g eq x = CEq f g eq (flip x)

flip x f = f x
```

The `app` 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 apply the functional equality `incrExtGood` to a bad and a good input.

```
{-@ badFO ::PEq Integer {incrPos 0} {incrInt 0} @-}
badFO = app incrPos incrInt incrExtGood 0  -- REJECTED

{-@ goodFO :: x:{Integer | 42 < x } → PEq Integer {incrPos x} {incrNat x} @-}
goodFO x = app incrPos incrInt incrExtGood x -- ACCEPTED
```

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 `Integer`. 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.)

2.2 Why Isn't funext Inconsistent in Agda?

At the beginning of §2, we present Agda's `Extensionality` axiom, whose return type is `f ≡ g`. Agda's equality appears to lack a type index. Why doesn't Agda also suffer from inconsistency?

Agda's equality only seems to be unindexed. In fact, Agda's built-in equality is the standard, type-indexed Leibniz equality used in most dependent type theories (omitting `Level` polymorphism):

```
data _≡_ {A : Set} (x : A) : A → Set a where
  refl : x ≡ x
```

The curly braces around the type index `A` marks it as *implicit*, i.e., to be inferred. If we were to explicitly give implicit arguments by wrapping them in curly braces, Agda's extensionality axiom returns `(_≡_) {a→b} f g`.

Our `XEq` axiom recovers the type indexing in Agda's equivalence that's missing in our original `funext` encoding. Of course, (Liquid) Haskell's lack of implicit type indices makes reasoning about function equalities verbose. On the other hand, Liquid Haskell's subtyping can reinterpret functions at many domains (see §4.2). In Agda, however, it is much more complex to reinterpret functions and to generate heterogeneous equality relating `incrInt` and `incrPos` only on positive inputs.

3 PEQ: A LIBRARY AND GADT FOR 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` GADT [Cheney and Hinze 2003; Xi et al. 2003] (§3.2). Refinements on the GADT enforce the typing rules of our formal model (§6), but we 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 with by axiomatizing equality that can be ported to SMT: such an equality should be an equivalence relation that implies SMT equality. We use refinements on typeclasses [Liu et al. 2020] to define a typeclass `AEq`, which contains the (operational) 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}
    symmP  :: x:a → y:a → { x ≡ y ⇒ y ≡ x }
```

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

Fig. 1. Implementation of the propositional equality PEq as a refinement of Haskell's GADT PBEq.

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

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. Finally, we require the proof `smtP` that captures that (≡) implies equality provable by SMT (e.g., structural equality).[2]

3.2 The PBEq GADT and its PEq Refinement

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

First, we define the structure of our proofs of equality as PBEq, an unrefined, i.e., Haskell, GADT (Figure 1, (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 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. 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.

[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.

Finally, we refine the type constructors of `PBEq` to axiomatize the uninterpreted (≐) and generate proofs of `PEq` (Figure 1, (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 (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`! `CEq` implements congruence closure: 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}`.

Design Alternatives. The first design choice we made was to define `PEq` as a GADT and not an axiomatized opaque type. While there's no reason to pattern match on `PEq` terms, there's also no harm in it. A GADT provides a clean interface on how `PEq` can be generated: it collects all the axioms as data contructors and prevents the user from arbitrarily adding new constructors. The second choice we made was to define the type `PEq` using a fresh uninterpreted equality symbol (Figure 1, (2)) instead of SMT equality. Again, we made this decision to ensure that all `PEq` terms are constructed via the constructors and not implicit SMT automation. The final choice we made was to define the base case using the `AEq` constraints. We considered two alternatives:

```
BEq ::        x:a → y:a → {v:() | x = y } → PEq a {x} {y} -- alternative I
BEq :: Eq a ⇒ x:a → y:a → {v:() | x = y } → PEq a {x} {y} -- alternative II
```

We rejected the first to ensure that the base case does not include functions (which don't generally have `Eq` instances) and to support our metatheory (§3.3). We rejected the second to exclude user-defined `Eq` instances that do not correspond to SMT equality (since in §3.4 we define a machanism to turn `PEq` to SMT equalities).

Example: Having seen `AEq` and the `BEq` case of `PEq`, we can define the `incrEq` function from §2:

```
{-@ incrEq :: x:Pos → PEq Integer {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 `BEq` proof term `{incrPos x ≡ incrInt x}`.

3.3 Equivalence Properties and Classy Induction

We can prove metaproperties of the actual implementation of `PEq`—reflexivity, symmetry, and transitivity—within Liquid Haskell itself.

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. 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 §7 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. Here's the proof for reflexivity.

[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.

```
-- (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 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, but both also make use congruence closure. The full proofs can be found in supplementary material [2021]. Here is the inductive case from symmetry:

```
instance Symmetry b ⇒ Symmetry (a → b) where
-- sym :: l:(a→b) → r:(a→b) → PEq (a→b) {l} {r} → PEq (a→b) {r} {l}
   sym l r pf = XEq r l $ \a → sym (l a) (r a) (CEq l r pf ($ a) ? ($ a l) ? ($ a r)))
```

Here `l` and `r` are functions of type `a → b` and we know that `l ≐ r`; we must prove that `r ≐ l`. We do so using: (a) `XEq` for extensionality, letting `a` of type `a` be given; (b) `sym (l a) (r a)` as the IH on the codomain `b` on (c) `CEq` for congruence closure on `l ≐ r` in the context `($ a)`. The last step is the most interesting: if `l` is equal to `r`, then plugging them into the same context yields equal results; as our context, we pick `($ a)`, i.e., `\f → f a`, showing that `l a ≐ r a`; the IH on the codomain `b` yields `r a ≐ l a`, and extensionality shows that `r ≐ l`, as desired. The operator `?`, defined as `x ? p = x`, asks Liquid Haskell to encode 'p' into the SMT solver to help prove 'x'. Our use of `?` unfolds the definitions `$ a l` and `$ a r` to help `CEq`.

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 GADT `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 [Barrett et al. 2010] permits

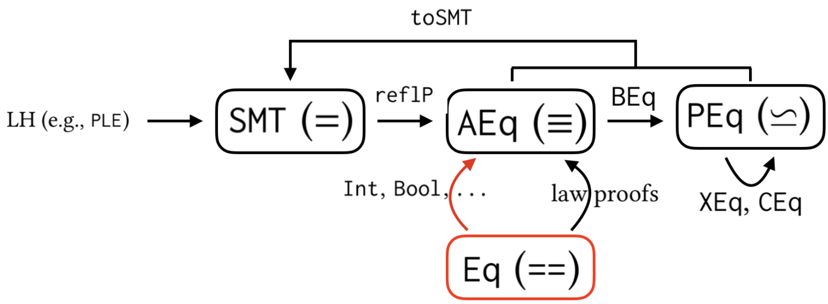

Fig. 2. The four different equalities and their interactions. Haskell equality is in red to highlight its potential for unsoundness.

the equality symbol on functions but does not specify its behavior. Implementations vary. CVC4 allows for functional extensionality and higher-order reasoning [Barbosa et al. 2019]. 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. [2018b] (see also §4.1). An $e_1 \equiv e_2$ equality can be generated one of three ways: (1) If SMT can prove an SMT 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 [Liu et al. 2020] one can explicitly define instances of `AEq`, which may or may not coincide with Haskell `Eq` instances.

Constructors generate `PEq` proofs, bottoming out at `AEq`: `BEq` combined with an `AEq` term and `XEq` or `CEq` 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 SMT 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}
```

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

Equivalence Relation Axioms. Each of the four equalities has a different relationship to th equivalence relation axioms (reflexivity, symmetry, transitivity). `AEq` comes with explicit proof methods that capture the axioms. For `PEq`, we prove the equality axioms 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.

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
```

Fig. 3. Reasoning about list reversal.

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 CASE STUDIES

We demonstrate our propositional equality in seven case studies. 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 (`succ`, §4.2). Proofs about higher-order functions demonstrate the contextual equivalence axiom (`map`, §4.3). Then, we see that `PEq` plays well with multi-argument functions (`foldl`, §4.4). Next, we present how a `PEq` proof can speedup code (`spec`, §4.5). Finally, we present two bigger case studies that prove the monoid laws for endofunctions (§4.6) and the monad laws for reader monads (§4.7). Complete code is available in the [supplementary material 2021].

4.1 Reverse: from First- to Higher-Order Equality

Consider three candidate definitions of the list-reverse function (Figure 3, 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. The `reverseEq` theorem neatly relates the two list reversals (Figure 3). 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 [Vazou et al. 2018b].

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 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}
```

There are several different ways 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, 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. To give a sense of what these proofs are like, 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 (§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 :: (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. Here `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. 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 :: Integer → Integer
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:Integer | 0 < x }
posDom :: PEq (Pos → Integer) {incrInt} {incrPos}
posDom = XEq incrInt incrPos $ \x → 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:Integer | 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))
```

Equalities Rejected by Our System. Liquid Haskell correctly rejects various wrong, (non-)proofs of equality between the functions `incrInt` and `incrPos`. We highlight three:

```
badDom  :: PEq (  Integer → Integer)              {incrInt} {incrPos}
badCod  :: PEq (  Pos     → {v:Integer | v < 0})   {incrInt} {incrPos}
badDCod :: PEq (x:Pos     → {v:Integer | v = x+2}) {incrInt} {incrPos}
```

`badDom` expresses that `incrInt` and `incrPos` are equal for any `Integer` input, which is wrong, e.g., `incrInt (-2)` yields `-1`, but `incrPos (-2)` yields 0. Correctly constrained to positive domains, `badCod` specifies a negative codomain (wrong) while `badDCod` specifies that the result is increased by 2 (also wrong). Our system rejects all three with a refinement type error.

4.3 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`. 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 `fMapEq` calls.

Proof Reuse in Context. Finally, we use the `posDom` proof (§4.2) to show how existing proofs can be reused with `map`.

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

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

`client` proves that `map incrInt xs` is equivalent 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 equivalent on the domain of 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 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 the second, typeclass-oriented style of §4.1, (see supplementary material [2021] 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 (a → Bool) {fastSpec} {spec} → a → Maybe a
client pf x =
  if fastSpec x ? toSMT (fastSpec x) (spec x) (CEq fastSpec spec pf (\x f → f x))
    then Just (critical x)
    else Nothing
```

Monoid Instance for Endofunctions

```
type Endo a = a → a                             _ =~= y = y

mempty :: Endo a                                mappend :: Endo a → Endo a → Endo a
mempty a = a                                    mappend f g a = f (g a) -- a/k/a (<>)
```

Endofunction Monoid Laws

```
mLeftIdentity  :: (Reflexivity a, Transitivity a)
               ⇒ x:Endo a → PEq (Endo a) {mappend mempty x} {x}
mRightIdentity :: (Reflexivity a, Transitivity a)
               ⇒ x:Endo a → PEq (Endo a) {x} {mappend x mempty}
mAssociativity :: (Reflexivity a, Transitivity a)
               ⇒ x:(Endo a) → y:(Endo a) → z:(Endo a)
               → PEq (Endo a) {mappend (mappend x y) z} {mappend x (mappend y z)}
```

Proofs By Reflexivity and Transitivity

```
mLeftIdentity x = XEq (mappend mempty x) x $ \a →
      refl (mappend mempty x a) ? (mappend mempty x a =~= mempty (x a) =~= x a *** QED)

mRightIdentity x = XEq x (mappend x mempty) $ \a →
      refl (x a) ? (x a =~= x (mempty a) =~= mappend x mempty a *** QED)

mAssociativity x y z =
    XEq (mappend (mappend x y) z) (mappend x (mappend y z)) $ \a →
      refl (mappend (mappend x y) z a) ?
           (mappend (mappend x y) z a =~= (mappend x y) (z a) =~= x (y (z a))
                                      =~= x (mappend y z a)
                                      =~= mappend x (mappend y z) a *** QED)
```

Fig. 4. Case study: Endofunction Monoid Proofs.

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 Monoid Laws for Endofunctions

Endofunctions form a law-abiding monoid. A function f is an *endofunction* when its domain and codomain types are the same. A *monoid* is an algebraic structure comprising an identity element (`mempty`) and an associative operation (`mappend`). For the monoid of endofunctions, `mempty` is the identity function and `mappend` is function composition (Figure 4; top).

To be a monoid, `mempty` must really be an identity with respect to `mappend` (`mLeftIdentity` and `mRightIdentity`) and `mappend` must really be associative (`mAssociativity`) (Figure 4; middle).

Proving the monoid laws for endofunctions demands functional extensionality (Figure 4; bottom). For example, consider the proof that `mempty` is a left identity for `mappend`, i.e., `mappend mempty x = x`. To prove this equation between *functions*, we can't use SMT equality. With functional

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
```

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))
```

Fig. 5. Case study: Reader Monad Proofs.

extensionality, each proof reduces to three parts: `XEq` to take an input of type `a`; `refl`on the left-hand side of the equation, to generate an equality proof; and `(=~=)` to give unfolding hints to the SMT solver. The `(=~=)` operator is defined as `_ =~= y = y`, and it is unrefined, i.e., it is not checking equality of its arguments.

The `Reflexivity` constraints on the theorems make our proofs general in the underlying type `a`: endofunctions on the type `a` form a monoid whether `a` admits SMT equality or if it's a complex higher-order type (whose ultimate result admits equality). Haskell's typeclass resolution ensures that an appropriate `refl` method will be constructed whatever type `a` happens to be.

4.7 Monad Laws for Reader Monads

A *reader* is a function with a fixed domain `r`, i.e., the partially applied type `Reader r` (Figure 5, top left). Readers form a monad and their composition is a useful way of defining and composing functions that take some fixed information, like command-line arguments or configuration files. Our propositional equality can prove the monad laws for readers.

The monad instance for the reader type is defined using function composition (Figure 5, 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 5, bottom). We also prove that they satisfy the functor and applicative laws in supplementary material [2021]. The reader monad laws are expressed as refinement type specifications using `PEq`. We prove the left and right identities following the pattern of §4.6, i.e., `XEq`, followed by reflexivity with `(=~=)` for function unfolding (Figure 5, middle). We use transitivity to conduct the more complicated proof of associativity (Figure 5, bottom).

Proof by Associativity and Error Locality. As noted earlier, 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` to connect `el` and `em` via `eml` completes the `monadAssociativity` proof (Figure 5, 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).

Our reader proofs use the `Reflexivity` and `Transitivity` typeclasses to ensure that readers are monads whatever the return type `a` may be (with the type of 'read' values fixed to `r`). 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 generic proof of associativity, we can rewrite the above to `readFile >>= (kleisli parse eval)` even though `parse` and `eval` are higher-order terms without `Eq` instances. Doing so could, in theory, trigger inlining/fusion rules that would combine the parser and the interpreter.

5 TYPE CHECKING XEQ: DID WE GET IT RIGHT?

We've seen that `XEq` is effective at proving equalities between functions (§4) and we've argued that we avoid the inconsistency with `funext`. Things *seem* to work in Liquid Haskell. But: Why do things go so wrong with `funext`? Does `XEq` really avoid `funext`'s issues? We give a schematic example showing why Liquid Haskell works with `XEq` consistently but works with `funext` inconsistently. (We give a detailed, formal model of our propositional equality in §6.)

Suppose we have two functions `h` and `k`, defined on domains d_{h} and d_{k} and codomains r_{h} and r_{k}, respectively. Let's also assume we have some `lemma` that proves, for all x in some domain d_p, we have an equality $e_l \simeq e_r$, where e_l and e_r are arbitrary expressions of type $\{\mathtt{v}{:}\beta \mid r_p\}$.

$$\begin{array}{l}
\mathtt{h} :: \mathtt{x}{:}\{\alpha \mid d_{\mathsf{h}}\} \to \{\mathtt{v}{:}\beta \mid r_{\mathsf{h}}\} \\
\mathtt{k} :: \mathtt{x}{:}\{\alpha \mid d_{\mathsf{k}}\} \to \{\mathtt{v}{:}\beta \mid r_{\mathsf{k}}\} \\
\mathtt{lemma} :: \mathtt{x}{:}\{\alpha \mid d_p\} \to \mathtt{PEq}\ \{\mathtt{v}{:}\beta \mid r_p\}\ \{e_l\}\ \{e_r\}
\end{array}$$

We can pass these along to our `XEq` constructor (of §3) to form a proof that `h` equals `k` on some domain d_e:

Typing Environmennt

$$\begin{array}{lll}\Gamma & \doteq & \{\ \ \mathsf{XEq} : \forall \alpha\beta . f : (\alpha \to \beta) \to g : (\alpha \to \beta) \to (x : \alpha \to \mathsf{PEq}\ \beta\{f\ x\}\{g\ x\}) \to \mathsf{PEq}\ (\alpha \to \beta)\{f\}\{g\} \\ & , & \ \ \mathsf{h} : x : \{\!\mid d_{\mathsf{h}} \mid\!\} \to \{\!\mid r_{\mathsf{h}} \mid\!\}, \mathsf{k} : x : \{\!\mid d_{\mathsf{k}} \mid\!\} \to \{\!\mid r_{\mathsf{k}} \mid\!\}, \mathtt{lemma} : x : \{\!\mid d_p \mid\!\} \to \mathsf{PEq}\ \{\!\mid r_p \mid\!\}\{e_l\}\{e_r\} \quad \}\end{array}$$

Type Checking

$1.\Gamma \vdash \mathsf{XEq} :: \forall \alpha\beta . f{:}(\alpha \to \beta) \to g{:}(\alpha \to \beta) \to (x{:}\alpha \to \mathsf{PEq}\ \beta\ \{f\ x\}\{g\ x\}) \to \mathsf{PEq}\ (\alpha \to \beta)\{f\}\{g\}$

$2.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\} :: \forall \beta . f{:}(\{\!\mid\kappa_\alpha\mid\!\} \to \beta) \to g{:}(\{\!\mid\kappa_\alpha\mid\!\} \to \beta) \to (x{:}\{\!\mid\kappa_\alpha\mid\!\} \to \mathsf{PEq}\ \beta\ \{f\ x\}\{g\ x\}) \to \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \beta)\{f\}\{g\}$

$3.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\}\ @\{\!\mid\kappa_\beta\mid\!\} :: f{:}(\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\}) \to g{:}(\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\}) \to (x{:}\{\!\mid\kappa_\alpha\mid\!\} \to \mathsf{PEq}\ \{\!\mid\kappa_\beta\mid\!\}\ \{f\ x\}\{g\ x\}) \to \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\})\{f\}\{g\}$

$4.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\}\ @\{\!\mid\kappa_\beta\mid\!\}\ \mathsf{h} :: g : (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\}) \to (x{:}\ \{\!\mid\kappa_\alpha\mid\!\} \to \mathsf{PEq}\ \{\!\mid\kappa_\beta\mid\!\}\{\mathsf{h}\ x\}\{g\ x\}) \to \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\})\{\mathsf{h}\}\{g\}$ $\boxed{\textsc{Sub-H}}$

$5.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\}\ @\{\!\mid\kappa_\beta\mid\!\}\ \mathsf{h}\ \mathsf{k} :: (x : \{\!\mid\kappa_\alpha\mid\!\} \to \mathsf{PEq}\ \{\!\mid\kappa_\beta\mid\!\}\{\mathsf{h}\ x\}\{\mathsf{k}\ x\}) \to \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\})\{\mathsf{h}\}\{\mathsf{k}\}$ $\boxed{\textsc{Sub-K}}$

$6.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\}\ @\{\!\mid\kappa_\beta\mid\!\}\ \mathsf{h}\ \mathsf{k}\ \mathtt{lemma} :: \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\})\{\mathsf{h}\}\{\mathsf{k}\}$ $\boxed{\textsc{Sub-L}}$

$7.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\}\ @\{\!\mid\kappa_\beta\mid\!\}\ \mathsf{h}\ \mathsf{k}\ \mathtt{lemma} :: \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\})\{\mathsf{h}\}\{\mathsf{k}\}$ $\boxed{\textsc{Sub-Sub}}$

$8.\Gamma \vdash \mathsf{XEq}\ @\{\!\mid\kappa_\alpha\mid\!\}\ @\{\!\mid\kappa_\beta\mid\!\}\ \mathsf{h}\ \mathsf{k}\ \mathtt{lemma} :: \mathsf{PEq}\ (\{\!\mid d_e\mid\!\} \to \{\!\mid r_e\mid\!\})\{\mathsf{h}\}\{\mathsf{k}\}$

Subtyping Derivation Leaves

$$\dfrac{\dfrac{i.\ \kappa_\alpha \Rightarrow d_{\mathsf{h}}}{\Gamma \vdash \{\!\mid\kappa_\alpha\mid\!\} \preceq \{\!\mid d_{\mathsf{h}}\mid\!\}} \qquad \dfrac{\kappa_\alpha \Rightarrow r_{\mathsf{h}} \Rightarrow \kappa_\beta}{\Gamma, x : \{\!\mid\kappa_\alpha\mid\!\} \vdash \{\!\mid r_{\mathsf{h}}\mid\!\} \preceq \{\!\mid\kappa_\beta\mid\!\}}}{\Gamma \vdash x : \{\!\mid d_{\mathsf{h}}\mid\!\} \to \{\!\mid r_{\mathsf{h}}\mid\!\} \preceq \{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\}}\textsc{Sub-H} \qquad \dfrac{\dfrac{ii.\ \kappa_\alpha \Rightarrow d_{\mathsf{k}}}{\Gamma \vdash \{\!\mid\kappa_\alpha\mid\!\} \preceq \{\!\mid d_{\mathsf{k}}\mid\!\}} \qquad \dfrac{\kappa_\alpha \Rightarrow r_{\mathsf{k}} \Rightarrow \kappa_\beta}{\Gamma, x : \{\!\mid\kappa_\alpha\mid\!\} \vdash \{\!\mid r_{\mathsf{k}}\mid\!\} \preceq \{\!\mid\kappa_\beta\mid\!\}}}{\Gamma \vdash x : \{\!\mid d_{\mathsf{k}}\mid\!\} \to \{\!\mid r_{\mathsf{k}}\mid\!\} \preceq \{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\}}\textsc{Sub-K}$$

$$\dfrac{\dfrac{iii.\ \kappa_\alpha \Rightarrow d_p}{\Gamma \vdash \{\!\mid\kappa_\alpha\mid\!\} \preceq \{\!\mid d_p\mid\!\}} \qquad \dfrac{\dfrac{\kappa_\alpha \Rightarrow r_p \Rightarrow \kappa_\beta}{\Gamma, x : \{\!\mid\kappa_\alpha\mid\!\} \vdash \{\!\mid r_p\mid\!\} \preceq \{\!\mid\kappa_\beta\mid\!\}} \quad \dfrac{\kappa_\alpha \Rightarrow \kappa_\beta \Rightarrow r_p}{\Gamma, x : \{\!\mid\kappa_\alpha\mid\!\} \vdash \{\!\mid\kappa_\beta\mid\!\} \preceq \{\!\mid r_p\mid\!\}} \quad iv.\ \kappa_\alpha \Rightarrow e_l \simeq e_r \Rightarrow \mathsf{h}\ x \simeq \mathsf{k}\ x}{\Gamma, x : \{\!\mid\kappa_\alpha\mid\!\} \vdash \mathsf{PEq}\ \{\!\mid r_p\mid\!\}\{e_l\}\{e_r\} \preceq \mathsf{PEq}\ \{\!\mid\kappa_\beta\mid\!\}\{\mathsf{h}\ x\}\{\mathsf{k}\ x\}}}{\Gamma \vdash x : \{\!\mid d_p\mid\!\} \to \mathsf{PEq}\ \{\!\mid r_p\mid\!\}\{e_l\}\{e_r\} \preceq x : \{\!\mid\kappa_\alpha\mid\!\} \to \mathsf{PEq}\ \{\!\mid\kappa_\beta\mid\!\}\{\mathsf{h}\ x\}\{\mathsf{k}\ x\}}\textsc{Sub-L}$$

$$\dfrac{\dfrac{\dfrac{vi.\ d_e \Rightarrow \kappa_\alpha}{\Gamma \vdash \{\!\mid d_e\mid\!\} \preceq \{\!\mid\kappa_\alpha\mid\!\}} \quad \dfrac{\kappa_\alpha \Rightarrow \kappa_\beta \Rightarrow r_e}{\Gamma, x : \{\!\mid d_e\mid\!\} \vdash \{\!\mid\kappa_\beta\mid\!\} \preceq \{\!\mid r_e\mid\!\}}}{\Gamma \vdash \{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\} \preceq \{\!\mid d_e\mid\!\} \to \{\!\mid r_e\mid\!\}} \quad \dfrac{\dfrac{v.\ \kappa_\alpha \Rightarrow d_e}{\Gamma \vdash \{\!\mid\kappa_\alpha\mid\!\} \preceq \{\!\mid d_e\mid\!\}} \quad \dfrac{\kappa_\alpha \Rightarrow r_e \Rightarrow \kappa_\beta}{\Gamma, x : \{\!\mid\kappa_\alpha\mid\!\} \vdash \{\!\mid r_e\mid\!\} \preceq \{\!\mid\kappa_\beta\mid\!\}}}{\Gamma \vdash \{\!\mid d_e\mid\!\} \to \{\!\mid r_e\mid\!\} \preceq \{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\}} \quad \mathsf{h} \simeq \mathsf{k} \Rightarrow \mathsf{h} \simeq \mathsf{k}}{\Gamma \vdash \mathsf{PEq}\ (\{\!\mid\kappa_\alpha\mid\!\} \to \{\!\mid\kappa_\beta\mid\!\})\{\mathsf{h}\}\{\mathsf{k}\} \preceq \mathsf{PEq}\ (\{\!\mid d_e\mid\!\} \to \{\!\mid r_e\mid\!\})\{\mathsf{h}\}\{\mathsf{k}\}}\textsc{Sub-Sub}$$

Fig. 6. Type checking XEq h k lemma. For space, we write $\{\!\mid d \mid\!\}$ to mean the refined type $\{v : t \mid d\}$.

```
XEq h k lemma :: PEq ({v:α | d_e} → {v:β | r_e}) {h} {k}
```

When type checking this use of XEq, we need to check that the lemma equates the right expressions (i.e., forall x. $e_l \simeq e_r$ implies $\mathsf{h}\ x \simeq \mathsf{k}\ x$). Critically, type checking must also ensure that the final equality domain (d_e) is stronger than the domains for the functions (d_{h}, d_{k}) and for the lemma (d_p).

Liquid Haskell goes through a complex series of steps to enforce both required checks (Figure 6). We haven't modified Liquid Haskell's typing rules or implementation *at all*; we merely defined PEq in such a way that the existing type checking rules in Liquid Haskell implement the right checks to soundly show extensional equality between functions.

It's easiest to understand how type checking works from top to bottom ("Type Checking", Figure 6). First, we look up XEq's type in the environment (1). Since the XEq is polymorphic, we instantiate the type arguments with the types, $\{v : \alpha \mid \kappa_\alpha\}$ (2) and $\{v : \beta \mid \kappa_\beta\}$ (3). (We write $\{\!\mid\kappa_\alpha\mid\!\}$ as a short for $\{v : \alpha \mid \kappa_\alpha\}$, since we focus on the refinements assuming the Haskell types match.) Here κ_α and κ_β are refinement type variables; type checking will generate constraints on them that liquid type inference will try to resolve [Rondon et al. 2008]. Next we apply each of the arguments: h (4), k (5), and lemma (6). Each application applies standard dependent function application, with consideration for subtyping. That is, each application (a) substitutes the applied argument in the codomain type and (b) checks that the type of the argument is a subtype of the function's domain type. Application leads to the subtyping constraints Sub-H, Sub-K, and Sub-L set off in boxes, resolved below. Now Liquid Haskell has *inferred* a type for the checked expression (7). To conclude the check, it introduces the final subtype constraint Sub-Sub: the inferred type should be a subtype of the type the user specified (8).

The four instances of subtyping during type checking generate 13 logical implications to resolve for the original expression to type check ("Subtyping Derivation Leaves", Figure 6). The six purple implications with Roman numerals place requirements on the domain; we'll ignore the others, which impose less interesting constraints on the functions' codomains. The Sub-H and Sub-K derivations require (via contravariance) that the refinement variable κ_α implies the refinements on the functions' domains, d_h and d_k. Similarly, the derivation Sub-L requires that κ_α implies the proof domain d_p. Since `PEq` is defined as refined type alias (§3), Sub-L also checks that the refinements given imply the top level refinements of `PEq`, i.e., that the result of the lemma is sufficient to show `XEq`'s precondition. The Sub-Sub derivation checks subtyping of two `PEq` types, by treating the type arguments *invariantly*. (We mark covariant implications in red and contravariant implications in blue.) Liquid Haskell treats checks invariantly because `PEq`'s definition uses its type parameter in both positive and negative positions. Sub-Sub will ultimately require that the refinement variable κ_α is *equivalent* to the equality domain d_e.

To sum up, type checking imposes the following six implications as constraints:

$$\begin{array}{llllll} i. & \kappa_\alpha \Rightarrow d_\mathsf{h} & ii. & \kappa_\alpha \Rightarrow d_\mathsf{k} & iii. & \kappa_\alpha \Rightarrow d_p \\ iv. & \kappa_\alpha \Rightarrow e_l \simeq e_r \Rightarrow \mathsf{h}\ x \simeq \mathsf{k}\ x & v. & \kappa_\alpha \Rightarrow d_e & vi. & d_e \Rightarrow \kappa_\alpha \end{array}$$

Implications *v* and *vi* require the refinement variable κ_α to be equivalent to the equality domain d_e. Given that equality, implications *i*–*iii* state that the equality domain d_e should imply the domains of the functions (*i* and *ii*) and lemma (*iii*). Implication *iv* requires that the lemma's domain implies equality of the two functions for each argument `x` that satisfies the domain d_e. All together, these constraints exactly capture the requirements of functional extensionality.

Naive Functional Extensionality with `funext`. When, in §2, we use the non-type-indexed `funext` in Liquid Haskell, the typing derivation looks almost exactly the same, but one critical thing changes: the type-indexed `PEq t {`e_l`} {`e_r`}` is replaced by a refined unit `{v:() | `e_l` = `e_r`}`. This only affects the Sub-L and Sub-Sub derivations, which lose the red and blue parts and become:

$$\dfrac{\dfrac{iii'.\ \kappa_\alpha \Rightarrow d_p}{\Gamma \vdash \{\!|\kappa_\alpha|\!\} \ \preceq\ \{\!|d_p|\!\}} \qquad \dfrac{iv'.\ \kappa_\alpha \Rightarrow e_l = e_r \Rightarrow h\ x = k\ x}{\Gamma, x : \{\!|\kappa_\alpha|\!\} \vdash \{v : () \mid e_l = e_r\} \ \preceq\ \{v : () \mid h\ x = k\ x\}}}{\Gamma \vdash x{:}\{\!|d_p|\!\} \to \{v : () \mid e_l = e_r\} \ \preceq\ x{:}\{\!|\kappa_\alpha|\!\} \to \{v : () \mid h\ x = k\ x\}}\ \textsc{Sub-L-Naive}$$

$$\dfrac{h\ x = k\ x \Rightarrow h\ x = k\ x}{\Gamma \vdash \{v : () \mid h\ x = k\ x\} \ \preceq\ \{v : () \mid h\ x = k\ x\}}\ \textsc{Sub-Sub-Naive}$$

Sub-L-Naive generates the implications iii' and iv' that are essentially the same as before. But, Sub-Sub-Naive won't generate any meaningful checks, because equality is just a unit type. We lost implications v and vi! We are now left with an implication system in which the refinement variable κ_α only appears in the assumptions. Since Liquid Haskell always tries to infer the most specific refinement possible, it will find a very specific refinement for κ_α: `false`! Having inferred `false` for κ_α, the entire use of `funext` trivially holds and can be used on other, nontrivial domains—with inconsistent results.

6 A REFINEMENT CALCULUS WITH BUILT-IN TYPE-INDEXED EQUALITY

Because `funext` is inconsistent in Liquid Haskell (§2), we developed `PEq` to reason consistently about extensional equality, using the GADT `PBEq` and the uninterpreted equality `PEq` (§3). We're able to prove some interesting equalities (§4) and Liquid Haskell's type checking seems to be doing the right thing (§5). But how do we know that our definitions suffice? Formalizing *all* of Liquid Haskell is a challenge, but we can build a model to check the features we use. We formalize a core calculus λ^{RE} with *R*efinement types, semantic subtyping, and type-indexed propositional *E*quality.

$$
\begin{array}{rrcl}
\textit{Constants} & c & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{unit} \mid (==_b) \mid (==_{(c,b)}) \\
\textit{Expressions} & e & ::= & c \mid x \mid e\ e \mid \lambda x{:}\tau.\ e \mid \mathsf{bEq}_b\ e\ e\ e \ \mid\ \mathsf{xEq}_{x:\tau\to\tau}\ e\ e\ e \\
\textit{Values} & v & ::= & c \mid \lambda x{:}\tau.\ e \mid \mathsf{bEq}_b\ e\ e\ v \ \mid\ \mathsf{xEq}_{x:\tau\to\tau}\ e\ e\ v \\
\textit{Refinements} & r & ::= & e \\
\textit{Basic Types} & b & ::= & \mathtt{Bool} \mid () \\
\textit{Types} & \tau & ::= & \{x{:}b \mid r\} \mid x{:}\tau \to \tau \mid \mathsf{PEq}_\tau\ \{e\}\ \{e\} \\
\textit{Typing Environment} & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau \\
\textit{Closing Substitutions} & \theta & ::= & \emptyset \mid \theta, x \mapsto v \\
\textit{Equivalence Environments} & \delta & ::= & \emptyset \mid \delta, (v, v)/x \\
\textit{Evaluation Contexts} & \mathcal{E} & ::= & \bullet \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathsf{bEq}_b\ e\ e\ \mathcal{E} \mid \mathsf{xEq}_{x:\tau\to\tau}\ e\ e\ \mathcal{E}
\end{array}
$$

Reduction $\boxed{e \hookrightarrow e}$

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

Fig. 7. Syntax and Dynamic Semantics of λ^{RE}.

λ^{RE} contains just enough to check the core interactions between refinement types and a type-indexed propositional equality resembling our `PBEq` definition (§6.1). We omit plenty of important features from Liquid Haskell (e.g., algebraic data types): 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 (§6.2), we prove several metatheorems (§6.3). Most importantly, we define a logical relation that characterizes λ^{RE} equivalence and reflects λ^{RE}'s propositional equality. Propositional equivalence in λ^{RE} implies equivalence in the logical relation (Theorem 6.2); both are reflexive, symmetric, and transitive (Theorems 6.3 and 6.4).

6.1 Syntax and Semantics of λ^{RE}

We present λ^{RE}, a core calculus with *R*efinement types and type-indexed *E*quality (Figure 7).

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: 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` constructors of §3; we did not encode congruence closure since it can be proved by induction on expressions, which is impossible in Haskell.

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* $\mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$, 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 $\{x{:}b \mid \mathtt{true}\}$.

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

Fig. 8. Semantic typing: a unary syntactic logical relation interprets types.

We omit polymorphic types to avoid known and resolved metatheoretical problems [Sekiyama et al. 2017]. Yet, xEq equality primitive is defined as a family of operators, one for each refinement function type, capturing the essence of polymorphic function equality.

Environments. The typing environment Γ binds variables to types, the (semantic typing) closing substitution θ binds variables to values, and the (logical relation) pending substitution 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 7, bottom). The semantics are standard with bEq_b and $\mathsf{xEq}_{x:\tau_x\to\tau}$ 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 8; open terms, Figure 9).

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 §B.1). 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 $\mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$ includes all expressions that satisfy the basic typing (PBEq_τ is the unrefined version of $\mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}$) and reduce to a basic equality proof whose first arguments reduce to equal b-constants. Finally, the interpretation of the function equality type $\mathsf{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}$ includes all expressions that satisfy the basic typing (based on the $\lfloor\cdot\rfloor$ operator; §B.1). These expressions reduce to a proof 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 $\mathsf{PEq}_{\tau[e_x/x]}\ \{e_l\ e_x\}\ \{e_r\ e_x\}$. We write these proofs as xEq_, since the type index does not need to be syntactically equal to the index of the type.

Constants. For simplicity, λ^{RE} constants are only the two boolean values, unit, and equality operators for basic types. For each basic type 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 $\mathsf{TyCon}(c)$, as defined below.

$$
\begin{array}{rcl}
\mathsf{TyCon}(\texttt{true}) & \doteq & \{x{:}\mathsf{Bool} \mid x ==_{\mathsf{Bool}} \texttt{true}\} \\
\mathsf{TyCon}(\texttt{false}) & \doteq & \{x{:}\mathsf{Bool} \mid x ==_{\mathsf{Bool}} \texttt{false}\} \\
\mathsf{TyCon}(\texttt{unit}) & \doteq & \{x{:}() \mid x ==_{()} \texttt{unit}\} \\
\mathsf{TyCon}(==_b) & \doteq & x{:}b \to y{:}b \to \{z{:}\mathsf{Bool} \mid z ==_{\mathsf{Bool}} (x ==_b y)\}
\end{array}
$$

Our system could of course be extended with further constants, as long as they belong in the interpretation of their type. This requirement is formally defined by the Property 1 which, for the four constants of our system is proved in Theorem B.1

Property 1 (Constants). $c \in [\![\mathsf{TyCon}(c)]\!]$

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

$$\frac{\Gamma \vdash e :: \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e :: \tau'}\textsc{TSub} \qquad \frac{\Gamma \vdash e :: \{z{:}b \mid r\}}{\Gamma \vdash e :: \{z{:}b \mid z ==_b e\}}\textsc{TSelf} \qquad \frac{}{\Gamma \vdash c :: \mathsf{TyCon}(c)}\textsc{TCon}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau}\textsc{TVar} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash e :: \tau}{\Gamma \vdash \lambda x{:}\tau_x.\ e :: x{:}\tau_x \to \tau}\textsc{TLam} \qquad \frac{\Gamma \vdash e_x :: \tau_x \qquad \Gamma \vdash e :: x{:}\tau_x \to \tau}{\Gamma \vdash e\ e_x :: \tau[e_x/x]}\textsc{TApp}$$

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

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

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

$$\frac{\lfloor \Gamma \rfloor, x : b \vdash_B r :: \mathsf{Bool}}{\Gamma \vdash \{x{:}b \mid r\}}\textsc{WFBase} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau}\textsc{WFFun}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash e_l :: \tau \qquad \Gamma \vdash e_r :: \tau}{\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}}\textsc{WFEq} \qquad \frac{}{\vdash \emptyset}\textsc{WFEmp} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}\textsc{WFBind}$$

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

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

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

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

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

Fig. 9. Typing of λ^{RE}.

6.2 Static Semantics of λ^{RE}

λ^{RE}'s static semantics comes in two parts: as typing judgments (§6.2.1) and as a binary logical relation characterizing equivalence (§6.2.2).

6.2.1 Typing of λ^{RE}. Type checking in λ^{RE} uses three mutually recursive judgments (Figure 9): *type checking*, $\Gamma \vdash e :: \tau$, for when e has type τ in Γ; *well formedness*, $\Gamma \vdash \tau$, for when when τ is well formed in Γ; and *subtyping*, $\Gamma \vdash \tau_l \preceq \tau_r$, for when when τ_l is a subtype of τ_r in Γ.

Type Checking. Beyond the conventional rules for refinement type systems [Knowles and Flanagan 2010; Ou et al. 2004; Rondon et al. 2008], the interesting rules are concerned with 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, 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 (rule TSelf), but does not define an algorithmic procedure to generate them. In Liquid Haskell, type inference [Rondon et al. 2008]

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

$$
\begin{array}{lcl}
c \sim c \;\; :: \{x{:}b \mid r\}; \delta & \doteq & \vdash_B c :: b \wedge \delta_1 \cdot r[c/x] \hookrightarrow^* \texttt{true} \wedge \delta_2 \cdot r[c/x] \hookrightarrow^* \texttt{true} \\
v_1 \sim v_2 :: x{:}\tau_x \to \tau; \delta & \doteq & \forall v_3 \sim v_4 :: \tau_x; \delta.\ v_1\ v_3 \sim v_2\ v_4 :: \tau; \delta, (v_3, v44)/x \\
v_1 \sim v_2 :: \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}; \delta & \doteq & \delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: \tau; \delta
\end{array}
$$

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

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

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

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

Fig. 10. Definition of equivalence logical relation.

automatically and algorithmically derives such strong types. We don't encumber λ^{RE} with inference, since, formally speaking, we can always 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.

Well Formedness. 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. Basic types are related by set inclusion on the interpretation of those types (SBase, and Figure 8). 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 indices, 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 invariantly to be consistent with the implementation (§5) where the GADT encoding of `PEq` is invariant. Our subtyping rule allows equality proofs between functions with convertible types (§4.2).

6.2.2 *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 10). 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. We read $v_1 \sim v_2 :: \tau; \delta$ as saying that 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—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 $\delta_2 \cdot e_r$. 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 pending substitions 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 relates pairs of values at the type in Γ. 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 $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau; \delta$. 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.

6.3 Metaproperties: PEq is an Equivalence Relation

Finally, we show various metaproperties of λ^{RE}. Theorem 6.1 proves soundness of syntactic typing with respect to semantic typing. Theorem 6.2 proves that propositional equality implies equivalence in the term model. Theorems 6.3 and 6.4 prove that both the equivalence relation and propositional equality define equivalences, i.e., satisfy the three equality axioms. All the proofs are in Appendix B.

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

Theorem 6.1 (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 [Wright and Felleisen 1994], for two reasons. First, and most essentially, SBase needs to quantify over all closing substitutions and purely syntactic approaches flirt with non-monotonicity (though others have attempted syntactic approaches in similar systems [Zalewski et al. 2020]). Second, and merely coincidentally, our system does not enjoy subject reduction. In particular, SEq allows us to change the type index of propositional equality, but not the term index. Why? Consider the term:

$$(\lambda x{:}\{x{:}\mathsf{Bool} \mid \mathsf{true}\}.\ \mathsf{bEq}_{\mathsf{Bool}}\ x\ x\ ())\ e$$

such that $e \hookrightarrow e'$ for some e'. The whole application has type $\mathsf{PEq}_{\mathsf{Bool}}\ \{e\}\ \{e\}$; after we take a step, it will have type $\mathsf{PEq}_{\mathsf{Bool}}\ \{e'\}\ \{e'\}$. Subject reduction demands that the latter is a subtype of the former. We have

$$\mathsf{PEq}_{\mathsf{Bool}}\ \{e\}\ \{e\} \rightrightarrows \mathsf{PEq}_{\mathsf{Bool}}\ \{e'\}\ \{e'\}$$

so we could recover subject reduction by allowing a supertype's terms to parallel reduce (or otherwise convert) to a subtype's terms. Adding this condition would not be hard: the logical relations' metatheory already demands a variety of lemmas about parallel reduction, relegated to supplementary material(Appendix C) to avoid distraction and preserve space for our main contributions. We haven't made this change because subject reduction isn't necessary for our purposes.

Theorem 6.2 (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 (see Theorem B.13)is a corollary of the fundamental property of the logical relation(Theorem B.22), 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 6.3 (The logical relation is an Equivalence). $\Gamma \vdash e_1 \sim e_2 :: \tau$ *is reflexive, symmetric, and transitive:*

- *Reflexivity: If* $\Gamma \vdash e :: \tau$*, then* $\Gamma \vdash e \sim e \ :: \ \tau$.
- *Symmetry: If* $\Gamma \vdash e_1 \sim e_2 \ :: \ \tau$*, then* $\Gamma \vdash e_2 \sim e_1 \ :: \ \tau$.
- *Transitivity: If* $\Gamma \vdash e_2 :: \tau$*,* $\Gamma \vdash e_1 \sim e_2 \ :: \ \tau$*, and* $\Gamma \vdash e_2 \sim e_3 \ :: \ \tau$*, then* $\Gamma \vdash e_1 \sim e_3 \ :: \ \tau$.

Reflexivity is also called the *fundamental property* of the logical relation. The other proofs go by structural induction on τ (Theorem B.23). Transitivity requires reflexivity on e_2, so we also assume that $\Gamma \vdash e_2 :: \tau$.

Theorem 6.4 (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 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* $\Gamma \vdash v_{23} :: \mathsf{PEq}_\tau\ \{e_2\}\ \{e_3\}$*, 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(Theorem B.24). Reflexivity requires us to generalize the inductive hypothesis to generate appropriate τ_l and τ_r for the `PEq` proofs.

7 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 [FStarLang 2018]. Their extensionality axiom makes a more roundabout connection with SMT: function equality uses ==, a proof-irrelevant, propositional 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 [Leino 2012]. Function equality is utterable but neither provable nor disprovable in their encoding into Z3.

Ou et al. [2004] introduce *selfification*, which assigns singleton types using equality (as in our TSelf rule). SAGE assigns selfified types to *all* variables, implying equality on functions [Knowles et al. 2006]. Dminor avoids the question: it lacks first-class functions [Bierman et al. 2012].

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) [Martin-Löf 1975] as well as an undecidable, extensional one (ETT) [Martin-Löf 1984]. NuPRL implements ETT [Constable et al. 1986], while Coq and Agda implement ITT [2008; 2020]. Lean's quotient-based reasoning can *prove* `funext` [de Moura et al. 2015]. Extensionality axioms are independent of the rules of ITT; `funext` is a common axiom, but is not consistent in every model of type theory [von Glehn 2014]. Hofmann [1996] shows that ETT is a conservative but less computational extension of ITT with `funext` and UIP. Pfenning [2001] and Altenkirch and McBride [2006] 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 allowed equalities between equalities, we could add UIP. Our propositional equality isn't exactly Leibniz equality, so axiom K would be harder to encode.

Zombie's type theory uses an adaptation of a congruence closure algorithm to automatically reason about equality [Sjöberg and Weirich 2015]. 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 offers alternatives to our propositional equality [Sterling et al. 2019]. 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 (§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 five independent uses of "classy induction" in the literature [Boulier et al. 2017; Dagand et al. 2018; Guillemette and Monnier 2008; Tabareau et al. 2019; Weirich 2017].

Any typeclass system that accommodates ad-hoc polymorphism and a notion of proof can use classy induction. Sozeau [2008] 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 [2008] 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 [Nipkow and Prehofer 1993; Nipkow and Snelting 1991; Wadler and Blott 1989], with no notion of classy induction even in settings with proofs [Wenzel 1997].

8 CONCLUSION

In a refinement type system with subtyping a naive encoding of `funext` is inconsistent. We explained the inconsistency by examples (that proved `false`) and by standard type checking (where the equality domain is inferred as `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. Stephanie Weirich, Éric Tanter, and Nicolas Tabareau offered valuable insights into the folklore of classy induction.

REFERENCES

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

Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. 2019. Extending SMT Solvers to Higher-Order Logic. In *Automated Deduction – CADE 27*, Pascal Fontaine (Ed.). Springer International Publishing, Cham, 35–54.

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. Available at `www.SMT-LIB.org`.

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

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

James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types.* Technical Report. Cornell University.

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

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

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

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, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

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.

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* (Victoria, BC, Canada) *(ICFP âĂŹ08)*. Association for Computing Machinery, New York, NY, USA, 75âĂŞ86. https://doi.org/10.1145/1411204.1411218

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.

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

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*.

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

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)*.

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

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

Tobias Nipkow and Christian Prehofer. 1993. Type Checking Type Classes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL âĂŹ93)*. Association for Computing Machinery, New York, NY, USA, 409âĂŞ418. https://doi.org/10.1145/158511.158698

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, 1âĂŞ14.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFPâĂŹ08)*. Springer-Verlag, Berlin, Heidelberg, 230âĂŞ266.

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, Vol. 155)*, Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell (Eds.). Kluwer/Springer, 437–450. https://doi.org/10.1007/1-4020-8141-3_34

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.

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* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

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.

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

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* (Mumbai, India) *(POPL âĂŹ15)*. Association for Computing Machinery, New York, NY, USA, 369âĂŞ382. https://doi.org/10.1145/2676726.2676974

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

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.

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 supplementary material. 2021. Supplementary Material for Functional Extensionality for Refinement Types.

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/

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:1909.05027 [cs.PL]

Masako Takahashi. 1989. Parallel Reductions in lambda-Calculus. *J. Symb. Comput.* 7, 2 (1989), 113–123. https://doi.org/10.1016/S0747-7171(89)80045-8

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

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018a. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. ACM, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

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

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

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* (Austin, Texas, USA) *(POPL âĂŽ89)*. Association for Computing Machinery, New York, NY, USA, 60âĂŞ76. https://doi.org/10.1145/75277.75283

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

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.

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

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224âĂŞ235. https://doi.org/10.1145/604131.604150

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.

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

A COMPLETE TYPE CHECKING OF EXTENSIONALITY EXAMPLE

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{funext}) \;=\; \forall a\,b.\mathtt{Eq}\ b \Rightarrow f:(a\to b)\to g:(a\to b)\to(x:a\to\{f\,x==g\,x\})\to\{f\simeq g\}}{\Gamma\vdash\mathtt{funext}::\forall a\,b.\mathtt{Eq}\ b \Rightarrow f:(a\to b)\to g:(a\to b)\to(x:a\to\{f\,x==g\,x\})\to\{f\simeq g\}}}{\Gamma\vdash\mathtt{funext}\ @\{\!|\kappa_\alpha|\!\}::\forall b.\mathtt{Eq}\ b\Rightarrow f:(\{\!|\kappa_\alpha|\!\}\to b)\to g:(\{\!|\kappa_\alpha|\!\}\to b)\to(x:\{\!|\kappa_\alpha|\!\}\to\{f\,x==g\,x\})\to\{f\simeq g\}}}{\Gamma\vdash\mathtt{funext}\ @\{\!|\kappa_\alpha|\!\}\ @\{\!|\kappa_\beta|\!\}::\mathtt{Eq}\ \{\!|\kappa_\beta|\!\}\Rightarrow f:(\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\})\to g:(\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\})\to(x:\{\!|\kappa_\alpha|\!\}\to\{f\,x==g\,x\})\to\{f\simeq g\}} \quad \dfrac{\Gamma(\mathsf{d}) \;=\; \mathtt{Eq}\ \alpha}{\Gamma\vdash\mathsf{d}::\mathtt{Eq}\ \alpha} \quad \dfrac{}{\Gamma\vdash\mathtt{Eq}\ \alpha\;\preceq\;\mathtt{Eq}\ \{\!|\kappa_\beta|\!\}}\textsc{Sub-D}}{\Gamma\vdash\mathtt{funext}\ @\{\!|\kappa_\alpha|\!\}\ @\{\!|\kappa_\beta|\!\}\ \mathsf{d}::f:(\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\})\to g:(\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\})\to(x:\{\!|\kappa_\alpha|\!\}\to\{f\,x==g\,x\})\to\{f\simeq g\}} \quad \dfrac{\Gamma(\mathsf{h}) \;=\; x:\{\!|d_{\mathsf{h}}|\!\}\to\{\!|r_{\mathsf{h}}|\!\}}{\Gamma\vdash\mathsf{h}::x:\{\!|d_{\mathsf{h}}|\!\}\to\{\!|r_{\mathsf{h}}|\!\}} \quad \dfrac{\cdots}{\Gamma\vdash x:\{\!|d_{\mathsf{h}}|\!\}\to\{\!|r_{\mathsf{h}}|\!\}\;\preceq\;\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\}}\textsc{Sub-H}}{\Gamma\vdash\mathtt{funext}\ @\{\!|\kappa_\alpha|\!\}\ @\{\!|\kappa_\beta|\!\}\ \mathsf{d\ h}::g:(\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\})\to(x:\{\!|\kappa_\alpha|\!\}\to\{\mathsf{h}\,x==g\,x\})\to\{\mathsf{h}\simeq g\}} \quad \dfrac{\Gamma(\mathsf{k}) \;=\; x:\{\!|d_{\mathsf{k}}|\!\}\to\{\!|r_{\mathsf{k}}|\!\}}{\Gamma\vdash\mathsf{k}::x:\{\!|d_{\mathsf{k}}|\!\}\to\{\!|r_{\mathsf{k}}|\!\}} \quad \dfrac{\cdots}{\Gamma\vdash x:\{\!|d_{\mathsf{k}}|\!\}\to\{\!|r_{\mathsf{k}}|\!\}\;\preceq\;\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\}}\textsc{Sub-K}}{\Gamma\vdash\mathtt{funext}\ @\{\!|\kappa_\alpha|\!\}\ @\{\!|\kappa_\beta|\!\}\ \mathsf{d\ h\ k}::(x:\{\!|\kappa_\alpha|\!\}\to\{\mathsf{h}\,x==\mathsf{k}\,x\})\to\{\mathsf{h}\simeq\mathsf{k}\}} \quad \dfrac{\Gamma(\mathtt{lemma}) \;=\; x:\{\!|d_p|\!\}\to\{p\}}{\Gamma\vdash\mathtt{lemma}::x:\{\!|d_p|\!\}\to\{p\}} \quad \dfrac{\cdots}{\Gamma\vdash x:\{\!|d_p|\!\}\to\{p\}\;\preceq\;x:\{\!|\kappa_\alpha|\!\}\to\{\mathsf{h}\,x==\mathsf{k}\,x\}}\textsc{Sub-L}}{\Gamma\vdash\mathtt{funext}\ @\{\!|\kappa_\alpha|\!\}\ @\{\!|\kappa_\beta|\!\}\ \mathsf{d\ h\ k}\ \mathtt{lemma}::\{h\simeq k\}}
$$

$$
\dfrac{\dfrac{\kappa_\alpha\Rightarrow d_{\mathsf{h}}}{\Gamma\vdash\{\!|\kappa_\alpha|\!\}\;\preceq\;\{v:\alpha\mid d_{\mathsf{h}}\}} \qquad \dfrac{\kappa_\alpha\Rightarrow r_{\mathsf{h}}\Rightarrow\kappa_\beta}{\Gamma,x:\{\!|\kappa_\alpha|\!\}\vdash\{v:\beta\mid r_{\mathsf{h}}\}\;\preceq\;\{\!|\kappa_\beta|\!\}}}{\Gamma\vdash x:\{\!|d_{\mathsf{h}}|\!\}\to\{\!|r_{\mathsf{h}}|\!\}\;\preceq\;\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\}}\textsc{Sub-H}
$$

$$
\dfrac{\dfrac{\kappa_\alpha\Rightarrow d_{\mathsf{k}}}{\Gamma\vdash\{\!|\kappa_\alpha|\!\}\;\preceq\;\{v:\alpha\mid d_{\mathsf{k}}\}} \qquad \dfrac{\kappa_\alpha\Rightarrow r_{\mathsf{k}}\Rightarrow\kappa_\beta}{\Gamma,x:\{\!|\kappa_\alpha|\!\}\vdash\{v:\beta\mid r_{\mathsf{k}}\}\;\preceq\;\{\!|\kappa_\beta|\!\}}}{\Gamma\vdash x:\{\!|d_{\mathsf{k}}|\!\}\to\{\!|r_{\mathsf{k}}|\!\}\;\preceq\;\{\!|\kappa_\alpha|\!\}\to\{\!|\kappa_\beta|\!\}}\textsc{Sub-K}
$$

$$
\dfrac{\dfrac{\kappa_\alpha\Rightarrow\mathtt{true}}{\Gamma\vdash\{\!|\kappa_\alpha|\!\}\;\preceq\;\alpha} \qquad \dfrac{\kappa_\alpha\Rightarrow p\Rightarrow h\,x==k\,x}{\Gamma,x:\{\!|\kappa_\alpha|\!\}\vdash\{p\}\;\preceq\;\{h\,x==k\,x\}}}{\Gamma\vdash x:\{\!|d_p|\!\}\to\{p\}\;\preceq\;x:\{\!|\kappa_\alpha|\!\}\to\{\mathsf{h}\,x==\mathsf{k}\,x\}}\textsc{Sub-L}
$$

Fig. 11. Complete type checking of naive extensionality in theoremEq.

$$\begin{array}{rrcl} \textit{Expressions} & e & ::= & \text{as in } \lambda^{RE} \\ \textit{Types} & t & ::= & \mathsf{Bool} \mid () \mid \mathsf{PBEq}_t ee \mid t \rightarrow t \\ \textit{Typing Environment} & G & ::= & \emptyset \mid G, x : t \end{array}$$

Basic Type checking $\boxed{G \vdash_B e :: t}$

$$\frac{}{G \vdash_B c :: \lfloor \mathsf{TyCon}(c) \rfloor}\textsc{BTCon} \qquad \frac{x : t \in G}{G \vdash_B x :: t}\textsc{BTVar}$$

$$\frac{G \vdash_B e :: t_x \rightarrow t \qquad G \vdash_B e_x :: t_x}{G \vdash_B e\, e_x :: t}\textsc{BTApp} \qquad \frac{G, x : \lfloor \tau_x \rfloor \vdash_B e :: t}{G \vdash_B \lambda x{:}\tau_x.\, e :: \lfloor \tau_x \rfloor \rightarrow t}\textsc{BTLam}$$

$$\frac{\begin{array}{c} G \vdash_B e :: () \\ G \vdash_B e_1 :: b \qquad G \vdash_B e_2 :: b \end{array}}{G \vdash_B \mathsf{bEq}_b\ e_1\ e_2\ e :: \mathsf{PBEq}_b e_1 e_2}\textsc{BTEqBase} \qquad \frac{\begin{array}{c} G \vdash_B e :: () \\ G \vdash_B e_1 :: \lfloor \tau_x \rightarrow \tau \rfloor \qquad G \vdash_B e_2 :: \lfloor \tau_x \rightarrow \tau \rfloor \end{array}}{G \vdash_B \mathsf{xEq}_{x:\tau_x \rightarrow \tau}\ e_1\ e_2\ e :: \mathsf{PBEq}_{\lfloor \tau_x \rightarrow \tau \rfloor} e_1 e_2}\textsc{BTEqFun}$$

Fig. 12. Syntax and Typing of λ^E.

B PROOFS AND DEFINITIONS FOR METATHEORY

In this section we provide proofs and definitions ommitted from §6.

B.1 Base Type Checking

For completeness, we defined λ^E, the unrefined version of λ^{RE}, that ignores the refinements on basic types and the expression indices from the typed equality.

The function $\lfloor \cdot \rfloor$ is defined to turn λ^{RE} types to their unrefined counterparts.

$$\begin{array}{rcl} \lfloor \mathsf{Bool} \rfloor & \doteq & \mathsf{Bool} \\ \lfloor () \rfloor & \doteq & () \\ \lfloor \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\} \rfloor & \doteq & \mathsf{PBEq}_{\lfloor \tau \rfloor} \\ \lfloor \{v{:}b \mid r\} \rfloor & \doteq & b \\ \lfloor x{:}\tau_x \rightarrow \tau \rfloor & \doteq & \lfloor \tau_x \rfloor \rightarrow \lfloor \tau \rfloor \end{array}$$

Figure 12 defines the syntax and typing of λ^E that we use to define type denotations of λ^{RE}.

B.2 Constant Property

Theorem B.1. *For the constants* $c = \mathtt{true}, \mathtt{false}, \mathtt{unit},$ *and* $==_b$*, Property 1 holds, i.e.,* $c \in [\![\mathsf{TyCon}(c)]\!]$.

Proof. Below are the proofs for each of the four constants.

- $e \equiv \mathtt{true}$ and $e \in [\![\{x{:}\mathsf{Bool} \mid x ==_{\mathsf{Bool}} \mathtt{true}\}]\!]$. We need to prove the below three requirements of membership in the interpretation of basic types:
  - $e \hookrightarrow^* v$, which holds because $\mathtt{true}$ is a value, thus $v = \mathtt{true}$;
  - $\vdash_B e :: \mathsf{Bool}$, which holds by the typing rule BTCon; and
  - $(x ==_{\mathsf{Bool}} \mathtt{true})[e/x] \hookrightarrow^* \mathtt{true}$, which holds because

$$\begin{array}{rcl} (x ==_{\mathsf{Bool}} \mathtt{true})[e/x] & = & \mathtt{true} ==_{\mathsf{Bool}} \mathtt{true} \\ & \hookrightarrow & (==_{(\mathtt{true},\mathsf{Bool})})\ \mathtt{true} \\ & \hookrightarrow & \mathtt{true} = \mathtt{true} \\ & = & \mathtt{true} \end{array}$$

- $e \equiv \texttt{false}$ and $e \in [\![\{x{:}\texttt{Bool} \mid x ==_{\texttt{Bool}} \texttt{false}\}]\!]$. We need to prove the below three requirements of membership in the interpretation of basic types:
  - $e \hookrightarrow^* v$, which holds because $\texttt{false}$ is a value, thus $v = \texttt{false}$;
  - $\vdash_B e :: \texttt{Bool}$, which holds by the typing rule BTCon; and
  - $(x ==_{\texttt{Bool}} \texttt{false})[e/x] \hookrightarrow^* \texttt{true}$, which holds because

$$\begin{array}{rcl} (x ==_{\texttt{Bool}} \texttt{false})[e/x] & = & \texttt{false} ==_{\texttt{Bool}} \texttt{false} \\ & \hookrightarrow & (==_{(\texttt{false},\texttt{Bool})})\ \texttt{false} \\ & \hookrightarrow & \texttt{false} = \texttt{false} \\ & = & \texttt{true} \end{array}$$

- $e \equiv \texttt{unit}$ and $e \in \left[\!\!\left[\{x{:}() \mid x ==_{()} \texttt{unit}\}\right]\!\!\right]$. We need to prove the below three requirements of membership in the interpretation of basic types:
  - $e \hookrightarrow^* v$, which holds because $\texttt{unit}$ is a value, thus $v = \texttt{unit}$;
  - $\vdash_B e :: ()$, which holds by the typing rule BTCon; and
  - $(x ==_{()} \texttt{unit})[e/x] \hookrightarrow^* \texttt{true}$, which holds because

$$\begin{array}{rcl} (x ==_{()} \texttt{unit})[e/x] & = & \texttt{unit} ==_{()} \texttt{unit} \\ & \hookrightarrow & (==_{(\texttt{unit},())})\ \texttt{unit} \\ & \hookrightarrow & \texttt{unit} = \texttt{unit} \\ & = & \texttt{true} \end{array}$$

- $==_b \in [\![x{:}b \rightarrow y{:}b \rightarrow \{z{:}\texttt{Bool} \mid z ==_{\texttt{Bool}} (x ==_b y)\}]\!]$. By the definition of interpretation of function types, we fix $e_x, e_y \in [\![b]\!]$ and we need to prove that $e \equiv e_x ==_b e_y \in \left[\!\!\left[(\{z{:}\texttt{Bool} \mid z ==_{\texttt{Bool}} (x ==_b\right.\right.$
We prove the below three requirements of membership in the interpretation of basic types:
  - $e \hookrightarrow^* v$, which holds because

$$\begin{array}{rcll} e & = & e_x ==_b e_y & \\ & \hookrightarrow^* & v_x ==_b e_y & \text{because } e_x \in [\![b]\!] \\ & \hookrightarrow^* & v_x ==_b v_y & \text{because } e_y \in [\![b]\!] \\ & \hookrightarrow & (==_{(v_x,b)})\ v_y & \\ & \hookrightarrow & v_x = v_y & \\ & = & v & \text{with } v = \texttt{true} \text{ or } v = \texttt{false} \end{array}$$

  - $\vdash_B e :: \texttt{Bool}$, which holds by the typing rule BTCon and because $e_x, e_y \in [\![b]\!]$ thus $\vdash_B e_x :: b$ and $\vdash_B e_y :: b$; and

– $(z ==_{\texttt{Bool}} (x ==_b y))[e/z][e_x/x][e_y/y] \hookrightarrow^* \texttt{true}$. Since $e_x, e_y \in [\![b]\!]$ both expressions evaluate to values, say $e_x \hookrightarrow^* v_x$ and $e_y \hookrightarrow^* v_y$ which holds because

$$
\begin{array}{rcll}
(z ==_{\texttt{Bool}} (x ==_b y))[e/z][e_x/x][e_y/y] & = & e ==_{\texttt{Bool}} (e_x ==_b e_y) & \\
& = & (e_x ==_b e_y) ==_{\texttt{Bool}} (e_x ==_b e_y) & \\
& \hookrightarrow^* & (v_x ==_b e_y) ==_{\texttt{Bool}} (e_x ==_b e_y) & \text{since } e_x \hookrightarrow^* v_x \\
& \hookrightarrow^* & (v_x ==_b v_y) ==_{\texttt{Bool}} (e_x ==_b e_y) & \text{since } e_y \hookrightarrow^* v_y \\
& \hookrightarrow & ((==_{(v_x, b)})\ v_y) ==_{\texttt{Bool}} (e_x ==_b e_y) & \\
& \hookrightarrow & (v_x = v_y) ==_{\texttt{Bool}} (e_x ==_b e_y) & \\
& \hookrightarrow^* & (v_x = v_y) ==_{\texttt{Bool}} (v_x ==_b e_y) & \text{since } e_x \hookrightarrow^* v_x \\
& \hookrightarrow^* & (v_x = v_y) ==_{\texttt{Bool}} (v_x ==_b v_y) & \text{since } e_y \hookrightarrow^* v_y \\
& \hookrightarrow & (v_x = v_y) ==_{\texttt{Bool}} ((==_{(v_x, b)})\ v_y) & \\
& \hookrightarrow & (v_x = v_y) ==_{\texttt{Bool}} (v_x = v_y) & \\
& \hookrightarrow & (v_x = v_y) ==_{\texttt{Bool}} (v_x = v_y) & \\
& \hookrightarrow & ((==_{((v_x = v_y), \texttt{Bool})})\ (v_x = v_y) & \\
& \hookrightarrow & (v_x = v_y) = (v_x = v_y) & \\
& = & \texttt{true} &
\end{array}
$$

□

B.3 Type Soundness

Theorem B.2 (Semantic soundness). *If* $\Gamma \vdash e :: \tau$ *then* $\Gamma \models e \in \tau$.

Proof. By induction on the typing derivation.

TSub By inversion of the rule we have
(1) $\Gamma \vdash e :: \tau'$
(2) $\Gamma \vdash \tau' \preceq \tau$
By IH on (1) we have
(3) $\Gamma \models e \in \tau'$
By Theorem B.6 and (2) we have
(4) $\Gamma \vdash \tau' \subseteq \tau$
By (3), (4), and the definition of subsets we directly get $\Gamma \models e \in \tau$.

TSelf Assume $\Gamma \vdash e :: \{z{:}b \mid z ==_b e\}$. By inversion we have
(1) $\Gamma \vdash e :: \{z{:}b \mid r\}$
By IH we have
(2) $\Gamma \models e \in \{z{:}b \mid r\}$
We fix $\theta \in [\![\Gamma]\!]$. By the definition of semantic typing we get
(3) $\theta \cdot e \in [\![\theta \cdot \{z{:}b \mid r\}]\!]$
By the definition of denotations on basic types we have
(4) $\theta \cdot e \hookrightarrow^* v$
(5) $\vdash_B \theta \cdot e :: b$
(6) $\theta \cdot r[\theta \cdot e/z] \hookrightarrow^* \texttt{true}$
Since θ contains values, by the definition of $==_b$ we have
(7) $\theta \cdot e ==_b \theta \cdot e \hookrightarrow^* \texttt{true}$
Thus
(8) $\theta \cdot (z ==_b e)[\theta \cdot e/z] \hookrightarrow^* \texttt{true}$
By (4), (5), and (8) we have
(9) $\theta \cdot e \in [\![\theta \cdot \{z{:}b \mid z ==_b e\}]\!]$
Thus, $\Gamma \models e \in \{z{:}b \mid z ==_b e\}$.

TCon This case holds exactly because of Property B.1.

TVar This case holds by the definition of closing substitutions.

TLam Assume $\Gamma \vdash \lambda x{:}\tau_x.\ e :: x{:}\tau_x \to \tau$. By inversion of the rule we have $\Gamma, x : \tau_x \vdash e :: \tau$. By IH we get $\Gamma, x : \tau_x \models e \in \tau$.

We need to show that $\Gamma \models \lambda x{:}\tau_x.\ e \in x{:}\tau_x \to \tau$. Which, for some $\theta \in [\![\Gamma]\!]$ is equivalent to $\lambda x{:}\theta \cdot \tau_x.\ \theta \cdot e \in [\![x{:}\theta \cdot \tau_x \to \theta \cdot \tau]\!]$.

We pick a random $e_x \in [\![\theta \cdot \tau_x]\!]$ thus we need to show that $\theta \cdot e[e_x/x] \in [\![\theta \cdot \tau[e_x/x]]\!]$. By Lemma B.3, there exists v_x so that $e_x \hookrightarrow^* v_x$ and $v_x \in [\![\tau_x]\!]$. By the inductive hypothesis, $\theta \cdot e[v_x/x] \in [\![\theta \cdot \tau[v_x/x]]\!]$. By Lemma B.4, $\theta \cdot e[e_x/x] \in [\![\theta \cdot \tau[e_x/x]]\!]$, which concludes our proof.

TApp Assume $\Gamma \vdash e\ e_x :: \tau[e_x/x]$. By inversion we have

(1) $\Gamma \vdash e :: x{:}\tau_x \to \tau$

(2) $\Gamma \vdash e_x :: \tau_x$

By IH we get

(3) $\Gamma \models e \in x{:}\tau_x \to \tau$

(4) $\Gamma \models e_x \in \tau_x$

We fix $\theta \in [\![\Gamma]\!]$. By the definition of semantic types

(5) $\theta \cdot e \in [\![\theta \cdot x{:}\tau_x \to \tau]\!]$

(6) $\theta \cdot e_x \in [\![\theta \cdot \tau_x]\!]$

By (5), (6), and the definition of semantic typing on functions:

(7) $\theta \cdot e\ e_x \in [\![\theta \cdot \tau[e_x/x]]\!]$

Which directly leads to the required $\Gamma \models e\ e_x \in \tau[e_x/x]$

TEqBase Assume $\Gamma \vdash \mathsf{bEq}_b\ e_l\ e_r\ e :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$. By inversion we get:

(1) $\Gamma \vdash e_l :: \tau_l$

(2) $\Gamma \vdash e_r :: \tau_r$

(3) $\Gamma \vdash \tau_l \preceq \{x{:}b \mid \mathtt{true}\}$

(4) $\Gamma \vdash \tau_r \preceq \{x{:}b \mid \mathtt{true}\}$

(5) $\Gamma, r : \tau_r, l : \tau_l \vdash e :: \{x{:}() \mid l ==_b r\}$

By IH we get

(4) $\Gamma \models e_l \in \tau_l$

(5) $\Gamma \models e_r \in \tau_r$

(6) $\Gamma, r : \tau_r, l : \tau_l \models e \in \{x{:}() \mid l ==_b r\}$

We fix $\theta \in [\![\Gamma]\!]$. Then (4) and (5) become

(7) $\theta \cdot e_l \in [\![\theta \cdot \tau_l]\!]$

(8) $\theta \cdot e_r \in [\![\theta \cdot \tau_r]\!]$

(9) $\Gamma \models e_r \in \tau_r$

(10) $\Gamma, r : \tau_r, l : \tau_l \models e \in \{x{:}() \mid l ==_b r\}$

Assume

(11) $\theta \cdot e_l \hookrightarrow^* v_l$

(12) $\theta \cdot e_r \hookrightarrow^* v_r$

By (7), (8), (11), (12), and Lemma B.3 we get

(13) $v_l \in [\![\theta \cdot \tau_l]\!]$

(14) $v_r \in [\![\theta \cdot \tau_r]\!]$

By (10), (11), and (12) we get

(15) $v_l ==_b v_r \hookrightarrow^* \mathtt{true}$

By (11), (12), (15), ane Lemma B.5 we have

(16) $\theta \cdot e_l ==_b \theta \cdot e_r \hookrightarrow^* \mathtt{true}$

By (1-5) we get:

(17) $\vdash_B \theta \cdot \mathsf{bEq}_b\ e_l\ e_r\ e\ ::\ \mathsf{PBEq}_b$
Trivially, with zero evaluation steps we have:
(18) $\theta \cdot \mathsf{bEq}_b\ e_l\ e_r\ e \hookrightarrow^* \mathsf{bEq}_b\ (\theta \cdot e_l)\ (\theta \cdot e_l)\ (\theta \cdot e)$
By (16), (17), (18) and the definition of semantic types on basic equality types we have
(19) $\theta \cdot \mathsf{bEq}_b\ e_l\ e_r\ e \in [\![\theta \cdot \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}]\!]$
Which leads to the required $\Gamma \models \mathsf{bEq}_b\ e_l\ e_r\ e \in \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$.

TEqFun Assume $\Gamma \vdash \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e :: \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}$. By inversion we have
(1) $\Gamma \vdash e_l :: \tau_l$
(2) $\Gamma \vdash e_r :: \tau_r$
(3) $\Gamma \vdash \tau_l \preceq x{:}\tau_x \to \tau$
(4) $\Gamma \vdash \tau_r \preceq x{:}\tau_x \to \tau$
(5) $\Gamma, r : \tau_r, l : \tau_l \vdash e :: (x{:}\tau_x \to \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\})$
(6) $\Gamma \vdash x{:}\tau_x \to \tau$
By IH and Theorem B.6 we get
(7) $\Gamma \models e_l \in \tau_l$
(8) $\Gamma \models e_r \in \tau_r$
(9) $\Gamma \vdash \tau_l \subseteq x{:}\tau_x \to \tau$
(10) $\Gamma \vdash \tau_r \subseteq x{:}\tau_x \to \tau$
(11) $\Gamma, r : \tau_r, l : \tau_l \models e \in (x{:}\tau_x \to \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\})$
By (1-5) we get
(12) $\vdash_B \theta \cdot \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e\ ::\ \mathsf{PBEq}_{\lfloor \theta \cdot (x:\tau_x \to \tau) \rfloor}$
Trivially, by zero evaluation steps, we get
(13) $\theta \cdot \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \hookrightarrow^* \mathsf{xEq}_{x:\theta \cdot \tau_x \to \theta \cdot \tau}\ (\theta \cdot e_l)\ (\theta \cdot e_r)\ (\theta \cdot e)$
By (7-10) we get
(14) $\theta \cdot e_l, \theta \cdot e_r \in [\![\theta \cdot x{:}\tau_x \to \tau]\!]$
By (7), (8), (11), the definition of semantic types on functions, and Lemmata B.3 and B.4 (similar to the previous case) we have
– $\forall e_x \in [\![\tau_x]\!] . e\ e_x \in \left[\!\left[\mathsf{PEq}_{\tau[e_x/x]}\ \{e_l\ e_x\}\ \{e_r\ e_x\}\right]\!\right]$
By (12), (13), (14), and (15) we get
(19) $\theta \cdot \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \in \left[\!\left[\theta \cdot \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}\right]\!\right]$
Which leads to the required $\Gamma \models \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \in \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}$.

□

Lemma B.3. *If $e \in [\![\tau]\!]$, then $e \hookrightarrow^* v$ and $v \in [\![\tau]\!]$.*

Proof. By structural induction of the type τ. □

Lemma B.4. *If $e_x \hookrightarrow^* v_x$ and $e[v_x/x] \in [\![\tau[v_x/x]]\!]$, then $e[e_x/x] \in [\![\tau[e_x/x]]\!]$.*

Proof. We can use parallel reductions (of §C) to prove that if $e_1 \rightrightarrows e_2$, then (1) $[\![\tau[e_1/x]]\!] = [\![\tau[e_2/x]]\!]$ and (2) $e_1 \in [\![\tau]\!]$ *iff* $e_2 \in [\![\tau]\!]$. The proof directly follows by these two properties. □

Lemma B.5. *If $e_x \hookrightarrow^* e'_x$ and $e[e'_x/x] \hookrightarrow^* c$, then $e[e_x/x] \hookrightarrow^* c$.*

Proof. As an instance of Corollary C.17. □

We define semantic subtyping as follows: $\Gamma \vdash \tau \subseteq \tau'$ iff $\forall \theta \in [\![\Gamma]\!] . [\![\theta \cdot \tau]\!] \subseteq [\![\theta \cdot \tau']\!]$.

Theorem B.6 (Subtyping semantic soundness). *If $\Gamma \vdash \tau \preceq \tau'$ then $\Gamma \vdash \tau \subseteq \tau'$.*

Proof. By induction on the derivation tree:

SBase Assume $\Gamma \vdash \{x{:}b \mid r\} \preceq \{x'{:}b \mid r'\}$. By inversion $\forall \theta \in [\![\Gamma]\!]$, $[\![\theta \cdot \{x{:}b \mid r\}]\!] \subseteq [\![\theta \cdot \{x'{:}b \mid r'\}]\!]$, which exactly leads to the required.

SFun Assume $\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'$. By inversion

(1) $\Gamma \vdash \tau'_x \preceq \tau_x$
(2) $\Gamma, x : \tau'_x \vdash \tau \preceq \tau'$
By IH
(3) $\Gamma \vdash \tau'_x \subseteq \tau_x$
(4) $\Gamma, x : \tau'_x \vdash \tau \subseteq \tau'$
We fix $\theta \in \Gamma$. We pick e. We assume $e \in [\![\theta \cdot x{:}\tau_x \to \tau]\!]$ and we will show that $e \in \left[\!\!\left[\theta \cdot x{:}\tau'_x \to \tau'\right]\!\!\right]$. By assumption
(5) $\forall e_x \in [\![\theta \cdot \tau_x]\!].\ e\ e_x \in [\![\theta \cdot \tau[e_x/x]]\!]$
We need to show $\forall e_x \in \left[\!\!\left[\theta \cdot \tau'_x\right]\!\!\right].\ e\ e_x \in [\![\theta \cdot \tau'[e_x/x]]\!]$. We fix e_x. By (3), if $e_x \in \left[\!\!\left[\theta \cdot \tau'_x\right]\!\!\right]$, then $e_x \in [\![\theta \cdot \tau_x]\!]$ and (5) applies, so $e\ e_x \in [\![\theta \cdot \tau[e_x/x]]\!]$, which by (4) gives $e\ e_x \in [\![\theta \cdot \tau'[e_x/x]]\!]$. Thus, $e \in \left[\!\!\left[\theta \cdot x{:}\tau'_x \to \tau'\right]\!\!\right]$. This leads to $[\![\theta \cdot x{:}\tau_x \to \tau]\!] \subseteq \left[\!\!\left[\theta \cdot x{:}\tau'_x \to \tau'\right]\!\!\right]$, which by definition gives semantic subtyping: $\Gamma \vdash x{:}\tau_x \to \tau \subseteq x{:}\tau'_x \to \tau'$.

SEq Assume $\Gamma \vdash \mathsf{PEq}_{\tau_i}\ \{e_l\}\ \{e_r\} \preceq \mathsf{PEq}_{\tau'_i}\ \{e_l\}\ \{e_r\}$. We split cases on the structure of τ_i.

– *If* τ_i is a basic type, then τ_i is trivially refined to true. Thus, $\tau_i = \tau'_i = b$ and for each $\theta \in \Gamma$, $[\![\theta \cdot \mathsf{PEq}_{\tau}\ \{e_l\}\ \{e_r\}]\!] = [\![\theta \cdot \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}]\!]$, thus set inclusion reduces to equal sets.
– *If* τ_i is a function type, thus $\Gamma \vdash \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\} \preceq \mathsf{PEq}_{x:\tau'_x \to \tau'}\ \{e_l\}\ \{e_r\}$
By inversion
(1) $\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'$
(2) $\Gamma \vdash x{:}\tau'_x \to \tau' \preceq x{:}\tau_x \to \tau$
By inversion on (1) and (2) we get
(3) $\Gamma \vdash \tau'_x \preceq \tau_x$
(4) $\Gamma, x : \tau'_x \vdash \tau \preceq \tau'$
(5) $\Gamma, x : \tau_x \vdash \tau' \preceq \tau$
By IH on (1) and (3) we get
(6) $\Gamma \vdash x{:}\tau_x \to \tau \subseteq x{:}\tau'_x \to \tau'$
(7) $\Gamma \vdash \tau'_x \subseteq \tau_x$
We fix $\theta \in \Gamma$ and some e. If $e \in \left[\!\!\left[\theta \cdot \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\}\right]\!\!\right]$ we need to show that $e \in \left[\!\!\left[\theta \cdot \mathsf{PEq}_{x:\tau'_x \to \tau'}\ \{e_l\}\ \{e_r\}\right]\!\!\right]$. By the assumption we have
(8) $\vdash_B e :: \mathsf{PBEq}_{\lfloor \theta \cdot (x:\tau_x \to \tau) \rfloor}$
(9) $e \hookrightarrow^* \mathsf{xEq\_}\ (\theta \cdot e_l)\ (\theta \cdot e_r)\ e_{pf}$
(10) $(\theta \cdot e_l), (\theta \cdot e_r) \in [\![\theta \cdot (x{:}\tau_x \to \tau)]\!]$
(11) $\forall e_x \in [\![\theta \cdot \tau_x]\!] .e_{pf}\ e_x \in \left[\!\!\left[\mathsf{PEq}_{\theta \cdot (\tau[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}\right]\!\!\right]$
Since (8) only depends on the structure of the type index, we get
(12) $\vdash_B e :: \mathsf{PBEq}_{\lfloor \theta \cdot (x:\tau'_x \to \tau') \rfloor}$
By (6) and (10) we get
(13) $(\theta \cdot e_l), (\theta \cdot e_r) \in \left[\!\!\left[\theta \cdot (x{:}\tau'_x \to \tau')\right]\!\!\right]$
By (4), (5), Lemma B.7, the rule SEq and the IH, we get that $\left[\!\!\left[\mathsf{PEq}_{\theta \cdot (\tau[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}\right]\!\!\right] \subseteq \left[\!\!\left[\mathsf{PEq}_{\theta \cdot (\tau'[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}\right]\!\!\right]$. By which, (11), (7), and reasoning similar to the SFun case, we get
(14) $\forall e_x \in \left[\!\!\left[\theta \cdot \tau'_x\right]\!\!\right] .e_{pf}\ e_x \in \left[\!\!\left[\mathsf{PEq}_{\theta \cdot (\tau'[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}\right]\!\!\right]$
By (12), (9), (13), and (14) we conclude that $e \in \left[\!\!\left[\theta \cdot \mathsf{PEq}_{x:\tau'_x \to \tau'}\ \{e_l\}\ \{e_r\}\right]\!\!\right]$, thus $\Gamma \vdash \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\} \subseteq \mathsf{PEq}_{x:\tau'_x \to \tau'}\ \{e_l\}\ \{e_r\}$.

□

Lemma B.7 (Strengthening). *If* $\Gamma_1 \vdash \tau_1 \preceq \tau_2$*, then:*
(1) If $\Gamma_1, x : \tau_2, \Gamma_2 \vdash e :: \tau$ *then* $\Gamma_1, x : \tau_1, \Gamma_2 \vdash e :: \tau$*.*
(2) If $\Gamma_1, x : \tau_2, \Gamma_2 \vdash \tau \preceq \tau'$ *then* $\Gamma_1, x : \tau_1, \Gamma_2 \vdash \tau \preceq \tau'$*.*
(3) If $\Gamma_1, x : \tau_2, \Gamma_2 \vdash \tau$ *then* $\Gamma_1, x : \tau_1, \Gamma_2 \vdash \tau$*.*
(4) If $\vdash \Gamma_1, x : \tau_2, \Gamma_2$ *then* $\vdash \Gamma_1, x : \tau_1, \Gamma_2$*.*

Proof. The proofs go by induction. Only the TVar case is insteresting; we use TSub and our assumption. □

Lemma B.8 (Semantic typing is closed under parallel reduction in expressions). *If* $e_1 \rightrightarrows^* e_2$*, then* $e_1 \in [\![\tau]\!]$ *iff* $e_2 \in [\![\tau]\!]$*.*

Proof. By induction on τ, using parallel reduction as a bisimulation (Lemma C.5 and Corollary C.15). □

Lemma B.9 (Semantic typing is closed under parallel reduction in types). *If* $\tau_1 \rightrightarrows^* \tau_2$ *then* $[\![\tau_1]\!] = [\![\tau_2]\!]$*.*

Proof. By induction on τ_1 (which necessarily has the same shape as τ_2). We use parallel reduction as a bisimulation (Lemma C.5 and Corollary C.15). □

Lemma B.10 (Parallel reducing types are equal). *If* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$ *and* $\tau_1 \rightrightarrows^* \tau_2$ *then* $\Gamma \vdash \tau_1 \preceq \tau_2$ *and* $\Gamma \vdash \tau_1 \preceq \tau_2$*.*

Proof. By induction on the parallel reduction sequence; for a single step, by induction on τ_1 (which must have the same structure as τ_2). We use parallel reduction as a bisimulation (Lemma C.5 and Corollary C.15). □

Lemma B.11 (Regularity). *(1) If* $\Gamma \vdash e :: \tau$ *then* $\vdash \Gamma$ *and* $\Gamma \vdash \tau$*.*
(2) If $\Gamma \vdash \tau$ *then* $\vdash \Gamma$*.*
(3) If $\Gamma \vdash \tau_1 \preceq \tau_2$ *then* $\vdash \Gamma$ *and* $\Gamma \vdash \tau_1$ *and* $\Gamma \vdash \tau_2$*.*

Proof. By a big ol' induction. □

Lemma B.12 (Canonical forms). *If* $\Gamma \vdash v :: \tau$*, then:*

- *If* $\tau = \{x{:}b \mid e\}$*, then* $v = c$ *such that* $\mathsf{TyCon}(c) = b$ *and* $\Gamma \vdash \mathsf{TyCon}(c) \preceq \{x{:}b \mid e\}$*.*
- *If* $\tau = x{:}\tau_x \to \tau'$*, then* $v = \textit{TLam}x\tau'_x e$ *such that* $\Gamma \vdash \tau_x \preceq \tau'_x$ *and* $\Gamma, x : \tau'_x \vdash e :: \tau''$ *such that* $\tau'' \vdash \tau' \preceq$ *.*
- *If* $\tau = \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$ *then* $v = \mathsf{bEq}_b\ e_l\ e_r\ v_p$ *such that* $\Gamma \vdash e_l :: \tau_l$ *and* $\Gamma \vdash e_r :: \tau_r$ *(for some* τ_l *and* τ_r *that are refinements of* b*) and* $\Gamma, r : \tau_r, l : \tau_l \vdash v_p :: \{x{:}() \mid l ==_b r\}$*.*
- *If* $\tau = \mathsf{PEq}_{x:\tau_x \to \tau'}\ \{e_l\}\ \{e_r\}$ *then* $v = \mathsf{xEq}_{x:\tau'_x \to \tau''}\ e_l\ e_r\ v_p$ *such that* $\Gamma \vdash \tau_x \preceq \tau'_x$ *and* $\Gamma, x : \tau_x \vdash \tau'' \preceq \tau'$ *and* $\Gamma \vdash e_l :: \tau_l$ *and* $\Gamma \vdash e_r :: \tau_r$ *(for some* τ_l *and* τ_r *that are subtypes of* $x{:}\tau'_x \to \tau''$*) and* $\Gamma, r : \tau_r, l : \tau_l \vdash v_p :: x{:}\tau'_x \to \mathsf{PEq}_{\tau''}\ \{e_l\ x\}\ \{e_r\ x\}$*.*

B.4 The Binary Logical Relation

Theorem B.13 (EqRT soundness). *If* $\Gamma \vdash e :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$*, then* $\Gamma \vdash e_1 \sim e_2 :: \tau$*.*

Proof. By $\Gamma \vdash e :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ and the Fundamental Property B.22 we have $\Gamma \vdash e \sim e :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$. Thus, for a fixed $\delta \in \Gamma$, $\delta_1 \cdot e \sim \delta_2 \cdot e :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}; \delta$. By the definition of the logical relation for EqRT, we have $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau; \delta$. So, $\Gamma \vdash e_1 \sim e_2 :: \tau$. □

Lemma B.14 (LR respects subtyping). *If* $\Gamma \vdash e_1 \sim e_2 :: \tau$ *and* $\Gamma \vdash \tau \preceq \tau'$*, then* $\Gamma \vdash e_1 \sim e_2 :: \tau'$*.*

Proof. By induction on the derivation of the subtyping tree.

SBase By assumption we have

(1) $\Gamma \vdash e_1 \sim e_2 \ :: \ \{x{:}b \mid r\}$

(2) $\Gamma \vdash \{x{:}b \mid r\} \ \preceq \ \{x'{:}b \mid r'\}$

By inversion on (2) we get

(3) $\forall \theta \in [\![\Gamma]\!],\ [\![\theta \cdot \{x{:}b \mid r\}]\!] \subseteq [\![\theta \cdot \{x'{:}b \mid r'\}]\!]$

We fix $\delta \in \Gamma$. By (1) we get

(4) $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \{x{:}b \mid r\}; \delta$

By the definition of logical relations:

(5) $\delta_1 \cdot e_1 \hookrightarrow^* v_1$

(6) $\delta_2 \cdot e_2 \hookrightarrow^* v_2$

(7) $v_1 \sim v_2 :: \{x{:}b \mid r\}; \delta$

By (7) and the definition of the logical relation on basic types we have

(8) $v_1 = v_2 = c$

(9) $\vdash_B c \ :: \ b$

(10) $\delta_1 \cdot r[c/x] \hookrightarrow^* \texttt{true}$

(11) $\delta_2 \cdot r[c/x] \hookrightarrow^* \texttt{true}$

By (3), (10) and (11) become

(12) $\delta_1 \cdot r'[c/x'] \hookrightarrow^* \texttt{true}$

(13) $\delta_2 \cdot r'[c/x'] \hookrightarrow^* \texttt{true}$

By (8), (9), (12), and (13) we get

(14) $v_1 \sim v_2 :: \{x'{:}b \mid r'\}; \delta$

By (5), (6), and (14) we have

(15) $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \{x'{:}b \mid r'\}; \delta$

Thus, $\Gamma \vdash e_1 \sim e_2 \ :: \ \{x'{:}b \mid r'\}$.

SFun By assumption:

(1) $\Gamma \vdash e_1 \sim e_2 \ :: \ x{:}\tau_x \to \tau$

(2) $\Gamma \vdash x{:}\tau_x \to \tau \ \preceq \ x{:}\tau'_x \to \tau'$

By inversion of the rule (2)

(3) $\Gamma \vdash \tau'_x \ \preceq \ \tau_x$

(4) $\Gamma, x : \tau'_x \vdash \tau \ \preceq \ \tau'$

We fix $\delta \in \Gamma$. By (1) and the definition of logical relation

(5) $\delta_1 \cdot e_1 \hookrightarrow^* v_1$

(6) $\delta_2 \cdot e_2 \hookrightarrow^* v_2$

(7) $v_1 \sim v_2 :: x{:}\tau_x \to \tau; \delta$

We fix v'_1 and v'_2 so that

(8) $v'_1 \sim v'_2 :: \tau'_x; \delta$

By (8) and the definition of logical relations, since the values are idempotent under substitution, we have

(9) $\Gamma \vdash v'_1 \sim v'_2 \ :: \ \tau'_x$

By (9) and inductive hypothesis on (3) we have

(10) $\Gamma \vdash v'_1 \sim v'_2 \ :: \ \tau_x$

By (10), idempotence of values under substitution, and the definition of logical relations, we have

(11) $v'_1 \sim v'_2 :: \tau_x; \delta$

By (7), (11), and the definition of logical relations on function values:

(12) $v_1\ v'_1 \sim v_2\ v'_2 :: \tau; \delta, (v'_1, v'_2)/x$

By (9), (12), and the definition of logical relations we have

(12) $\Gamma, x : \tau'_x \vdash v_1\ v'_1 \sim v_2\ v'_2 \ :: \ \tau$

By (12) and inductive hypothesis on (4) we have
(13) $\Gamma, x : \tau'_x \vdash v_1\ v'_1 \sim v_2\ v'_2\ ::\ \tau'$
By (8), (13), and the definition of logical relations, we have
(14) $v_1\ v'_1 \sim v_2\ v'_2 :: \tau'; \delta, (v'_1, v'_2)/x$
By (8), (14), and the definition of logical relations, we have
(15) $v_1 \sim v_2 :: x{:}\tau'_x \to \tau'; \delta$
By (5), (6), and (15), we get
(16) $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: x{:}\tau'_x \to \tau'; \delta$
So, $\Gamma \vdash e_1 \sim e_2\ ::\ x{:}\tau'_x \to \tau'$.

SEQ By hypothesis:
(1) $\Gamma \vdash e_1 \sim e_2\ ::\ \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}$
(2) $\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}\ \preceq\ \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}$
We fix $\delta \in \Gamma$. By (1)
(3) $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}; \delta$
By (3) and the definition of logical relations.
(4) $\delta_1 \cdot e_1 \hookrightarrow^* v_1$
(5) $\delta_2 \cdot e_2 \hookrightarrow^* v_2$
(6) $v_1 \sim v_2 :: \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}; \delta$
By (6) and the definition of logical relations
(7) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: \tau; \delta$
By (7) and the definition of logical relations.
(8) $\Gamma \vdash e_l \sim e_r\ ::\ \tau$
By inversion on (2)
(9) $\Gamma \vdash \tau\ \preceq\ \tau'$
(10) $\Gamma \vdash \tau'\ \preceq\ \tau$
By (8) and inductive hypothesis on (9)
(11) $\Gamma \vdash e_l \sim e_r\ ::\ \tau'$
Thus,
(12) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: \tau'; \delta$
By (12), (4), (5), and determinism of operational semantics:
(12) $v_1 \sim v_2 :: \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}; \delta$
By (4), (5), and (13)
(14) $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}; \delta$
So, by definition of logical relations, $\Gamma \vdash e_1 \sim e_2\ ::\ \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}$.

□

LEMMA B.15 (CONSTANT SOUNDNESS). $\Gamma \vdash c \sim c\ ::\ \mathsf{TyCon}(c)$

PROOF. The proof follows the same steps as Theorem B.1. □

LEMMA B.16 (SELFIFICATION OF CONSTANTS). *If* $\Gamma \vdash e \sim e\ ::\ \{z{:}b \mid r\}$ *then* $\Gamma \vdash x \sim x\ ::\ \{z{:}b \mid z ==_b x\}$.

PROOF. We fix $\delta \in \Gamma$. By hypothesis $(v_1, v_2)/x \in \delta$ with $v_1 \sim v_2 :: \{z{:}b \mid r\}; \delta$. We need to show that $\delta_1 \cdot x \sim \delta_2 \cdot x :: \{z{:}b \mid z ==_b x\}; \delta$. Which reduces to $v_1 \sim v_2 :: \{z{:}b \mid z ==_b x\}; \delta$. By the definition on the logical relation on basic values, we know $v_1 = v_2 = c$ and $\vdash_B c\ ::\ b$. Thus, we are left to prove that $\delta_1 \cdot ((z ==_b x)[c/z]) \hookrightarrow^* \mathtt{true}$ and $\delta_2 \cdot ((z ==_b x)[c/z]) \hookrightarrow^* \mathtt{true}$ which, both, trivially hold by the definition of $==_b$. □

LEMMA B.17 (VARIABLE SOUNDNESS). *If* $x : \tau \in \Gamma$, *then* $\Gamma \vdash x \sim x\ ::\ \tau$.

Proof. By the definition of the logical relation it suffices to show that $\forall \delta \in \Gamma . \delta_1(x) \sim \delta_2(x) :: \tau; \delta$; which is trivially true by the definition of $\delta \in \Gamma$. □

Lemma B.18 (Transitivity of Evaluation). *If $e \hookrightarrow^* e'$, then $e \hookrightarrow^* v$ iff $e' \hookrightarrow^* v$.*

Proof. Assume $e \hookrightarrow^* v$. Since the $\hookrightarrow$ is by definition deterministic, there exists a unique sequence $e \hookrightarrow e_1 \hookrightarrow \ldots \hookrightarrow e_i \hookrightarrow \ldots \hookrightarrow v$. By assumption, $e \hookrightarrow^* e'$, so there exists a j, so $e' \equiv e_j$, and $e' \hookrightarrow^* v$ following the same sequence.

Assume $e' \hookrightarrow^* v$. Then $e \hookrightarrow^* e' \hookrightarrow^* v$ uniquely evaluates e to v. □

Lemma B.19 (LR closed under evaluation). *If $e_1 \hookrightarrow^* e_1'$, $e_2 \hookrightarrow^* e_2'$, then $e_1' \sim e_2' :: \tau; \delta$ iff $e_1 \sim e_2 :: \tau; \delta$.*

Proof. Assume $e_1' \sim e_2' :: \tau; \delta$, by the definition of the logical relation on closed terms we have $e_1' \hookrightarrow^* v_1$, $e_2' \hookrightarrow^* v_2$, and $v_1 \sim v_2 :: \tau; \delta$. By Lemma B.18 and by assumption, $e_1 \hookrightarrow^* e_1'$ and $e_2 \hookrightarrow^* e_2'$, we have $e_1 \hookrightarrow^* v_1$ and $e_2 \hookrightarrow^* v_2$. By which and $v_1 \sim v_2 :: \tau; \delta$ we get that $e_1 \sim e_2 :: \tau; \delta$. The other direction is identical. □

Lemma B.20 (LR closed under parallel reduction). *If $e_1 \Rrightarrow^* e_1'$, $e_2 \Rrightarrow^* e_2'$, and $e_1' \sim e_2' :: \tau; \delta$, then $e_1 \sim e_2 :: \tau; \delta$.*

Proof. By induction on τ, using parallel reduction as a backward simulation (Corollary C.15). □

Lemma B.21 (LR Compositionality). *If $\delta_1 \cdot e_x \hookrightarrow^* v_{x_1}$, $\delta_2 \cdot e_x \hookrightarrow^* v_{x_2}$, $e_1 \sim e_2 :: \tau; \delta, (v_{x_1}, v_{x_2})/x$, then $e_1 \sim e_2 :: \tau[e_x/x]; \delta$.*

Proof. By the assumption we have that

(1) $\delta_1 \cdot e_x \hookrightarrow^* v_{x_1}$
(2) $\delta_2 \cdot e_x \hookrightarrow^* v_{x_2}$
(3) $e_1 \hookrightarrow^* v_1$
(4) $e_2 \hookrightarrow^* v_2$
(5) $v_1 \sim v_2 :: \tau; \delta, (v_{x1}, v_{x_2})/x$

and we need to prove that $v_1 \sim v_2 :: \tau[e_x/x]; \delta$. The proof goes by structural induction on the type τ.

- $\tau \doteq \{z{:}b \mid r\}$. For $i = 1, 2$ we need to show that if $\delta_i, [v_{x_i}/x] \cdot r[v_i/z] \hookrightarrow^* \texttt{true}$ then $\delta_i \cdot r[v_i/z][e_i/x] \hookrightarrow^* \texttt{true}$. We have $\delta_i, [v_{x_i}/x] \cdot r[v_i/z] \Rrightarrow^* \delta_i \cdot r[v_i/z][e_i/x]$ because substituting parallel reducing terms parallel reduces (Corollary C.3) and parallel reduction subsumes reduction (Lemma C.4). By cotermination at constants (Corollary C.17), we have $\delta_i \cdot r[v_i/z][e_i/x] \hookrightarrow^* \texttt{true}$.
- $\tau \doteq y{:}\tau_y' \to \tau'$. We need to show that if $v_1 \sim v_2 :: y{:}\tau_y' \to \tau'; \delta, (v_{x_1}, v_{x_2})/x$, then $v_1 \sim v_2 :: y{:}\tau_y' \to \tau'[e_x/x]; \delta$.
  We fix v_{y_1} and v_{y_2} so that $v_{y_1} \sim v_{y_2} :: \tau_y'; \delta, (v_{x_1}, v_{x_2})/x$.
  Then, we have that $v_1\ v_{y_1} \sim v_2\ v_{y_2} :: \tau'; \delta, (v_{x_1}, v_{x_2})/x, (v_{y_1}, v_{y_2})/y$.
  By inductive hypothesis, we have that $v_1\ v_{y_1} \sim v_2\ v_{y_2} :: \tau'[e_x/x]; \delta, (v_{y_1}, v_{y_2})/y$.
  By inductive hypothesis on the fixed arguments, we also get $v_{y_1} \sim v_{y_2} :: \tau_y'[e_x/x]; \delta$.
  Combined, we get $v_1 \sim v_2 :: y{:}\tau_y' \to \tau'[e_x/x]; \delta$.
- $\tau \doteq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}$. We need to show that if $v_1 \sim v_2 :: \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}; \delta, (v_{x_1}, v_{x_2})/x$, then $v_1 \sim v_2 :: \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}[e_x/x]; \delta$.
  This reduces to showing that if $\delta_1, [v_{x_1}/x] \cdot e_l \sim \delta_2, [v_{x_2}/x] \cdot e_r :: \tau'; \delta$, then $\delta_1 \cdot e_l[e_x/x] \sim \delta_2 \cdot e_r[e_x/x] :: \tau'; \delta$; we find $\delta_1 \cdot e_l[e_x/x] \Rrightarrow^* \delta_1, [v_{x_1}/x] \cdot e_l$ and $\delta_2 \cdot e_r[e_x/x] \Rrightarrow^* \delta_2, [v_{x_2}/x] \cdot e_r$

because substituting multiple parallel reduction is parallel reduction (Corollary C.3). The logical relation is closed under parallel reduction (Lemma B.20), and so $\delta_1 \cdot e_l[e_x/x] \sim \delta_2 \cdot e_r[e_x/x] :: \tau'; \delta$.

□

THEOREM B.22 (LR FUNDAMENTAL PROPERTY). *If* $\Gamma \vdash e :: \tau$, *then* $\Gamma \vdash e \sim e \ :: \ \tau$.

PROOF. The proof goes by induction on the derivation tree:

TSUB By inversion of the rule we have

(1) $\Gamma \vdash e :: \tau'$

(2) $\Gamma \vdash \tau' \ \preceq \ \tau$

By IH on (1) we have

(3) $\Gamma \vdash e \sim e \ :: \ \tau'$

By (3), (4), and Lemma B.14 we have $\Gamma \vdash e \sim e \ :: \ \tau$.

TCON By Lemma B.15.

TSELF By inversion of the rule, we have:

(1) $\Gamma \vdash e :: \{z{:}b \mid r\}$.

(2) By the IH on (1), we have:
$\Gamma \vdash e \sim e \ :: \ \{z{:}b \mid r\}$.

(3) We fix a δ such that:
$\delta \in \Gamma$ and
$\delta_1 \cdot e \sim \delta_2 \cdot e :: \{z{:}b \mid r\}; \delta$

(4) There must exist v_1 and v_2 such that:
$\delta_1 \cdot e \hookrightarrow^* v_1$
$\delta_2 \cdot e \hookrightarrow^* v_2$
$v_1 \sim v_2 :: \{z{:}b \mid r\}; \delta$

(5) By definition, $v_1 = v_2 = c$ such that:
$\vdash_B c \ :: \ b$
$\delta_1 \cdot r[c/x] \hookrightarrow^* \texttt{true}$
$\delta_2 \cdot r[c/x] \hookrightarrow^* \texttt{true}$

(6) We find $v_1 \sim v_2 :: \{z{:}b \mid z ==_b e\}; \delta$, because:
$\vdash_B c \ :: \ b$ by (5)
$\delta_1 \cdot (z ==_b e)[c/z] \hookrightarrow^* \texttt{true}$ because $\delta_1 \cdot e \hookrightarrow^* v_1 = c$ by (4)
$\delta_2 \cdot (z ==_b e)[c/z] \hookrightarrow^* \texttt{true}$ because $\delta_2 \cdot e \hookrightarrow^* v_2 = c$ by (4)

TVAR By inversion of the rule and Lemma B.17.

TLAM By hypothesis:

(1) $\Gamma \vdash \lambda x{:}\tau_x.\, e :: x{:}\tau_x \to \tau$

By inversion of the rule we have

(2) $\Gamma, x : \tau_x \vdash e :: \tau$

(3) $\Gamma \vdash \tau_x$

By inductive hypothesis on (2) we have

(4) $\Gamma, x : \tau_x \vdash e \sim e \ :: \ \tau$

We fix a δ, v_{x_1}, and v_{x_2} so that

(5) $\delta \in \Gamma$

(6) $v_{x_1} \sim v_{x_2} :: \tau_x; \delta$

Let $\delta' \doteq \delta, (v_{x_1}, v_{x_2})/x$.

By the definition of the logical relation on open terms, (4), (5), and (6) we have

(7) $\delta'_1 \cdot e \sim \delta'_2 \cdot e :: \tau; \delta'$

By the definition of substitution

(8) $\delta_1 \cdot e[v_{x_1}/x] \sim \delta_2 \cdot e[v_{x_2}/x] :: \tau; \delta'$

By the definition of the logical relation on closed expressions

(9) $\delta_1 \cdot e[v_{x_1}/x] \hookrightarrow^* v_1$, $\delta_2 \cdot e[v_{x_2}/x] \hookrightarrow^* v_2$, and $v_1 \sim v_2 :: \tau; \delta'$

By the definition and determinism of operational semantics

(10) $\delta_1 \cdot (\lambda x{:}\tau_x.\ e)\ v_{x_1} \hookrightarrow^* v_1$, $\delta_2 \cdot (\lambda x{:}\tau_x.\ e)\ v_{x_2} \hookrightarrow^* v_2$, and $v_1 \sim v_2 :: \tau; \delta'$

By (6) and the definition of logical relation on function values,

(11) $\delta_1 \cdot \lambda x{:}\tau_x.\ e \sim \delta_2 \cdot \lambda x{:}\tau_x.\ e :: x{:}\tau_x \to \tau; \delta$

Thus, by the definition of the logical relation, $\Gamma \vdash \lambda x{:}\tau_x.\ e \sim \lambda x{:}\tau_x.\ e\ ::\ x{:}\tau_x \to \tau$

TApp By hypothesis:

(1) $\Gamma \vdash e\ e_x :: \tau[e_x/x]$

By inversion we get

(2) $\Gamma \vdash e :: x{:}\tau_x \to \tau$

(3) $\Gamma \vdash e_x :: \tau_x$

By inductive hypothesis

(3) $\Gamma \vdash e \sim e\ ::\ x{:}\tau_x \to \tau$

(4) $\Gamma \vdash e_x \sim e_x\ ::\ \tau_x$

We fix a $\delta \in \Gamma$. Then, by the definition of the logical relation on open terms

(5) $\delta_1 \cdot e \sim \delta_2 \cdot e :: (x{:}\tau_x \to \tau); \delta$

(6) $\delta_1 \cdot e_x \sim \delta_2 \cdot e_x :: \tau_x; \delta$

By the definition of the logical relation on open terms:

(7) $\delta_1 \cdot e \hookrightarrow^* v_1$

(8) $\delta_2 \cdot e \hookrightarrow^* v_2$

(9) $v_1 \sim v_2 :: x{:}\tau_x \to \tau; \delta$

(10) $\delta_1 \cdot e_x \hookrightarrow^* v_{x_1}$

(11) $\delta_2 \cdot e_x \hookrightarrow^* v_{x_2}$

(12) $v_{x_1} \sim v_{x_2} :: \tau_x; \delta$

By (7) and (10)

(13) $\delta_1 \cdot e\ e_x \hookrightarrow^* v_1\ v_{x_1}$

By (8) and (11)

(14) $\delta_2 \cdot e\ e_x \hookrightarrow^* v_2\ v_{x_2}$

By (9), (12), and the definition of logical relation on functions:

(15) $v_1\ v_{x_1} \sim v_2\ v_{x_2} :: \tau; \delta, (v_{x_1}, v_{x_2})/x$

By (13), (14), (15), and Lemma B.19

(16) $\delta_1 \cdot e\ e_x \sim \delta_2 \cdot e\ e_x :: \tau; \delta, (v_{x_1}, v_{x_2})/x$

By (10), (11), (16), and Lemma B.21

(17) $\delta_1 \cdot e\ e_x \sim \delta_2 \cdot e\ e_x :: \tau[e_x/x]; \delta$

So from the definition of logical relations, $\Gamma \vdash e\ e_x \sim e\ e_x\ ::\ \tau[e_x/x]$.

TEqBase By hypothesis:

(1) $\Gamma \vdash \mathsf{bEq}_b\ e_l\ e_r\ e :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$

By inversion of the rule:

(2) $\Gamma \vdash e_l :: \tau_r$

(3) $\Gamma \vdash e_r :: \tau_l$

(4) $\Gamma \vdash \tau_r\ \preceq\ b$

(5) $\Gamma \vdash \tau_l\ \preceq\ b$

(6) $\Gamma, r : \tau_r, l : \tau_l \vdash e :: \{x{:}()\ |\ l ==_b r\}$

By inductive hypothesis on (2), (3), and (6) we have

(7) $\Gamma \vdash e_l \sim e_l\ ::\ \tau_r$

(8) $\Gamma \vdash e_r \sim e_r \ :: \ \tau_l$
(9) $\Gamma, r : \tau_r, l : \tau_l \vdash e \sim e \ :: \ \{x{:}() \mid l ==_b r\}$
We fix $\delta \in \Gamma$. Then (7) and (8) become
(10) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_l :: \tau_r; \delta$
(11) $\delta_1 \cdot e_r \sim \delta_2 \cdot e_r :: \tau_l; \delta$
By the definition of the logical relation on closed terms:
(12) $\delta_1 \cdot e_l \hookrightarrow^* v_{l_1}$
(13) $\delta_2 \cdot e_l \hookrightarrow^* v_{l_2}$
(14) $v_{l_1} \sim v_{l_2} :: \tau_l; \delta$
(15) $\delta_1 \cdot e_r \hookrightarrow^* v_{r_1}$
(16) $\delta_2 \cdot e_r \hookrightarrow^* v_{r_2}$
(17) $v_{r_1} \sim v_{r_2} :: \tau_r; \delta$
We define $\delta' \doteq \delta, (v_{r_1}, v_{r_2})/r, (v_{l_1}, v_{l_2})/l$.
By (9), (14), and (17) we have
(18) $\delta'_1 \cdot e \sim \delta'_2 \cdot e :: \{x{:}() \mid l ==_b r\}; \delta'$
By the definition of the logical relation on closed terms:
(19) $\delta' \cdot e \hookrightarrow^* v_1$
(20) $\delta' \cdot e \hookrightarrow^* v_2$
(21) $v_1 \sim v_2 :: \{x{:}() \mid l ==_b r\}; \delta'$
By (21) and the definition of logical relation on basic values:
(19) $\delta'_1 \cdot (l ==_b r) \hookrightarrow^* \texttt{true}$
(20) $\delta'_2 \cdot (l ==_b r) \hookrightarrow^* \texttt{true}$
By the definition of $==_b$
(21) $v_{l_1} = v_{r_1}$
(22) $v_{l_2} = v_{r_2}$
By (14) and (17) and since τ_l and τ_r are basic types
(23) $v_{l_1} = v_{l_2}$
(24) $v_{r_1} = v_{r_2}$
By (21) and (24)
(25) $v_{l_1} = v_{r_2}$
By the definition of the logical relation on basic types
(26) $v_{l_1} \sim v_{r_2} :: b; \delta$
By which, (12), (16), and Lemma B.19
(27) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: b; \delta$
By (12), (15), and (19)
(28) $\delta_1 \cdot \mathsf{bEq}_b\ e_l\ e_r\ e \hookrightarrow^* \mathsf{bEq}_b\ v_{l_1}\ v_{r_1}\ v_1$
By (13), (16), and (20)
(29) $\delta_2 \cdot \mathsf{bEq}_b\ e_l\ e_r\ e \hookrightarrow^* \mathsf{bEq}_b\ v_{l_2}\ v_{r_2}\ v_2$
By (27) and the definition of the logical relation on EqRT
(30) $\mathsf{bEq}_b\ v_{l_1}\ v_{r_1}\ v_1 \sim \mathsf{bEq}_b\ v_{l_2}\ v_{r_2}\ v_2 :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}; \delta$.
By (28), (29), and (30)
(31) $\delta_1 \cdot \mathsf{bEq}_b\ e_l\ e_r\ e \sim \delta_2 \cdot \mathsf{bEq}_b\ e_l\ e_r\ e :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}; \delta$.
So, by the definition on the logical relation, $\Gamma \vdash \mathsf{bEq}_b\ e_l\ e_r\ e \sim \mathsf{bEq}_b\ e_l\ e_r\ e \ :: \ \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$.

TEqFun By hypothesis
(1) $\Gamma \vdash \mathsf{xEq}_{\tau_x:\tau\to}\ e_l\ e_r\ e :: \mathsf{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}$
By inversion of the rule
(2) $\Gamma \vdash e_l :: \tau_r$
(3) $\Gamma \vdash e_r :: \tau_l$

(4) $\Gamma \vdash \tau_r \ \leq \ x{:}\tau_x \rightarrow \tau$
(5) $\Gamma \vdash \tau_l \ \leq \ x{:}\tau_x \rightarrow \tau$
(6) $\Gamma, r : \tau_r, l : \tau_l \vdash e :: (x{:}\tau_x \rightarrow \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\})$
(7) $\Gamma \vdash x{:}\tau_x \rightarrow \tau$
By inductive hypothesis on (2), (3), and (6) we have
(8) $\Gamma \vdash e_l \sim e_l \ :: \ \tau_r$
(9) $\Gamma \vdash e_r \sim e_r \ :: \ \tau_l$
(10) $\Gamma, r : \tau_r, l : \tau_l \vdash e \sim e \ :: \ (x{:}\tau_x \rightarrow \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\})$
By (8), (9), and Lemma B.14
(11) $\Gamma \vdash e_l \sim e_l \ :: \ x{:}\tau_x \rightarrow \tau$
(12) $\Gamma \vdash e_r \sim e_r \ :: \ x{:}\tau_x \rightarrow \tau$
We fix $\delta \in \Gamma$. Then (11), and (12) become
(13) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_l :: x{:}\tau_x \rightarrow \tau; \delta$
(14) $\delta_1 \cdot e_r \sim \delta_2 \cdot e_r :: x{:}\tau_x \rightarrow \tau; \delta$
By the definition of the logical relation on closed terms:
(15) $\delta_1 \cdot e_l \hookrightarrow^* v_{l_1}$
(16) $\delta_2 \cdot e_l \hookrightarrow^* v_{l_2}$
(17) $v_{l_1} \sim v_{l_2} :: x{:}\tau_x \rightarrow \tau; \delta$
(18) $v_{l_1} \sim v_{l_2} :: \tau_l; \delta$
(19) $\delta_1 \cdot e_r \hookrightarrow^* v_{r_1}$
(20) $\delta_2 \cdot e_r \hookrightarrow^* v_{r_2}$
(21) $v_{r_1} \sim v_{r_2} :: x{:}\tau_x \rightarrow \tau; \delta$
(22) $v_{r_1} \sim v_{r_2} :: \tau_r; \delta$
We fix v_{x_1} and v_{x_2} so that $v_{x_1} \sim v_{x_2} :: \tau_x; \delta$. Let $\delta_x \doteq \delta, (v_{x_1}, v_{x_2})/x$.
By the definition on the logical relation on function values, (17) and (21) become
(23) $v_{l_1}\ v_{x_1} \sim v_{l_2}\ v_{x_2} :: \tau; \delta_x$
(24) $v_{r_1}\ v_{x_1} \sim v_{r_2}\ v_{x_2} :: \tau; \delta_x$
Let $\delta_{lr} \doteq \delta, (v_{r_1}, v_{r_2})/r, (v_{l_1}, v_{l_2})/l$.
By the definition of the logical relation on closed terms, (10) becomes:
(25) $\delta_{lr} \cdot e \hookrightarrow^* v_1$
(26) $\delta_{lr} \cdot e \hookrightarrow^* v_2$
(27) $v_1 \sim v_2 :: x{:}\tau_x \rightarrow \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\}; \delta_{lr}$
By (27) and the definition of logical relation on function values:
(28) $v_1\ v_{x_1} \sim v_2\ v_{x_2} :: \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\}; \delta_{lr}, (v_{x_1}, v_{x_2})/x$
By the definition of the logical relation on EqRT
(29) $v_{l_1}\ v_{x_1} \sim v_{r_2}\ v_{x_2} :: \tau; \delta_{lr}, (v_{x_1}, v_{x_2})/x$
By the definition of logical relations on function values
(30) $v_{l_1} \sim v_{r_2} :: x{:}\tau_x \rightarrow \tau; \delta_{lr}$
By (7), l and r do not appear free in the relation, so
(31) $v_{l_1} \sim v_{r_2} :: x{:}\tau_x \rightarrow \tau; \delta$
By which, (15), (20), and Lemma B.19
(32) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: x{:}\tau_x \rightarrow \tau; \delta$
By (15), (19), and (25)
(33) $\delta_1 \cdot \mathsf{xEq}_{\tau_x:\tau\rightarrow}\ e_l\ e_r\ e \hookrightarrow^* \mathsf{xEq}_{\tau_x:\tau\rightarrow}\ v_{l_1}\ v_{r_1}\ v_1$
By (16), (20), and (26)
(34) $\delta_2 \cdot \mathsf{xEq}_{\tau_x:\tau\rightarrow}\ e_l\ e_r\ e \hookrightarrow^* \mathsf{xEq}_{\tau_x:\tau\rightarrow}\ v_{l_2}\ v_{r_2}\ v_2$
By (32) and the definition of the logical relation on EqRT
(35) $\mathsf{xEq}_{\tau_x:\tau\rightarrow}\ v_{l_1}\ v_{r_1}\ v_1 \sim \mathsf{xEq}_{\tau_x:\tau\rightarrow}\ v_{l_2}\ v_{r_2}\ v_2 :: \mathsf{PEq}_{x:\tau_x\rightarrow\tau}\ \{e_l\}\ \{e_r\}; \delta.$

By (33), (34), and (35)

(36) $\delta_1 \cdot \mathsf{xEq}_{\tau_x:\tau\to}\ e_l\ e_r\ e \sim \delta_2 \cdot \mathsf{xEq}_{\tau_x:\tau\to}\ e_l\ e_r\ e :: \mathsf{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}; \delta$.

So, by the definition on the logical relation, $\Gamma \vdash \mathsf{xEq}_{\tau_x:\tau\to}\ e_l\ e_r\ e \sim \mathsf{xEq}_{\tau_x:\tau\to}\ e_l\ e_r\ e :: \mathsf{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}$.

□

B.5 The Logical Relation and the EqRT Type are Equivalence Relations

Theorem B.23 (The logical relation is an equivalence relation). *$\Gamma \vdash e_1 \sim e_2 :: \tau$ is reflexive, symmetric, and transivite.*

- *Reflexivity: If $\Gamma \vdash e :: \tau$, then $\Gamma \vdash e \sim e :: \tau$.*
- *Symmetry: If $\Gamma \vdash e_1 \sim e_2 :: \tau$, then $\Gamma \vdash e_2 \sim e_1 :: \tau$.*
- *Transitivity: If $\Gamma \vdash e_2 :: \tau$ and $\Gamma \vdash e_1 \sim e_2 :: \tau$ and $\Gamma \vdash e_2 \sim e_3 :: \tau$, then $\Gamma \vdash e_1 \sim e_3 :: \tau$.*

Proof. **Reflexivity:** This is exactly the Fundamental Property B.22.

Symmetry: Let $\bar{\delta}$ be defined such that $\bar{\delta}_1(x) = \delta_2(x)$ and $\bar{\delta}_2(x) = \delta_1(x)$. First, we prove that $v_1 \sim v_2 :: \tau; \delta$ implies $v_2 \sim v_1 :: \tau; \bar{\delta}$, by structural induction on τ.

- $\tau \doteq \{z{:}b \mid r\}$. This case is immediate: we have to show that $c \sim c :: \{z{:}b \mid r\}; \bar{\delta}$ given $c \sim c :: \{z{:}b \mid r\}; \delta$. But the definition in this case is itself symmetric: the predicate goes to `true` under both substitutions.
- $\tau \doteq x{:}\tau'_x \to \tau'$. We fix v_{x_1} and v_{x_2} so that
  (1) $v_{x_1} \sim v_{x_2} :: \tau'_x; \delta$
  By the definition of logical relations on open terms and inductive hypothesis
  (2) $v_{x_2} \sim v_{x_1} :: \tau'_x; \bar{\delta}$
  By the definition on logical relations on functions
  (3) $v_1\ v_{x_1} \sim v_2\ v_{x_2} :: \tau'; \delta, (v_{x_1}, v_{x_2})/x$
  By the definition of logical relations on open terms and since the expressions $v_1\ v_{x_1}$ and $v_2\ v_{x_2}$ are closed, By the inductive hypothesis on τ':
  (4) $v_2\ v_{x_2} \sim v_1\ v_{x_1} :: \tau'; \bar{\delta}, x : \tau'_x$
  By (2) and the definition of logical relations on open terms
  (5) $v_2\ v_{x_2} \sim v_1\ v_{x_1} :: \tau'; \bar{\delta}, (v_{x_2}, v_{x_1})/x$
  By the definition of the logical relation on functions, we conclude that $v_2 \sim v_1 :: x{:}\tau'_x \to \tau'; \bar{\delta}$
- $\tau \doteq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}$. By assumption,
  (1) $v_1 \sim v_2 :: \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}; \delta$
  By the definition of the logical relation on EqRT types
  (2) $\delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: \tau'; \delta$
  i.e., $\delta_1 \cdot (e_l) \hookrightarrow^* v_l$ and similarly for v_r such that $v_l \sim v_r :: \tau'; \delta$.
  By the IH on τ', we have:
  (3) $v_r \sim v_l :: \tau'; \bar{\delta}$
  And so, by the definition of the LR on equality proofs:
  (4) $v_2 \sim v_1 :: \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}; \bar{\delta}$

Next, we show that $\delta \in \Gamma$ implies $\bar{\delta} \in \Gamma$. We go by structural induction on Γ.

- $\Gamma = \cdot$. This case is trivial.
- $\Gamma = \Gamma', x : \tau$. For $x : \tau$, we know that $\delta_1(x) \sim \delta_2(x) :: \tau; \delta$. By the IH on τ, we find $\delta_2(x) \sim \delta_1(x) :: \tau; \bar{\delta}$, which is just the same as $\bar{\delta}_1(x) \sim \bar{\delta}_2(x) :: \tau; \bar{\delta}$. By the IH on Γ', we can use similar reasoning to find $\bar{\delta}_1(y) \sim \bar{\delta}_2(y) :: \tau'; \bar{\delta}$ for all $y : \tau' \in \Gamma'$.

Now, suppose $\Gamma \vdash e_1 \sim e_2 :: \tau$; we must show $\Gamma \vdash e_2 \sim e_1 :: \tau$. We fix $\delta \in \Gamma$; we must show $\delta_1 \cdot e_2 \sim \delta_2 \cdot e_1 :: \tau; \delta$, i.e., there must exist v_1 and v_2 such that $\delta_1 \cdot e_2 \hookrightarrow^* v_2$ and $\delta_2 \cdot e_1 \hookrightarrow^* v_1$ and

$v_2 \sim v_1 :: \tau; \delta$. We have $\delta \in \Gamma$, and so $\bar{\delta} \in \Gamma$ by our second lemma. But then, by assumption, we have v_1 and v_2 such that $\bar{\delta_1} \cdot e_1 \hookrightarrow^* v_1$ and $\bar{\delta_2} \cdot e_2 \hookrightarrow^* v_2$ and $v_1 \sim v_2 :: \tau; \bar{\delta}$. Our first lemma then yields $v_2 \sim v_1 :: \tau; \delta$ as desired.

Transitivity: First, we prove an inner property: if $\delta \in \Gamma$ and $v_1 \sim v_2 :: \tau; \delta$ and $v_2 \sim v_3 :: \tau; \delta$, then $v_1 \sim v_3 :: \tau; \delta$. We go by structural induction on the type index τ.

- $\tau \doteq \{z{:}b \mid r\}$. Here all of the values must be the fixed constant c. Furthermore, we must have $\delta_1 \cdot r[c/x] \hookrightarrow^* \texttt{true}$ and $\delta_2 \cdot r[c/x] \hookrightarrow^* \texttt{true}$, so we can immediately find $v_1 \sim v_3 :: \tau; \delta$.
- $\tau \doteq x{:}\tau'_x \to \tau'$.
  Let $v_l \sim v_r :: \tau'_x; \delta$ be given. We must show that $v_1 \sim v_3 :: \tau; \delta, (v_l, v_r)/x$. We know by assumption that: $v_1\ v_l \sim v_2\ v_r :: \tau'; \delta, (v_l, v_r)/x$ and $v_2\ v_l \sim v_3\ v_r :: \tau'; \delta, (v_l, v_r)/x$. By the IH on τ', we find $v_1\ v_l \sim v_3\ v_r :: \tau'; \delta, (v_l, v_r)/x$; which gives $v_1 \sim v_3 :: \tau; \delta, (v_l, v_r)/x$.
- $\tau \doteq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}$.
  To find $v_1 \sim v_3 :: \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}; \delta$, we merely need to find that $\delta_1 \cdot e_l \sim \delta_2 \cdot e_r :: \tau; \delta$, which we have by inversion on $v_1 \sim v_2 :: \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}; \delta$.

With our proof that the value relation is transitive in hand, we turn our attention to the open relation. Suppose $\Gamma \vdash e_1 \sim e_2 :: \tau$ and $\Gamma \vdash e_2 \sim e_3 :: \tau$; we want to see $\Gamma \vdash e_1 \sim e_3 :: \tau$. Let $\delta \in \Gamma$ be given. We have $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau; \delta$ and $\delta_1 \cdot e_2 \sim \delta_2 \cdot e_3 :: \tau; \delta$. By the definition of the logical relations, we have $\delta_1 \cdot e_1 \hookrightarrow^* v_1$, $\delta_2 \cdot e_2 \hookrightarrow^* v_2$, $\delta_1 \cdot e_2 \hookrightarrow^* v'_2$, $\delta_2 \cdot e_3 \hookrightarrow^* v_3$, $v_1 \sim v_2 :: \tau; \delta$, and $v'_2 \sim v_3 :: \tau; \delta$.

Moreover, we know that e_2 is well typed, so by the fundamental theorem (Theorem B.22), we know that $\Gamma \vdash e_2 \sim e_2 :: \tau$, and so $v_2 \sim v'_2 :: \tau; \delta$.

By our transitivity lemma on the value relation, we can find that v_1 is equivalent to v_2 is equivalent to v'_2 is equivalent to v_3, and so $v_1 \sim v_3 :: \tau; \delta$.

□

$$
\begin{array}{rcl}
\mathsf{pf} & : & e \to e \to \tau \\
\mathsf{pf}(l, r, b) & = & \{x{:}() \mid l ==_b r\} \\
\mathsf{pf}(l, r, x{:}\tau_x \to \tau) & = & x{:}\tau_x \to \mathsf{PEq}_\tau\ \{l\ x\}\ \{r\ x\}
\end{array}
$$

Our propositional equality $\mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}$ is a reflection of the logical relation, so it is unsurprising that it is also an equivalence relation. We can prove that our propositional equality is treated as an equivalence relation by the syntactic type system. There are some tiny wrinkles in the syntactic system: symmetry and transitivity produce normalized proofs, but reflexivity produces unnormalized ones in order to generate the correct invariant types τ_l and τ_r in the base case.

Theorem B.24 (EqRT is an equivalence relation). $\mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ *is reflexive, symmetric, and transitive on equable types. That is, for all* τ *that contain only refinements and functions:*

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

Proof. **Reflexivity**: We strengthen the IH, simultaneously proving that there exist e_p, e_{pf} and $\Gamma \vdash \tau_l \preceq \tau$ and $\Gamma \vdash \tau_r \preceq \tau$ such that $\Gamma, l : \tau_l, r : \tau_r \vdash e_{\mathsf{pf}} :: \mathsf{pf}(e, e, \tau)$ and $\Gamma \vdash e_p :: \mathsf{PEq}_\tau\ \{e\}\ \{e\}$ by induction on τ, leaving e general.

- $\tau \doteq \{x{:}b \mid e'\}$.

(1) Let $e_{\mathsf{pf}} = ()$.
(2) Let $e_p = \mathsf{bEq}_b\ e\ e\ e_{\mathsf{pf}}$.

(3) Let $\tau_l = \tau_r = \{x{:}b \mid x ==_b e\}$.
(4) We have $\Gamma \vdash x ==_b e \ \preceq \ \tau$ by SBase and semantic typing.
(5) We find $\Gamma \vdash e_p :: \mathsf{PEq}_b \ \{e\} \ \{e\}$ by TEqBase, with $e_l = e_r = e$. We must show:
  (a) $\Gamma \vdash e_l :: \tau_l$ and $\Gamma \vdash e_r :: \tau_r$, i.e., $\Gamma \vdash e :: \{x{:}b \mid x ==_b e\}$;
  (b) $\Gamma \vdash \tau_r \ \preceq \ \{x{:}b \mid \mathtt{true}\}$ and $\Gamma \vdash \tau_l \ \preceq \ \{x{:}b \mid \mathtt{true}\}$; and
  (c) $\Gamma, r : \tau_r, l : \tau_l \vdash e_{\mathrm{pf}} :: \{x{:}() \mid l ==_b r\}$.
(6) We find (5a) by TSelf.
(7) We find (5b) immediately by SBase.
(8) We find (5c) by TVar, using TSub to see that if $l, r : \{x{:}b \mid x ==_b e\}$ then $\mathtt{unit}$ will be typeable at the refinement where both l and r are equal to e.

- $\tau \doteq x{:}\tau_x \rightarrow \tau'$.

(1) $\Gamma, x : \tau_x \vdash e \ x :: \tau[x/x]$ by TApp and TVar, noting that $\tau[x/x] = \tau$.
(2) By the IH on $\Gamma, x : \tau_x \vdash e \ x :: \tau'[x/x] = \tau'$, there exist $e'_p, e'_{\mathrm{pf}}, \tau'_l$, and τ'_r such that:
  (a) $x : \tau_x \vdash \tau'_l \ \preceq \ \tau$ and $x : \tau_x \vdash \tau'_r \ \preceq \ \tau$;
  (b) $\Gamma, x : \tau_x, l : \tau'_l, r : \tau'_r \vdash e'_{\mathrm{pf}} :: \mathrm{pf}(e\,x, e\,x, \tau')$; and
  (c) $\Gamma, x : \tau_x \vdash e'_p :: \mathsf{PEq}_{\tau'} \ \{e \ x\} \ \{e \ x\}$.
(3) If $\tau' = \{x{:}() \mid \tau'\} e \ x e \ x$, then $\mathrm{pf}(e\,x, ex, b) = \{x{:}() \mid ex ==_b ex\}$; otherwise, $\mathrm{pf}(l, r, x{:}\tau_x \rightarrow \tau) = x{:}\tau_x \rightarrow \mathsf{PEq}_\tau \ \{e \ x\} \ \{e \ x\}$.
  In the former case, let $e''_{\mathrm{pf}} = \mathsf{bEq}_b \ (e \ x)(e \ x)e'_{\mathrm{pf}}$. In the latter case, let $e''_{\mathrm{pf}} = e'_{\mathrm{pf}}$.
  Either way, we have $\Gamma, x : \tau_x, l : \tau'_l, r : \tau'_r \vdash e''_{\mathrm{pf}} :: \mathsf{PEq}_{\tau'} \ \{e \ x\} \ \{e \ x\}$ by TEqBase or TEqFun, respectively.
(4) Let $e_{\mathrm{pf}} = x{:}\tau_x \rightarrow e''_{\mathrm{pf}}$.
(5) Let $e_p = \mathsf{xEq}_{x:\tau_x \rightarrow \tau} \ e \ e \ e_{\mathrm{pf}}$.
(6) Let $e_l = e_r = e$ and $\tau_l = x{:}\tau_x \rightarrow \tau'_l$ and $\tau_r = x{:}\tau_x \rightarrow \tau'_r$.
(7) We find subtyping by SFun and (2a).
(8) By TEqFun. We must show:
  (a) $\Gamma \vdash e_l :: \tau_l$ and $\Gamma \vdash e_r :: \tau_r$;
  (b) $\Gamma \vdash \tau_l \ \preceq \ x{:}\tau_x \rightarrow \tau$ and $\Gamma \vdash \tau_r \ \preceq \ x{:}\tau_x \rightarrow \tau$;
  (c) $\Gamma, r : \tau_r, l : \tau_l \vdash e_{\mathrm{pf}} :: (x{:}\tau_x \rightarrow \mathsf{PEq}_\tau \ \{l \ x\} \ \{r \ x\})$
  (d) $\Gamma \vdash x{:}\tau_x \rightarrow \tau$
(9) We find (8a) by assumption, TSub, and (7).
(10) We find (8b) by (7).
(11) We find (8c) by TLam and (2b).

- $\tau \doteq \mathsf{PEq}_{\tau'} \ \{e_1\} \ \{e_2\}$. These types are not equable, so we ignore them.

Symmetry: By induction on τ.

- $\tau \doteq \{x{:}b \mid e\}$.

(1) We have $\Gamma \vdash v_{12} :: \mathsf{PEq}_b \ \{e_1\} \ \{e_2\}$.
(2) By canonical forms, $v_{12} = \mathsf{bEq}_b \ e_l \ e_r \ v_p$ such that $\Gamma \vdash e_l :: \tau_l$ and $\Gamma \vdash e_r :: \tau_r$ (for some τ_l and τ_r that are refinements of b) and $\Gamma, r : \tau_r, l : \tau_l \vdash v_p :: \{x{:}() \mid l ==_b r\}$ (Lemma B.12).
(3) Let $v_{21} = \mathsf{bEq}_b \ e_r \ e_l \ v_p$.
(4) By TEqBase, swapping τ_l and τ_r from (2). We already have appropriate typing and subtyping derivations; we only need to see $\Gamma, l : \tau_l, r : \tau_r \vdash v_p :: \{x{:}() \mid r ==_b l\}$.
(5) We have $\Gamma, l : \tau_l, r : \tau_r \vdash \{x{:}() \mid r ==_b l\} \ \preceq \ \{x{:}() \mid l ==_b r\}$ by SBase and symmetry of $(==_b)$.

- $\tau \doteq x{:}\tau_x \rightarrow \tau'$.

(1) We have $\Gamma \vdash v_{12} :: \mathsf{PEq}_{x:\tau_x \rightarrow \tau'} \ \{e_1\} \ \{e_2\}$.

(2) By canonical forms, $v_{12} = \mathsf{xEq}_{x:\tau_x' \to \tau''}\ e_l\ e_r\ v_p$ such that $\tau_x \vdash \tau_x' \ \leq\ and\ \tau'' \vdash \tau' \ \leq\ and$ $\Gamma \vdash e_l :: \tau_l$ and $\Gamma \vdash e_r :: \tau_r$ (for some τ_l and τ_r that are subtypes of $x{:}\tau_x' \to \tau''$) and $\Gamma, r : \tau_r, l : \tau_l \vdash v_p :: x{:}\tau_x' \to \mathsf{PEq}_{\tau''}\ \{l\ x\}\ \{r\ x\}$.
(3) By canonical forms, this time on v_p from (2), $v_p = \text{TLam} x \tau_x' e_p$ such that $\Gamma \vdash \tau_x \ \leq\ \tau_x'$ and $\Gamma, r : \tau_r, l : \tau_l, x : \tau_x' \vdash e :: \tau'''$ such that $\Gamma, r : \tau_r, l : \tau_l, x : \tau_x' \vdash \tau''' \ \leq\ \mathsf{PEq}_{\tau''}\ \{l\ x\}\ \{r\ x\}$.
(4) By TSub, (3), and the IH on $\mathsf{PEq}_{\tau''}\ \{l\ x\}\ \{r\ x\}$, we know there exists some e_p' such that $\Gamma, l : \tau_l, r : \tau_r, x : \tau_x' \vdash e_p' :: \mathsf{PEq}_{\tau''}\ \{r\ x\}\ \{l\ x\}$.
(5) Let $v_p' = x{:}\tau_x' \to e_p'$.
(6) By (4) and TLam, and TSub (using subtyping from (3) and (2)), $\Gamma, l : \tau_l, r : \tau_r \vdash v_p' ::$ $\mathsf{PEq}_{x:\tau_x \to \tau'}\ \{e_r\ x\}\ \{e_l\ x\}$.
(7) Let $v_{21} = \mathsf{xEq}_{x:\tau_x \to \tau'}\ e_r\ e_l\ v_p'$.
(8) By TEqBase, with (6) for the proof and (3) and (2) for the rest.

- $\tau \doteq \mathsf{PEq}_{\tau'}\ \{e_1\}\ \{e_2\}$. These types are not equable, so we ignore them.

Transitivity: By induction on τ.

- $\tau \doteq \{x{:}b \mid e\}$.

(1) We have $\Gamma \vdash v_{12} :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ and $\Gamma \vdash v_{23} :: \mathsf{PEq}_\tau\ \{e_2\}\ \{e_3\}$.
(2) By canonical forms, $v_{12} = \mathsf{bEq}_b\ e_1\ e_2\ v_{12}'$ such that $\Gamma \vdash e_1 :: \tau_1$ and $\Gamma \vdash e_2 :: \tau_2$ (for some τ_1 and τ_2 that are refinements of b) and $\Gamma, r : \tau_2, l : \tau_1 \vdash v_{12}' :: \{x{:}() \mid l ==_b r\}$. and, similarly, $v_{23} = \mathsf{bEq}_b\ e_1\ e_2\ v_{23}'$ such that $\Gamma \vdash e_2 :: \tau_2'$ and $\Gamma \vdash e_3 :: \tau_3$ (for some τ_2' and τ_3 that are refinements of b) and $\Gamma, r : \tau_3, l : \tau_2' \vdash v_{23}' :: \{x{:}() \mid l ==_b r\}$.
(3) By canonical forms again, we know that $v_{12}' = v_{23}' = \mathtt{unit}$ and we have:

$$\begin{aligned}&\Gamma, r : \tau_2, l : \tau_1 \vdash \{x{:}() \mid x ==_{()} \mathtt{unit}\} \ \leq\ \{x{:}b \mid \{x{:}() \mid l ==_b r\}\}, \text{ and}\\ &\Gamma, r : \tau_3, l : \tau_2' \vdash \{x{:}() \mid x ==_{()} \mathtt{unit}\} \ \leq\ \{x{:}b \mid \{x{:}() \mid l ==_b r\}\}.\end{aligned}$$

(4) Elaborating on (3), we know that $\forall \theta \in [\![\Gamma, r : \tau_2, l : \tau_1]\!]$, we have:

$$\left[\!\left[\theta \cdot \{x{:}() \mid x ==_{()} \mathtt{unit}\}\right]\!\right] \subseteq [\![\theta \cdot \{x{:}() \mid l ==_b r\}]\!]$$

and $\forall \theta \in \left[\!\left[\Gamma, r : \tau_3, l : \tau_2'\right]\!\right]$, we have:

$$\left[\!\left[\theta \cdot \{x{:}() \mid x ==_{()} \mathtt{unit}\}\right]\!\right] \subseteq [\![\theta \cdot \{x{:}() \mid l ==_b r\}]\!] .$$

(5) Since $\{x{:}() \mid x ==_{()} \mathtt{unit}\}$ contains all computations that terminate with $\mathtt{unit}$ in all models (Theorem B.1), the right-hand sides of the equations must also hold all unit computations. That is, all choices for l and r_2 (resp. l and r) that are semantically well typed are necessarily equal.
(6) By (5), we can infer that in any given model, τ_1, τ_2, τ_2', and τ_3 identify just one b-constant. Why must τ_2 and τ_2' agree? In particular, e_2 has *both* of those types, but by semantic soundness (Theorem B.2), we know that it will go to a value in the appropriate type interpretation. By determinism of evaluation, we know it must be the *same* value. We can therefore conclude that $\forall \theta \in [\![\Gamma, r : \tau_3, l : \tau_1]\!]$, $\left[\!\left[\theta \cdot \{x{:}() \mid x ==_{()} \mathtt{unit}\}\right]\!\right] \subseteq [\![\theta \cdot \{x{:}() \mid l ==_b r\}]\!]$.
(7) By TEqBase, using τ_1 and τ_3 and $\mathtt{unit}$ as the proof. We need to show $\Gamma, r : \tau_3, l : \tau_1 \vdash$ $\mathtt{unit} :: \{x{:}() \mid l ==_b r\}$; all other premises follow from (2).
(8) By TSub and SBase, using (6) for the subtyping.

- $\tau \doteq x{:}\tau_x \to \tau'$.

(1) We have $\Gamma \vdash v_{12} :: \mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ and $\Gamma \vdash v_{23} :: \mathsf{PEq}_\tau\ \{e_2\}\ \{e_3\}$.
(2) By canonical forms, we have

$$\begin{aligned} v_{12} &= \mathsf{xEq}_{x:\tau_x \to \tau'}\ e_1\ e_2\ v_{12}' \\ v_{23} &= \mathsf{xEq}_{x:\tau_x \to \tau'}\ e_2\ e_3\ v_{23}' \end{aligned}$$

where there exist types τ_1, τ_2, τ_2', and τ_3 subtypes of $x{:}\tau_x \to \tau'$ such that

$$\begin{array}{ll}\Gamma \vdash e_1 :: \tau_1 & \Gamma \vdash e_2 :: \tau_2 \\ \Gamma \vdash e_2 :: \tau_2' & \Gamma \vdash e_3 :: \tau_3\end{array}$$

and there exist types $\tau_{x_{12}}$, $\tau_{x_{23}}$, τ_{12}', and τ_{23}' such that

$$\begin{array}{l}\Gamma, r : \tau_2, l : \tau_1 \vdash v_{p_{12}} :: x{:}\tau_{x_{12}} \to \mathsf{PEq}_{\tau_{12}'}\ \{l\ x\}\ \{r\ x\}, \\ \Gamma, r : \tau_2, l : \tau_1 \vdash \tau_x \ \preceq\ \tau_{x_{12}}, \\ \Gamma, r : \tau_2, l : \tau_1, x : \tau_x \vdash \tau_{12}' \ \preceq\ \tau', \\ \Gamma, r : \tau_3, l : \tau_2' \vdash v_{p_{23}} :: x{:}\tau_x' \to \mathsf{PEq}_{\tau_{23}'}\ \{l\ x\}\ \{r\ x\}, \\ \Gamma, r : \tau_3, l : \tau_2' \vdash \tau_x \ \preceq\ \tau_{x_{23}}, \text{ and} \\ \Gamma, r : \tau_3, l : \tau_2', x : \tau_x \vdash \tau_{23}' \ \preceq\ \tau'.\end{array}$$

(3) By canonical forms on $v_{p_{12}}$ and $v_{p_{23}}$ from (2), we know that:

$$v_{p_{12}} = \lambda x{:}\tau_{x_{12}}.\ e_{12}' \qquad v_{p_{23}} = \lambda x{:}\tau_{x_{23}}.\ e_{23}'$$

such that:

$$\begin{array}{l}\Gamma, r : \tau_2, l : \tau_1, x : \tau_{x_{12}} \vdash e_{12}' :: \tau_{12}'', \\ \Gamma, r : \tau_2, l : \tau_1, x : \tau_{x_{12}} \vdash \tau_{12}'' \ \preceq\ \tau_{12}', \\ \\ \Gamma, r : \tau_3, l : \tau_2', x : \tau_{x_{23}} \vdash e_{23}' :: \tau_{23}'', \\ \Gamma, r : \tau_3, l : \tau_2', x : \tau_{x_{23}} \vdash \tau_{23}'' \ \preceq\ \tau_{23}', \text{ and}\end{array}$$

(4) By strengthening (Lemma B.7) using (2), we can replace x's type with τ_x in both proofs, to find:

$$\begin{array}{l}\Gamma, r : \tau_2, l : \tau_1, x : \tau_x \vdash e_{12}' :: \tau_{12}', \text{and} \\ \Gamma, r : \tau_3, l : \tau_2', x : \tau_x \vdash e_{23}' :: \tau_{23}'.\end{array}$$

Then, by TSub, we can relax the type of the proof bodies:

$$\begin{array}{l}\Gamma, r : \tau_2, l : \tau_1, x : \tau_x \vdash e_{12}' :: \tau', \text{and} \\ \Gamma, r : \tau_3, l : \tau_2', x : \tau_x \vdash e_{23}' :: \tau'.\end{array}$$

(5) By (4, (3), and the IH on $\mathsf{PEq}_{\tau'}\ \{l\ x\}\ \{r\ x\}$, we know there exists some proof body e_{13}' such that $\Gamma, r : \tau_3, l : \tau_1 \vdash e_{13}' :: \mathsf{PEq}_{\tau'}\ \{l\ x\}\ \{r\ x\}$.
(6) Let $v_p = x{:}\tau_x \to e_{13}'$.
(7) By (5), and TLam.
(8) Let $v_{13} = \mathsf{xEq}_{x:\tau_x \to \tau'}\ e_1\ e_3\ v_p$.
(9) By TEqBase, with (7) for the proof and (2) for the rest.

- $\tau \doteq \mathsf{PEq}_{\tau'}\ \{e_1\}\ \{e_2\}$. These types are not equable, so we ignore them. □

C PARALLEL REDUCTION AND COTERMINATION

The conventional application rule for dependent types substitutes a term into a type, finding $e_1\ e_2 : \tau[e_2/x]$ when $e_1 : x{:}\tau_x \to \tau$. We define two logical relations: a unary interpretation of types (Figure 8) and a binary logical relation characterizing equivalence (Figure 10). Both of these logical relations are defined as fixpoints on types. The type index poses a problem: the function case of these logical relations quantify over values in the relation, but we sometimes need to reason about expressions, not values. If $e \hookrightarrow^* v$, are $\tau[e/x]$ and $\tau[v/x]$ treated the same by our logical relations? We encounter this problem in particular in proof of logical relation compositionality, which is precisely about exchanging expressions in types with the values the expressions reduce to in closing substitutions: for the unary logical relation and binary logical relation (Lemma B.21).

The key technical device to prove these compositionality lemmas is *parallel reduction* (Figure 13). Parallel reduction generalizes our call-by-value relation to allow multiple steps at once, throughout a term—even under a lambda. Parallel reduction is a bisimulation (Lemma C.5 for forward simulation; Corollary C.15 for backward simulation). That is, expressions that parallel reduce to each other go to identical constants or expressions that themselves parallel reduce, and the logical relations put terms that parallel reduce in the same equivalence class.

To prove the compositionality lemmas, we first show that (a) the logical relations are closed under parallel reduction (for the unary relation and Lemma B.20 for the binary relation) and (b) use the backward simulation to change values in the closing substitution to a substituted expression in the type.

Our proof comes in three steps. First, we establish some basic properties of parallel reduction (§C.1). Next, proving the forward simulation is straightforward (§C.2): if $e_1 \rightrightarrows e_2$ and $e_1 \hookrightarrow e_1'$, then either parallel reduction contracted the redex for us and $e_1' \rightrightarrows e_2$ immediately, or the redex is preserved and $e_2 \hookrightarrow e_2'$ such that $e_1' \rightrightarrows e_2'$. Proving the backward simulation is more challenging (§C.3). If $e_1 \rightrightarrows e_2$ and $e_2 \hookrightarrow e_2'$, the redex contracted in e_2 may not yet be exposed. The trick is to show a tighter bisimulation, where the outermost constructors are always the same, with the subparts parallel reducing. We call this relation *congruence* (Figure 14); it's a straightforward restriction of parallel reduction, eliminating β, eq1, and eq2 as outermost constructors (but allowing them deeper inside). The key lemma shows that if $e_1 \rightrightarrows e_2$, then there exists e_1' $e_1 \hookrightarrow^* e_1'$ such that $e_1' \risingdotseq\!\!\rightrightarrows e_2$ (Lemma C.11). Once we know that parallel reduction implies reduction to congruent terms, proving that congruence is a backward simulation allows us to reason "up to congruence". In particular, congruence is a sub-relation of parallel reduction, so we find that parallel reduction is a backward simulation. Finally, we can show that $e_1 \rightrightarrows e_2$ implies observational equivalence (§C.4); for our purposes, it suffices to find cotermination at constants (Corollary C.17).

One might think, in light of Takahashi's explanation of parallel reduction [Takahashi 1989], that the simulation techniques we use are too powerful for our needs: why not simply rely on the Church-Rosser property and confluence, which she proves quite simply? Her approach works well when relating parallel reduction to full β-reduction (and/or η-reduction): the transitive closure of her parallel reduction relation is equal to the transitive closure of plain β-reduction (resp. η- and $\beta\eta$-reduction). But we're interested in programming languages, so our underlying reduction relation isn't full β: we use call-by-value, and we will never reduce under lambdas. But even if we were call-by-name, we would have the same issue. Parallel reduction implies reduction, but not to the *same* value, as in her setting. Parallel reduction yields values that are equivalent, *up to parallel reduction and congruence* (see, e.g., Corollary C.13).

C.1 Basic Properties

Lemma C.1 (Parallel reduction is reflexive). *For all e and τ, $e \rightrightarrows e$ and $\tau \rightrightarrows \tau$.*

Proof. By mutual induction on e and τ.

Expressions.

- $e \doteq x$. By var.
- $e \doteq c$. By const.
- $e \doteq \lambda x{:}\tau.\ e'$. By the IHs on τ and e' and lam.
- $e \doteq e_1\ e_2$. By the IH on e_1 and e_2 and app.
- $e \doteq \mathsf{bEq}_b\ e_l\ e_r\ e'$. By the IHs on e_l, e_r, and e' and beq.
- $e \doteq \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e'$. By the IHs on τ_x, τ, e_l, e_r, and e' and xeq.

Types.

- $\tau \doteq \{x{:}b \mid r\}$. By the IH on r (an expression) and ref.
- $\tau \doteq x{:}\tau_x \to \tau'$. By the IHs on τ_x and τ' and fun.

$$\boxed{e \rightrightarrows e}$$

$$\frac{}{x \rightrightarrows x}\text{var} \qquad \frac{}{c \rightrightarrows c}\text{const} \qquad \frac{\tau \rightrightarrows \tau' \quad e \rightrightarrows e'}{\lambda x{:}\tau.\ e \rightrightarrows \lambda x{:}\tau'.\ e'}\text{lam} \qquad \frac{e_1 \rightrightarrows e_1' \quad e_2 \rightrightarrows e_2'}{e_1\ e_2 \rightrightarrows e_1'\ e_2'}\text{app}$$

$$\frac{e \rightrightarrows e' \quad v \rightrightarrows v'}{(\lambda x{:}\tau.\ e)\ v \rightrightarrows e'[v'/x]}\beta \qquad \frac{}{(==_b)\ c_1 \rightrightarrows (==_{(c_1,b)})}\text{eq1} \qquad \frac{}{(==_{(c_1,b)})\ c_2 \rightrightarrows c_1 = c_2}\text{eq2}$$

$$\frac{e_l \rightrightarrows e_l' \quad e_r \rightrightarrows e_r' \quad e \rightrightarrows e'}{\mathsf{bEq}_b\ e_l\ e_r\ e \rightrightarrows \mathsf{bEq}_b\ e_l'\ e_r'\ e'}\text{beq} \qquad \frac{\tau_x \rightrightarrows \tau_x' \quad \tau \rightrightarrows \tau' \quad e_l \rightrightarrows e_l' \quad e_r \rightrightarrows e_r' \quad e \rightrightarrows e'}{\mathsf{xEq}_{x:\tau_x\to\tau}\ e_l\ e_r\ e \rightrightarrows \mathsf{xEq}_{x:\tau_x'\to\tau'}\ e_l'\ e_r'\ e'}\text{xeq}$$

$$\boxed{\tau \rightrightarrows \tau}$$

$$\frac{r \rightrightarrows r'}{\{x{:}b \mid r\} \rightrightarrows \{x{:}b \mid r'\}}\text{ref} \qquad \frac{\tau_x \rightrightarrows \tau_x' \quad \tau \rightrightarrows \tau'}{x{:}\tau_x \to \tau \rightrightarrows x{:}\tau_x' \to \tau'}\text{fun}$$

$$\frac{\tau \rightrightarrows \tau' \quad e_l \rightrightarrows e_l' \quad e_r \rightrightarrows e_r'}{\mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\} \rightrightarrows \mathsf{PEq}_{\tau'}\ \{e_l'\}\ \{e_r'\}}\text{eq}$$

Fig. 13. Parallel reduction in terms and types.

- $\tau \doteq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}$. By the IHs on τ', e_l, and e_r and eq. □

Lemma C.2 (Parallel reduction is substitutive). *If $e \rightrightarrows e'$, then:*

(1) If $e_1 \rightrightarrows e_2$, then $e_1[e/x] \rightrightarrows e_2[e'/x]$.
(2) If $\tau_1 \rightrightarrows \tau_2$, then $\tau_1[e/x] \rightrightarrows \tau_2[e'/x]$.

Proof. By mutual induction on e_1 and τ_1.

Expressions.

var $y \rightrightarrows y$. If $y \neq x$, then the substitution has no effect and the case is trivial. If $y = x$, then $x[e/x] = e$ and we have $e \rightrightarrows e'$ by assumption. We have $e \rightrightarrows e$ by reflexivity (Lemma C.1).

const $c \rightrightarrows c$. This case is trivial: the substitution has no effect.

lam $\lambda y{:}\tau.\ e' \rightrightarrows \lambda y{:}\tau.\ e''$. If $y \neq x$, then by the IH on e' and lam. If $y = x$, then the substitution has no effect and the case is trivial.

app $e_{11}\ e_{12} \rightrightarrows e_{21}\ e_{22}$, where $e1i \rightrightarrows e2i$ for $i = 1, 2$. By the IHs on e_{1i} and app.

beta $(\lambda y{:}\tau.\ e')\ v \rightrightarrows e'[v'/y]$, where $e' \rightrightarrows e''$ and $v \rightrightarrows v'$. If $y \neq x$, then $(\lambda y{:}\tau.\ e'[e/x])\ v[e/x] \rightrightarrows e''[e/x][v'[e/x]/y]$ by β. Since $y \neq x$, $e''[e/x][v'[e/x]/y] = e''[v'/y][e/x]$ as desired.
If $y = x$, then the substitution in the lambda has no effect, and we find $(\lambda x{:}\tau.\ e')\ v[e/x] \rightrightarrows e''[v'[e/x]/x]$ by β. We have $e''[v'[e/x]/x] = e''[v'/x][e/x]$ as desired.

eq1 $(==_b)\ c_1 \rightrightarrows (==_{(c_1,b)})$. This case is trivial by eq1, as the substitution has no effect.

eq2 $(==_{(c_1,b)})\ c_2 \rightrightarrows c_1 = c_2$. This case is trivial by eq2, as the substitution has no effect.

beq $\mathsf{bEq}_b\ e_l\ e_r\ e_p \rightrightarrows \mathsf{bEq}_b\ e_l'\ e_r'\ e_p'$, where $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e_p \rightrightarrows e_p'$. By the IHs on e_l, e_r, and e_p and beq.

xeq $\mathsf{xEq}_{x:\tau_x\to\tau}\ e_l\ e_r\ e_p \rightrightarrows \mathsf{xEq}_{x:\tau_x\to\tau}\ e_l'\ e_r'\ e_p'$, where $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e_p \rightrightarrows e_p'$. By the IHs on e_l, e_r, and e_p and xeq.

Types.

ref $\{y{:}b \mid r\} \rightrightarrows \{y{:}b \mid r'\}$ where $r \rightrightarrows r'$. If $y \neq x$, then $r[e/x] \rightrightarrows r'[e'/x]$ by the IH on r; we are done by ref.
If $y = x$, then the substitution has no effect, and the case is immediate by reflexivity (Lemma C.1).

fun $y{:}\tau_y \to \tau \rightrightarrows y{:}\tau'_y \to \tau'$ where $\tau_y \rightrightarrows \tau'_y$ and $\tau \rightrightarrows \tau'$. If $y \neq x$, then by the IH on τ_y and τ and fun.
If $y = x$, then the substitution only has effect in the domain. The IH on τ_y finds $\tau_y[e/x] \rightrightarrows \tau'_y[e'/x]$ in the domain; reflexivity covers the codomain (Lemma C.1), and we are done by fun.

eq $\mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\} \rightrightarrows \mathsf{PEq}_{\tau'}\ \{e'_l\}\ \{e'_r\}$. By the IHs and eq. □

Corollary C.3 (Substituting multiple parallel reduction is parallel reduction). *If $e_1 \rightrightarrows^* e_2$, then $e[e_1/x] \rightrightarrows^* e[e2/x]$.*

Proof. First, notice that $e \rightrightarrows e$ by reflexivity (Lemma C.1). By induction on $e_1 \rightrightarrows^* e_2$, using reflexivity in the base case (Lemma C.1); the inductive step uses substituting parallel reduction (Lemma C.2) and the IH. □

Lemma C.4 (Parallel reduction subsumes reduction). *If $e_1 \hookrightarrow e_2$ then $e_1 \rightrightarrows e_2$.*

Proof. By induction on the evaluation derivation, using reflexivity of parallel reduction to cover expressions and types that didn't step (Lemma C.1).

ctx $\mathcal{E}[e] \hookrightarrow \mathcal{E}[e']$, where $e \hookrightarrow e'$. By the IH, $e \rightrightarrows e'$. By structural induction on $\mathcal{E}$.
- $\mathcal{E} \doteq \bullet$. By the outer IH.
- $\mathcal{E} \doteq \mathcal{E}_1\ e_2$. By the inner IH on $\mathcal{E}_1$, reflexivity on e_2, and app.
- $\mathcal{E} \doteq v_1\ \mathcal{E}_2$. By reflexivity on v_1, the inner IH on $\mathcal{E}_2$, and app.
- $\mathcal{E} \doteq \mathsf{bEq}_b\ e_l\ e_r\ \mathcal{E}'$. By reflexivity on e_l and e_r, the inner IH on and $\mathcal{E}'$, and beq.
- $\mathcal{E} \doteq \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ \mathcal{E}'$. By reflexivity on τ_x, τ, e_l and e_r, the inner IH on and $\mathcal{E}'$, and xeq.

β $(\lambda x{:}\tau.\ e)\ v \hookrightarrow e[v/x]$. By reflexivity (Lemma C.1, $e \rightrightarrows e$ and $v \rightrightarrows v$. By beta, $(\lambda x{:}\tau.\ e)\ v \rightrightarrows e[v/x]$.

eq1 By eq1.

eq2 By eq2. □

C.2 Forward Simulation

Lemma C.5 (Parallel reduction is a forward simulation). *If $e_1 \rightrightarrows e_2$ and $e_1 \hookrightarrow e'_1$, then there exists e'_2 such that $e_2 \hookrightarrow^* e'_2$ and $e'_1 \rightrightarrows e'_2$.*

Proof. By induction on the derivation of $e_1 \hookrightarrow e'_1$, leaving e_2 general.

ctx By structural induction on $\mathcal{E}$, using reflexivity (Lemma C.1) on parts where the IH doesn't apply.
- $\mathcal{E} \doteq \bullet$. By the outer IH on the actual step.
- $\mathcal{E} \doteq \mathcal{E}_1\ e_2$. By the IH on $\mathcal{E}_1$, reflexivity on e_2, and app.
- $\mathcal{E} \doteq v_1\ \mathcal{E}_2$. By reflexivity on v_1, the IH on $\mathcal{E}_2$, and app.
- $\mathcal{E} \doteq \mathsf{bEq}_b\ e_l\ e_r\ \mathcal{E}'$. By reflexivity on e_l and e_r, the IH on $\mathcal{E}'$, and beq.
- $\mathcal{E} \doteq \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ \mathcal{E}'$. By reflexivity on τ_x, τ, e_l and e_r, the IH on $\mathcal{E}'$, and xeq.

β $(\lambda x{:}\tau.\ e)\ v \hookrightarrow e[v/x]$. One of two rules could have applied to find $e_1 \rightrightarrows e_2$: app or β.

In the app case, we have $e_2 = (\lambda x{:}\tau'.\ e')\ v'$ where $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e'$ and $v \rightrightarrows v'$. Let $e_2' = e'[v'/x]$. We find $e_2 \hookrightarrow^* e_2'$ in one step by β. We find $e[v/x] \rightrightarrows e'[v'/x]$ by substitutivity of parallel reduction (Lemma C.2).

In the β case, we have $e_2 = e'[v'/x]$ such that $e \rightrightarrows e'$ and $v \rightrightarrows v'$. Let $e_2' = e_2$. We find $e_2 \hookrightarrow^* e_2'$ in no steps at all; we find $e_1' \rightrightarrows e_2'$ by substitutivity of parallel reduction (Lemma C.2).

eq1 $(==_b)\ c_1 \hookrightarrow (==_{(c_1,b)})$. One of two rules could have applied to find $(==_b)\ c_1 \rightrightarrows e_2$: app or eq1.

In the app case, we must have $e_2 = e_1 = (==_b)\ c_1$, because there are no reductions available in these constants. Let $e_2' = (==_{(c_1,b)})$. We find $e_2 \hookrightarrow^* e_2'$ in a single step by our assumption (or eq1). We find parallel reduction by reflexivity (Lemma C.1).

In the eq2 case, we have $e_2 = e_1' = (==_{(c_1,b)})$. Let $e_2' = e_2$. We find $e_2 \hookrightarrow^* e_2'$ in no steps at all. We find parallel reduction by reflexivity (Lemma C.1).

eq2 $(==_{(c_1,b)})\ c_2 \hookrightarrow c_1 = c_2$. One of two rules could have applied to find $(==_{(c_1,b)})\ c_2 \rightrightarrows e_2$: app or eq2.

In the app case, we have $e_2 = e_1 = (==_{(c_1,b)})\ c_2$, because there are no reductions available in these constants. Let $e_2' \doteq c_1 = c_2$, i.e. `true` when $c_1 = c_2$ and `false` otherwise. We find $e_2 \hookrightarrow^* e_2'$ in a single step by our assumption (or eq2). We find parallel reduction by reflexivity (Lemma C.1).

In the eq2 case, we have $e_2 = e_1' \doteq c_1 = c_2$, i.e. `true` when $c_1 = c_2$ and `false` otherwise. Let $e_2' = e_2$. We find $e_2 \hookrightarrow^* e_2'$ in no steps at all. We find parallel reduction by reflexivity (Lemma C.1). □

C.3 Backward Simulation

Lemma C.6 (Reduction is substitutive). *If $e_1 \hookrightarrow e_2$, then $e_1[e/x] \hookrightarrow e_2[e/x]$.*

Proof. By induction on the derivation of $e_1 \hookrightarrow e_2$.

ctx By structural induction on $\mathcal{E}$.
- $\mathcal{E} \doteq \bullet$. By the outer IH.
- $\mathcal{E} \doteq \mathcal{E}_1\ e_2$. By the IH on $\mathcal{E}_1$ and ctx.
- $\mathcal{E} \doteq v_1\ \mathcal{E}_2$. By the IH on $\mathcal{E}_2$ and ctx.
- $\mathcal{E} \doteq \mathsf{bEq}_b\ e_l\ e_r\ \mathcal{E}'$. By the IH on $\mathcal{E}'$ and ctx.
- $\mathcal{E} \doteq \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ \mathcal{E}'$. By the IH on $\mathcal{E}'$ and ctx.

β $(\lambda y{:}\tau.\ e')\ v \hookrightarrow e'[v/y]$. We must show $(\lambda y{:}\tau.\ e')[e/x]\ v[e/x] \hookrightarrow e'[v/y][e/x]$.

The exact result depends on whether $y = x$. If $y \neq x$, the substitution goes through, and we have $(\lambda y{:}\tau.\ e')[e/x] = \lambda y{:}\tau[e/x].\ e'[e/x]$. By β, $(\lambda y{:}\tau[e/x].\ e'[e/x])\ v[e/x] \hookrightarrow e'[e/x][v[e/x]/y]$. But $e'[e/x][v[e/x]/y] = e'[v/y][e/x]$, and we are done.

If, on the other hand, $y = x$, then the substitution has no effect in the body of the lambda, and $(\lambda y{:}\tau.\ e')[e/x] = \lambda y{:}\tau[e/x].\ e'$. By β again, we find $(\lambda y{:}\tau[e/x].\ e')\ v[e/x] \hookrightarrow e'[v[e/x]/y]$. Since $y = x$, we really have $e'[v[e/x]/x]$ which is the same as $e'[v/x][e/x] = e'[v/y][e/x]$, as desired.

eq1 The substitution has no effect; immediate, by eq1.

eq2 The substitution has no effect; immediate, by eq2. □

Corollary C.7 (Multi-step reduction is substitutive). *If $e_1 \hookrightarrow^* e_2$, then $e_1[e/x] \hookrightarrow^* e_2[e/x]$.*

Proof. By induction on the derivation of $e_1 \hookrightarrow^* e_2$. The base case is immediate ($e_1 = e_2$, and we take no steps). The inductive case follows by the IH and single-step substitutivity (Lemma C.6). □

$$\frac{}{x \rightsquigarrow x}\,\text{var} \qquad \frac{}{c \rightsquigarrow c}\,\text{const} \qquad \frac{\tau \rightrightarrows \tau' \quad e \rightrightarrows e'}{\lambda x{:}\tau.\ e \rightsquigarrow \lambda x{:}\tau'.\ e'}\,\text{lam} \qquad \frac{e_1 \rightrightarrows e_1' \quad e_2 \rightrightarrows e_2'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2'}\,\text{app}$$

$$\frac{e_l \rightrightarrows e_l' \quad e_r \rightrightarrows e_r' \quad e \rightrightarrows e'}{\mathsf{bEq}_b\ e_l\ e_r\ e \rightsquigarrow \mathsf{bEq}_b\ e_l'\ e_r'\ e'}\,\text{beq} \qquad \frac{\tau_x \rightrightarrows \tau_x' \quad \tau \rightrightarrows \tau' \quad e_l \rightrightarrows e_l' \quad e_r \rightrightarrows e_r' \quad e \rightrightarrows e'}{\mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \rightsquigarrow \mathsf{xEq}_{x:\tau_x' \to \tau'}\ e_l'\ e_r'\ e'}\,\text{xeq}$$

Fig. 14. Term congruence.

We say terms are *congruent* when they (a) have the same outermost constructor and (b) their subparts parallel reduce to each other.[4] That is, $\rightsquigarrow \subseteq \rightrightarrows$, where the outermost rule must be one of var, const, lam, app, beq, or xeq and cannot be a *real* reduction like β, eq1, or eq2.

Congruence is a key tool in proving that parallel reduction is a backward simulation. Parallel reductions under a lambda prevent us from having an "on-the-nose" relation, but reduction can keep up enough with parallel reduction to maintain congruence.

Lemma C.8 (Congruence implies parallel reduction). *If $e_1 \rightsquigarrow e_2$ then $e_1 \rightrightarrows e_2$.*

Proof. By induction on the derivation of $e_1 \rightsquigarrow e_2$.

var $x \rightsquigarrow x$. By var.
const $c \rightsquigarrow c$. By const.
lam $\lambda x{:}\tau.\ e \rightsquigarrow \lambda x{:}\tau'.\ e'$, with $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e'$. By lam.
app $e_1\ e_2 \rightsquigarrow e_1'\ e_2'$, with $e_1 \rightrightarrows e_1'$ and $e_2 \rightrightarrows e_2'$. By app.
beq $\mathsf{bEq}_b\ e_l\ e_r\ e \rightsquigarrow \mathsf{bEq}_b\ e_l'\ e_r'\ e$, with $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e \rightrightarrows e'$. By beq.
xeq By xeq. $\mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \rightsquigarrow \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l'\ e_r'\ e$, with $\tau_x \rightrightarrows \tau_x'$ and $\tau \rightrightarrows \tau'$ and $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e \rightrightarrows e'$. By xeq. □

We need to strengthen substitutivity (Lemma C.2) to show that it preserves congruence.

Corollary C.9 (Congruence is substitutive). *If $e_1 \rightsquigarrow e_1'$ and $e_2 \rightsquigarrow e_2'$, then $e_1[e_2/x] \rightsquigarrow e_2[e_2'/x]$.*

Proof. By cases on e_1.

- $e_1 = y$. It must be that $e_2 = y$ as well, since only var could have applied. If $y \neq x$, then the substitution has no effect and we have $y \rightsquigarrow y$ by assumption (or var). If $x = y$, then $e_1[e_2/x] = e_2$ and we have $e_2 \rightsquigarrow e_2'$ by assumption.
- $e_1 = c$. It must be that $e_2 = c$ as well. The substitution has no effect; immediate by var.
- $e_1 = \lambda y{:}\tau.\ e$. It must be that $e_2 = \lambda y{:}\tau'.\ e'$ such that $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e'$. If $y \neq x$, then we must show $\lambda y{:}\tau[e_2/x].\ e[e_2/x] \rightsquigarrow \lambda y{:}\tau'[e_2'/x].\ e'[e_2'/x]$, which we have immediately by lam and Lemma C.2 on our two subparts. If $y = x$, then we must show $\lambda y{:}\tau[e_2/x].\ e \rightsquigarrow \lambda y{:}\tau'[e_2'/x].\ e'$, which we have immediately by lam, Lemma C.2 on our $\tau \rightrightarrows \tau'$, and the fact that $e \rightrightarrows e'$.
- $e_1 = e_{11}\ e_{12}$. It must be that $e_2 = e_{21}\ e_{22}$, such that $e_{11} \rightrightarrows e_{21}$ and $e_{12} \rightrightarrows e_{22}$. By app and Lemma C.2 on the subparts.
- $e_1 = \mathsf{bEq}_b\ e_l\ e_r\ e$. It must be the case that $e_2 = \mathsf{bEq}_b\ e_l'\ e_r'\ e'$ where $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$. By beq and Lemma C.2 on the subparts.
- $e_1 = \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e$. It must be the case that $e_2 = \mathsf{xEq}_{x:\tau_x' \to \tau'}\ e_l'\ e_r'\ e'$ where $e_l \rightrightarrows e_l'$ (and similarly for τ_x, τ, e_r, and e). By xeq and Lemma C.2 on the subparts. □

[4] Congruent terms are related to Takahashi's $\tilde{M}$ operator: in that they characterize parallel reductions that preserve structure. They are not the same, though: Takahashi's $\tilde{M}$ will do $\beta\eta$-reductions on outermost redexes.

Lemma C.10 (Parallel reduction of values implies congruence). *If $v_1 \rightrightarrows v_2$ then $v_1 \leftrightarrows v_2$.*

Proof. By induction on the derivation of $v_1 \rightrightarrows v_2$.

var Contradictory: variables aren't values.
const Immediate, by const.
lam Immediate, by lam.
app Contradictory: applications aren't values.
beq Immediate, by beq.
xeq Immediate, by xeq.
β Contradictory: applications aren't values.
eq1 Contradictory: applications aren't values.
eq2 Contradictory: applications aren't values. □

Lemma C.11 (Parallel reduction implies reduction to congruent forms). *If $e_1 \rightrightarrows e_2$, then there exists e_1' $e_1 \hookrightarrow^* e_1'$ such that $e_1' \leftrightarrows e_2$.*

Proof. By induction on $e_1 \rightrightarrows e_2$.

Structural rules.

var $x \rightrightarrows x$. We have $e_1 = e_2 = x$ by var.
const $c \rightrightarrows c$. We have $e_1 = e_2 = c$ by const.
lam $\lambda x{:}\tau.\ e \rightrightarrows \lambda x{:}\tau'.\ e'$, where $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e'$. Immediate, by lam.
app $e_{11}\ e_{12} \rightrightarrows e_{21}\ e_{22}$, where $e_{11} \rightrightarrows e_{21}$ and $e_{12} \rightrightarrows e_{22}$. Immediate, by app.
beq $\mathsf{bEq}_b\ e_l\ e_r\ e \rightrightarrows \mathsf{bEq}_b\ e_l'\ e_r'\ e'$ where $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e \rightrightarrows e'$. Immediate, by beq.
xeq $\mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \rightrightarrows \mathsf{xEq}_{x:\tau_x' \to \tau'}\ e_l'\ e_r'\ e'$ where $\tau_x \rightrightarrows \tau_x'$ and $\tau \rightrightarrows \tau'$ and $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e \rightrightarrows e'$. Immediate, by xeq.

Reduction rules. These are the more interesting cases, where the parallel reduction does a reduction step—ordinary reduction has to do more work to catch up.

β $(\lambda x{:}\tau.\ e)\ v \rightrightarrows e'[v'/x]$, where $e \rightrightarrows e''$ and $v \rightrightarrows v''$.
We have $(\lambda x{:}\tau.\ e)\ v \hookrightarrow e[v/x]$ by β. By the IH on $e \rightrightarrows e''$, there exists e' such that $e \hookrightarrow^* e'$ such that $e' \leftrightarrows e''$. We *ignore* the IH on $v \rightrightarrows v''$, noticing instead that parallel reducing values are congruent (Lemma C.10) and so $v \leftrightarrows v''$. Since reduction is substitutive (Corollary C.7), we can find that $e[v/x] \hookrightarrow^* e'[v/x]$. Since congruence is substitutive (Lemma C.9), we have $e'[v/x] \leftrightarrows e''[v''/x]$, as desired.
eq1 $(==_b)\ c_1 \rightrightarrows (==_{(c_1,b)})$. We have $(==_b)\ c_1 \hookrightarrow (==_{(c_1,b)})$ in a single step; we find congruence by const.
eq2 $(==_{(c_1,b)})\ c_2 \rightrightarrows c_1 = c_2$. We have $(==_{(c_1,b)})\ c_2 \hookrightarrow c_1 = c_2$ in a single step; we find congruence by const. □

Lemma C.12 (Congruence to a value implies reduction to a value). *If $e \leftrightarrows v'$ then $e \hookrightarrow^* v$ such that $v \leftrightarrows v'$.*

Proof. By induction on v'.

- $v' \doteq c$. It must be the case that $e = c$. Let $v = c$. By const.
- $v' \doteq \lambda x{:}\tau'.\ e''$. It must be the case that $e = \lambda x{:}\tau.\ e'$ such that $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e''$. By lam.
- $v \doteq \mathsf{bEq}_b\ e_l'\ e_r'\ v_p'$. It must be the case that $e = \mathsf{bEq}_b\ e_l\ e_r\ e_p$ where $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e_p \rightrightarrows v_p'$. Since parallel reduction implies reduction to congruent forms (Lemma C.11), we have $e_p \hookrightarrow^* e_p'$ and $e_p' \leftrightarrows v_p'$. By the IH on v_p', we know that $e_p' \hookrightarrow^* v_p$ such that $v_p \leftrightarrows v_p'$.

By repeated use of ctx, we find $\mathsf{bEq}_b\ e_l\ e_r\ e_p \hookrightarrow^* \mathsf{bEq}_b\ e_l\ e_r\ v_p$. Since its proof part is a value, this term is a value. We find $\mathsf{bEq}_b\ e_l\ e_r\ v_p \rightleftarrows \mathsf{bEq}_b\ e_l'\ e_r'\ v_p'$ by ebeq.

- $v \doteq \mathsf{xEq}_{x:\tau_x' \to \tau}\ e_l'\ e_r'\ v_p'$. It must be the case that $e = \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e_p$ where $\tau_x \rightrightarrows \tau_x'$ and $\tau \rightrightarrows \tau'$ and $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$ and $e_p \rightrightarrows v_p'$. Since parallel reduction implies reduction to congruent forms (Lemma C.11), we have $e_p \hookrightarrow^* e_p'$ and $e_p' \rightleftarrows v_p'$. By the IH on v_p', we know that $e_p' \hookrightarrow^* v_p$ such that $v_p \rightleftarrows v_p'$. By repeated application of ctx, we find $\mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e_p \hookrightarrow^* \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ v_p$. Since its proof part is a value, this term is a value. We find $\mathsf{xEq}_{\tau_x:\tau \to}\ e_l\ e_r\ v_p \rightleftarrows \mathsf{xEq}_{x:\tau_x' \to \tau'}\ e_l'\ e_r'\ v_p'$ by exeq. □

Corollary C.13 (Parallel reduction to a value implies reduction to a related value). *If $e \rightrightarrows v'$ then there exists v such that $e \hookrightarrow^* v$ and $v \rightleftarrows v'$.*

Proof. Since parallel reduction implies reduction to congruent forms (Lemma C.11), we have $e \hookrightarrow^* e'$ such that $e' \rightleftarrows v'$. But congruence to a value implies reduction to a value (Lemma C.12), so $e' \hookrightarrow^* v$ such that $v \rightleftarrows v'$. By transitivity of reduction, $e \hookrightarrow^* v$. □

Lemma C.14 (Congruence is a backward simulation). *If $e_1 \rightleftarrows e_2$ and $e_2 \hookrightarrow e_2'$ then there exists e_1' where $e_1 \hookrightarrow^* e_1'$ such that $e_1' \rightleftarrows e_2'$.*

Proof. By induction on the derivation of $e_2 \hookrightarrow e_2'$.

ctx $\mathcal{E}[e] \hookrightarrow \mathcal{E}[e']$, where $e \hookrightarrow e'$.
- $\mathcal{E} \doteq \bullet$. By the outer IH.
- $\mathcal{E} \doteq \mathcal{E}_1\ e_2$. It must be that $e_1 = e_{11}\ e_{12}$, where $e_{11} \rightrightarrows \mathcal{E}_1[e]$ and $e_{12} \rightrightarrows e_2$. By the IH on $\mathcal{E}_1$, finding evaluation with ctx and congruence with app.
- $\mathcal{E} \doteq v_1'\ \mathcal{E}_2$. It must be that $e_1 = e_{11}\ e_{12}$, where $e_{11} \rightrightarrows v_1'$ and $e_{12} \rightrightarrows \mathcal{E}_2[e_2]$. We find that $e_{11} \hookrightarrow^* v_1$ such that $v_1 \rightleftarrows v_1'$ by Corollary C.13. By the IH on $\mathcal{E}_2$ and evaluation with ctx and congruence with app.
- $\mathcal{E} \doteq \mathsf{bEq}_b\ e_l'\ e_r'\ \mathcal{E}'$. It must be the case that $e_1 = \mathsf{bEq}_b\ e_l\ e_r\ e_p$ where $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$. By the IH on $\mathcal{E}'$; we find the evaluation with ctx and congruence with beq.
- $\mathcal{E} \doteq \mathsf{xEq}_{x:\tau_x' \to \tau'}\ e_l'\ e_r'\ \mathcal{E}'$. It must be the case that $e_1 = \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e_p$ such that $\tau_x \rightrightarrows \tau_x'$ and $\tau \rightrightarrows \tau'$ and $e_l \rightrightarrows e_l'$ and $e_r \rightrightarrows e_r'$. By the IH on $\mathcal{E}'$; we find the evaluation with ctx and congruence with xeq.

β $(\lambda x{:}\tau'.\ e')\ v' \hookrightarrow e'[v'/x]$. Congruence implies that $e_1 = e_{11}\ e_{12}$ such that $e_{11} \rightrightarrows \lambda x{:}\tau'.\ e'$ and $e_{12} \rightrightarrows v'$. Parallel reduction to a value implies reduction to a congruent value (Corollary C.13), $e_{11} \hookrightarrow^* v_{11}$ such that $v_{11}' \rightleftarrows \lambda x{:}\tau'.\ e'$, i.e., $v_{11} = \lambda x{:}\tau.\ e$ such that $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e'$. Similarly, $e_{12} \hookrightarrow^* v$ such that $v \rightleftarrows v'$.
By β, we find $(\lambda x{:}\tau.\ e)\ v \hookrightarrow^* e'[v/x]$; by transitivity of reduction, we have $e_1 = e_{11}\ e_{12} \hookrightarrow^* e'[v/x]$. Since congruence is substitutive (Corollary C.9), we have $e[v/x] \rightleftarrows e'[v'/x]$.

eq1 $(==_b)\ c_1 \hookrightarrow (==_{(c_1,b)})$. Congruence implies that $e_1 = e_{11}\ e_{12}$ such that $e_{11} \rightrightarrows (==_b)$ and $e_{12} \rightrightarrows c_1$. Parallel reduction to a value implies reduction to a related value (Corollary C.13), $e_{11} \hookrightarrow^* v_{11}$ such that $v_{11} \rightleftarrows (==_b)$ (and similarly for e_{12} and c_1). But the each constant is congruent only to itself, so $v_{11} = (==_b)$ and $v_{12} = c_1$. We have $(==_b)\ c_1 \hookrightarrow (==_{(c_1,b)})$ by assumption. So $e_1 = e_{11}\ e_{12} \hookrightarrow^* (==_{(c_1,b)})$ by transitivity, and we have congruence by const.

eq2 $(==_{(c_1,b)})\ c_2 \hookrightarrow c_1 = c_2$. Congruence implies that $e_1 = e_{11}\ e_{12}$ such that $e_{11} \rightrightarrows (==_{(c_1,b)})\ c_2$ and $e_{12} \rightrightarrows c_2$. Parallel reduction to a value implies reduction to a related value (Corollary C.13), $e_{11} \hookrightarrow^* v_{11}$ such that $v_{11} \rightrightarrows (==_{(c_1,b)})\ c_2$ (and similarly for e_{12} and c_2). But the each constant is congruent only to itself, so $v_{11} = (==_{(c_1,b)})\ c_2$ and $v_{12} = c_2$. We have $(==_{(c_1,b)})\ c_2 \hookrightarrow c_1 = c_2$ already, by assumption. So $e_1 = e_{11}\ e_{12} \hookrightarrow^* c_1 = c_2$ by transitivity, and we have congruence by const. □

Corollary C.15 (Parallel reduction is a backward simulation). *If* $e_1 \rightrightarrows e_2$ *and* $e_2 \hookrightarrow e_2'$, *then there exists* e_1' *such that* $e_1 \hookrightarrow^* e_1'$ *and* $e_1' \rightrightarrows e_2'$.

Proof. Parallel reduction implies reduction to congruent forms, so $e_1 \hookrightarrow^* e_1'$ such that $e_1' \rightsquigarrow e_2$. But congruence is a backward simulation (Lemma C.14), so $e_1' \hookrightarrow^* e_1''$ such that $e_1'' \rightsquigarrow e_2'$. By transitivity of evaluation, $e_1 \hookrightarrow^* e_1''$. Finally, congruence implies parallel reduction (Lemma C.8), so $e_1'' \rightrightarrows e_2'$, as desired. □

C.4 Cotermination

Theorem C.16 (Cotermination at constants). *If* $e_1 \rightrightarrows e_2$ *then* $e_1 \hookrightarrow^* c$ *iff* $e_2 \hookrightarrow^* c$.

Proof. By induction on the evaluation steps taken, using direct reduction in the base case (Corollary C.13) and using parallel reduction as a forward and backward simulation (Lemmas C.5 and Corollary C.15) in the inductive case. □

Corollary C.17 (Cotermination at constants (multiple parallel steps)). *If* $e_1 \rightrightarrows^* e_2$ *then* $e_1 \hookrightarrow^* c$ *iff* $e_2 \hookrightarrow^* c$.

Proof. By induction on the parallel reduction derivation. The base case is immediate ($e_1 = e_2$); the inductive case follows from cotermination at constants (Theorem C.16) and the IH. □