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, SMT-backed refinement type systems with type inference treat naïve phrasings of functional extensionality inadequately, leading to *unsoundness*. We show how to extend a refinement type theory with a type-indexed propositional equality that is adequate for SMT. We implement our theory in `PEq`, a Liquid Haskell library that defines propositional equality and apply `PEq` to several small examples and two larger case studies. Our implementation proves metaproperties inside Liquid Haskell itself using an unnamed folklore technique, which we dub 'classy induction'.

Additional Key Words and Phrases: refinement types, function equality, function extensionality

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 function (e.g., maps and folds) or do the proofs need to be redone for each fully applied context? Without functional extensionality (a/k/a funext), one must duplicate proofs for each higher-order function. 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 that allow for refinement types, are no exception: function equalities come up regularly. But, in both these systems, the first attempt to give an axiom for functional extensionality was inadequate,[1] A naïve funext axiom unsoundly proves equalities between unequal functions.

Our first contribution is to expose why a naïve function equality encoding is inadequate (§2). At first sight, function equality can be encoded as a refinement type stating that for functions `f` and `g`, if we can prove that `f x` equals `g x` for all `x`, then the functions `f` and `g` are equal:

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

(The 'refinement proposition' `{e}` is equivalent to `{_:() | e}`.) On closer inspection, `funext` does not encode function equality, since it is not reasoning about equality on the domains of the functions. What if type inference instantiates the domain type parameter `a`'s refinement to an intersection of the domains of the input functions or, worse, to an uninhabited type? Would such an instantiation of `funext` still prove equality of the two input functions? We explore the inadequacy of this naïve

[1] See https://github.com/FStarLang/FStar/issues/1542 for F*'s initial, inadequate encoding and the corresponding unsoundness. The Liquid Haskell case is elaborated in §2. See §7 for a discussion of F*'s different solution.

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

extensionality axiom in detail (§2). We work in Liquid Haskell, but the problem generalizes to any refinement type system that allows for polymorphism, semantic subtyping, and refinement type inference. Sound proofs of function equality must carry information about the domain type on which the compared functions are considered equal.

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

Our third contribution is to implement λ^{RE} as a Liquid Haskell library (§4). We implement λ^{RE}'s type-indexed propositional equality using Haskell's GADTs and Liquid Haskell's refinement types. We call the propositional equality `PEq` and find that it adequately reasons about function equality. Further, we prove in Liquid Haskell *itself* that the implementation of `PEq` is an equivalence relation, i.e., it is reflexive, symmetric, and transitive. To conduct these proofs—which go by induction on the structure of the type index—we applied an heretofore-unnamed folklore proof methodology, which we dub *classy induction*. Classy induction encodes theorems as typeclass definitions, where proofs by induction on types give an instance definition for each case of the inductive proof (§4.2; §7).

Our fourth and final contribution is to use `PEq` to prove equalities between functions (§5; §6). 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 ranges, lift equalities to higher-order contexts (i.e., `map`), prove equivalences with multi-argument higher-order functions (i.e., `fold`), and showcase how higher-order, propositional equalities can co-exist with and speedup executable code. We also provide two more substantial case studies, proving the monoid laws for endofunctions and the monad laws for reader monads.

2 THE PROBLEM: NAIVE FUNCTION EXTENSIONALITY IS INSOLUBLE

Refinement types, as used for theorem proving [Vazou et al. 2018a], work naturally with first-order equalities. For instance, consider two functions `h` and `k` with equable ranges and a `lemma` that encodes that for each input `x` the functions `h` and `k` return the same result.[2]

```
h, k :: Eq b => a → b                         lemma :: x:a → { h x == k x }
```

An instantiation of the above lemma might express that fast and slow implementations of the same algorithm (e.g., list reversal) return the same output for every input. Since programmers care about performance, such optimization statements are common in refinement typing. Proving such a lemma justifies substituting fast implementations for slow ones—either manually or via rewrites in GHC using the `rules` pragma [Peyton Jones et al. 2001].

The equality expressed by `lemma` is more-or-less first-order, making use of `Eq b`. Without functional extensionality, we cannot lift the equality in `lemma` to a higher ordering setting, e.g., we can't show that common higher-order functions, like `map :: (a → b) → [a] → [b]` or `first :: (a → b) → (a,c) → (b,c)` behave equivalently when applied to `h` or `k`, even though we know that `h` and `k` behave the same on all inputs. As it stands, to prove statements like `map h xs == map k xs` for all lists `xs` or `first h p == first k p` for all pairs `p`, one must duplicate the proof of `lemma` in the context of `map` and `first`, respectively.

In the small, duplicated proofs are merely annoying. But in the large, duplicated proofs are an engineering impediment, making it hard to iterate on designs, change implementations, or introduce new operations. Without extensionality, it is hard—or even impossible—to do proofs about higher-order definitions behind abstraction barriers, e.g., proving the monad laws for readers.

[2] The `(==)` in the refinements represents SMT interpreted equality. In this paper (unlike the Liquid Haskell implementation) we assume that `(==)` in the refinements also imposes the required `Eq` constraints. Haskell's equality `(==)` appearing in code is approximated by SMT equality using the assumed refinement type presented in §4.4.

In an ideal world we would be able to use `lemma` to derive that the functions `h` and `k` are equal in any context, with no duplicated proofs at all. Liquid Haskell already has two kinds of equality, but neither yields a meaningful *function equality*; concretely, that means we need: a *syntax* for expressing function equality (§2.1), an *axiom* for proving function equality (we'll use extensionality; §2.2), and a *system of checks* that is adequate for function extensionality (§2.3).

2.1 Syntax of Equality between Functions in the Refinements.

We want to name and use equalities between functions in refinement types and proofs, but we must be careful to distinguish our extensional equality from the definitional equalities found in SMT and Haskell. So as a first step, we need a symbol that signifies that two functions are extensionally equal. A single equal sign (=) is interpreted as SMT's definitional equality; a double equal sign (==) is interpreted as Haskell's `Eq` instances' computational equality. We use the symbol ($\doteq$) to signify a functionally extensional propositional equality. We leave $\doteq$ uninterpreted in SMT and without computational interpretation in Haskell.

Function Equality in SMT. Function equality in the SMT world is flexible. The SMT-LIB standard [Barrett et al. 2010] defines the equality symbol = and does not explicitly forbid equality between functions. In fact, CVC4 allows for function 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`.

Function Equality in Haskell. Functional equality is, by default, unutterable in Haskell. Haskell's equality `(==)` has an `Eq` typeclass constraint: `(==) :: Eq a => a → a → Bool`. A sound, general typeclass instance `Eq (a → b)` cannot be provided, since function equality isn't computable.

Function Equality in Refinements. Here, we introduce `(≐)` to denote a new, propositional equality that can relate functions. You can only write `(≐)` in refinements because it does not have computational content. Using separate syntax offers several advantages. First, we won't confuse our extensional equality with Haskell's computational equality `(==)` or SMT equality `(=)`. Second, by distinguishing `(≐)` from other notions of equality, we can leave our extensional equality uninterpreted in SMT. Since different SMT implementations reason differently about function equality, leaving $\doteq$ uninterpreted keeps function equality independent of the underlying SMT implementation's representation of functions.

2.2 Expressing of Naïve Function Extensionality

Equipped with a syntax for function equality in the refinements, the next step is to generate proofs of `f ≐ g`. We begin with a *non*-solution: simply adding an extensionality axiom. In short, a naïve extensionality axiom loses type information that in turn leads to unsoundness. Our solution defines a propositional equality that tracks the appropriate type information, using `Eq` at base types and function extensionality at higher types (§3; §4).

Naïve Extensionality as a Refinement Type. A natural (but, unfortunately, inadequate) approach is to encode functional extensionality (funext) as a refinement type whose postcondition generates function equalities. We can express the extensionality axiom as a refinement type as follows:

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

That is, given functions `f` and `g` and a proof that forall `x`, `f x` equals `g x`, then `f` is equal to `g`. (We use (==) in the proof for now to avoid questions about base type equality.)

```
{-@ assume funext :: Eq b
              => f:(a → b) → g:(a → b) → (x:a → {f x == g x}) → {f ≘ g} @-}
funext _f _g _pf = ()

{-@ allFunEq :: Eq b => h:(a → b) → k:(a → b) → {h ≘ k} @-}
allFunEq h k = funext h k (\_ → ())
```

```
{-@ reflect add1 @-}
add1 :: Int → Int
add1 x = x + 1

{-@ unsound :: { add1 ≘ add2 } @-}
unsound = allFunEq add1 add2
```

```
{-@ reflect add2 @-}
add2 :: Int → Int
add2 x = x + 2

-- (≘) is an SMT uninterpreted function
{-@ measure (≘) :: a → a → Bool @-}
```

Fig. 1. Naïve extensionality proofs gone bad: a proof of add1 ≘ add2 is marked SAFE by Liquid Haskell.

Extensionality can be assumed by the refinement system, but cannot be proved, i.e., we can't actually define a well typed implementation for `funext`. Type theory typically has to axiomatize extensionality (or something stronger, like univalence). Refinement type systems need to use an axiom, too. Why? First, there is no available value of type `a` to "unlock" the `f x == g x` proof argument. And even if the `f x == g x` statement were available, it is not sufficient to generate the `f ≘ g` proof, since `(≘)` is treated as uninterpreted in the logic. To give an uninterpreted symbol any actual meaning in the SMT logic, one *must* use an axiom.

Using `funext`. If two functions produce equal outputs for each input, `funext` proves those functions are equal. `funext` is easy enough to assume in Liquid Haskell (Figure 1, top). Unfortunately, this naïve framing is inadequate and leads to unsound proofs (Figure 1, unsound). Why?

The naïve extensionality axiom loses critical information. Type inference will select a refinement of `false` for `allFunEq`'s domain (Figure 1), as it is the strongest possible type given the constraints—we explain the details below. All functions with a trivial domain are equal, so the inadequate `funext`proves that arbitrary h and k are equal. Finally, `allFunEq` is used by unsound to equate two clearly unequal functions: one increases its argument by 1 and the other by 2!

2.3 Refinement Type Checking of Naïve Function Extensionality is Inadequate

The naïve extensionality axiom leads to unsoundness (Figure 1) due to an interaction with type inference and subtyping. In order to explain the issue, we abstract our concrete Liquid Haskell counterexample into a *generic* refinement type checking system with semantic subtyping (basing concrete details on Liquid Haskell, though other systems work similarly [Barthe et al. 2015; Knowles and Flanagan 2010]). Consider two functions h and k of type $\alpha \rightarrow \beta$ with different domain ($d_\mathsf{h}/d_\mathsf{k}$) and range ($r_\mathsf{h}/r_\mathsf{k}$) refinements.[3] Suppose we've proved a lemma `lemma` that proves some property p relating h and k for all x of type α:

$$\mathtt{h} :: x : \{v : \alpha \mid d_\mathsf{h}\} \rightarrow \{v : \beta \mid r_\mathsf{h}\}$$
$$\mathtt{k} :: x : \{v : \alpha \mid d_\mathsf{k}\} \rightarrow \{v : \beta \mid r_\mathsf{k}\} \qquad\qquad \mathtt{lemma} :: x : \alpha \rightarrow \{p\}$$

[3]We are indeed considering a heterogeneous equality—a natural possibility when using unrefined types (as in the naïve extensionality axiom). Our solution indexes our propositional equality by type (§3).

What might the predicate p be? We could define p as `h` x `==` `k` x, i.e., `h` and `k` produce equal results even outside their prescribed domains. Alternatively, we could restrict p, saying d_{h} `=>` `h` x `==` `k` x, i.e., the two functions are equal only on `h`'s domain, $\{v : \alpha \mid d_{\mathsf{h}}\}$.

Using our naïve extensionality axiom, `funext`, we produce an equality between the two functions:

```
theoremEq :: { h ≐ k }
theoremEq = funext h k lemma
```

If `funext` adequately captures functional extensionality, `theoremEq` should pass the refinement type checker iff `lemma` correctly showed equalities between the results of `h` and `k` on all inputs.

The critical question is: which inputs x should we consider? In our statement of `lemma`, we leave the type of x unrefined—a bare α. By refining α or restricting p, we can restrict the set of xs we consider. The way Liquid Haskell implements semantic subtyping leads to a bad situation: `funext h k lemma` passes the refinement type checker *iff* `lemma` proves first-order equality of the functions `h` and `k` *on some subset* of their domains. Liquid Haskell will choose the smallest subset possible—$\{v : \alpha \mid \texttt{false}\}$—and so calls to `funext` trivially pass. How does this happen?

Desugaring Calls to Extensionality. First, we desugar type inference and typeclass instantiation. After desugaring, the explicit `theoremEq` looks like the following:

```
theoremEq :: { h ≐ k }
theoremEq = funext @{v : α | κ_α} @{v : β | κ_β} d h k lemma
```

The instantiated types α and β are inferred by GHC using its ordinary, unrefined type inference; the dictionary `d` for the `Eq b` constraint is inferred by GHC using typeclass elaboration and constraint solving. Liquid Haskell (but not F*) will infer refinements for the type variables, refining the α and β to $\{v : \alpha \mid \kappa_\alpha\}$ and $\{v : \beta \mid \kappa_\beta\}$, where κ_α and κ_β are *refinement variables* to be resolved during liquid type inference [Rondon et al. 2008].

The core issue, explained at length below, is that these refinement variables will be set to `false`. So $\{v : \alpha \mid \kappa_\alpha\}$ and $\{v : \beta \mid \kappa_\beta\}$ will be trivial, empty types. But all functions to and from empty types are equivalent... meaning `lemma` is irrelevant! Worse still, `theoremEq` proves a *general* equality between `h` and `k`, which can be used outside of the (trival) type at which it was proved, leading to unsoundness (Figure 1, `allFunEq`, `unsound`).

Checking Desugared Calls. After type inference and desugaring, the desugared call is given to the refinement type checker. The derivation is not uncomplicated (see Appendix A, Figure 11 for a full derivation), but at core it only involves invoking basic expression and type application rules, with a few subtyping derivations (Sub-* of Figure 2).

Figure 2 presents the structure of derivation tree that reduces type checking of `theoremEq` to three subtyping rules; we name these subderivations Sub-H, Sub-K, and Sub-L. The expression-level application rule we use is nearly the usual dependent one; the only wrinkle is *subtyping*, which isn't always present in dependent type systems (Figure 5, T-App).

Refinement Subtyping. There are three uses of subtyping in play here: we name the derivations Sub-H, Sub-K, and Sub-L. All of them are instances of subtyping on function types, which uses the standard contravariant subtyping rule (Figure 2, top, Sub-Fun).

Subtyping on refined types reduces to implication checking: to find $\Gamma \vdash \{v : \alpha \mid r_1\} \preceq \{v : \alpha \mid r_2\}$ the top-level refinements in Γ, together with the refinement r_1 of the left-hand-side should imply the refinement r_2 of the right-hand-side. We write the implications to be checked using $\Rightarrow$; implication checks appear at the leaves of every subtyping derivation (Figure 2, top, Sub-B).

Sybtyping Rules

$$\frac{\text{“top-level-refinements of }\Gamma\text{”} \wedge r_1 \Rightarrow r_2}{\Gamma \vdash \{v:\alpha \mid r_1\} \preceq \{v:\alpha \mid r_2\}}\ \textsc{Sub-B} \qquad \frac{\Gamma \vdash \tau'_x \preceq \tau_x \qquad \Gamma, x:\tau'_x \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'}\ \textsc{Sub-Fun}$$

Subtyping Derivation Leaves

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

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

$$\frac{\dfrac{\kappa_\alpha \Rightarrow \mathtt{true}}{\Gamma \vdash \{v:\alpha \mid \kappa_\alpha\} \preceq \alpha} \qquad \dfrac{\kappa_\alpha \Rightarrow p \Rightarrow h\ x == k\ x}{\Gamma, x:\{v:\alpha \mid \kappa_\alpha\} \vdash \{p\} \preceq \{h\ x == k\ x\}}}{\Gamma \vdash x:\alpha \to \{p\} \preceq x:\{v:\alpha \mid \kappa_\alpha\} \to \{\mathsf{h}\ x == \mathsf{k}\ x\}}\ \textsc{Sub-L}$$

Definitions

$$\begin{array}{rcl}
\tau_g & \doteq & \{v:\alpha \mid \kappa_\alpha\} \to \{v:\beta \mid \kappa_\beta\} \\
\Gamma & \doteq & \{\quad \mathtt{funext} : \forall a\, b. \mathtt{Eq}\ b \Rightarrow f:(a \to b) \to g:(a \to b) \to (x:a \to \{f\ x == g\ x\}) \to \{f \simeq g\} \\
 & & ,\quad \mathsf{h} : x:\{v:\alpha \mid d_{\mathsf{h}}\} \to \{v:\beta \mid r_{\mathsf{h}}\}, \mathsf{k} : x:\{v:\alpha \mid d_{\mathsf{k}}\} \to \{v:\beta \mid r_{\mathsf{k}}\} \\
 & & ,\quad \mathtt{lemma} : x:\alpha \to \{p\}, \mathsf{d} : \mathtt{Eq}\ \ \alpha \quad \}
\end{array}$$

Derivation Structure

$$\frac{\dfrac{\dfrac{\boxed{\textsc{Sub-H}} \quad \ldots}{\Gamma \vdash e :: g{:}\tau_g \to (x:\{v:\alpha \mid \kappa_\alpha\} \to \{\mathsf{h}\ x == g\ x\}) \to \{h \simeq g\}} \quad \boxed{\textsc{Sub-K}} \quad \ldots}{\Gamma \vdash e\ \mathsf{k} :: (x:\{v:\alpha \mid \kappa_\alpha\} \to \{\mathsf{h}\ x == \mathsf{k}\ x\}) \to \{h \simeq k\}} \quad \boxed{\textsc{Sub-L}} \quad \ldots}{\Gamma \vdash \underbrace{\mathtt{funext}\ @\{v:\alpha \mid \kappa_\alpha\}\ @\{v:\beta \mid \kappa_\beta\}\ \mathsf{d}\ \mathsf{h}}_{e}\ \mathsf{k}\ \mathtt{lemma} :: \{h \simeq k\}}$$

Fig. 2. Part of type checking of naïve extensionality in `theoremEq` (see full in Appendix Fig. 11).

Implication Checking. Collecting the implications from the subtyping derivations (Figure 2, rules Sub-H, Sub-K, and Sub-L), the checks done for `theoremEq` amount to checking the validity of a relatively small set of implications:

$$\begin{array}{llllll}
(1) & \kappa_\alpha \Rightarrow d_{\mathsf{h}} & (2) & \kappa_\alpha \Rightarrow d_{\mathsf{k}} & (3) & \kappa_\alpha \Rightarrow \mathtt{true} \\
(4) & \kappa_\alpha \Rightarrow r_{\mathsf{h}} \Rightarrow \kappa_\beta & (5) & \kappa_\alpha \Rightarrow r_{\mathsf{k}} \Rightarrow \kappa_\beta & (6) & \kappa_\alpha \Rightarrow p \Rightarrow h\ x == k\ x
\end{array}$$

The predicates d_{h} and d_{k} represent the functions' domains, r_{h} and r_{k} represent the functions' ranges, and p captures the first order equality predicate. The variables κ_α and κ_β are the refinements on the domain and range of the instantiation of `funext`, the naïve extensionality axiom.

On the surface, the implication system seems like a good encoding. Implications (1) and (2) ensures κ_α is at least as restrictive as the two functions' domains. Assuming κ_α, implications (4) and (5) assign to ensure κ_β is at least as inclusive as the two functions' ranges. So far, so good: we've correctly implemented contravariance of functions. Finally, implication (6) requires that κ_α and the property p jointly imply first order equality of the two applications, $h\ x == k\ x$. To sum up: if we can find a common domain, the implication system will check that every application of the two functions on that domain yields equal results. If the domains d_{k} and d_{h} unify to κ_α, the implication system adequately *checks* function extensionality. Unfortunately, type inference will choose a meaningless domain. Later, we forget that choice of trivial domain and unsoundly apply the equality at any domain.

The implication system has a trivial solution: set κ_α to `false`. Such a solution is valid: choosing `false` as the subset of the two functions' domains, the check always succeeds. Liquid type inference [Rondon et al. 2008] always returns the strongest solution for the refinement variables, and so it will always set κ_α to `false`. Setting κ_α to `false` is natural enough in light of `funext`'s type. The function domain α only appears in positive positions. Since functions are contravariant, `funext` never actually touches a value of type α—so Liquid Haskell (soundly!) infers the strongest possible refinement, setting κ_α to `false` meaning that a value of such type is never actually used.

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

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

3 THE SOLUTION: EXPLICIT ENCODING OF TYPED EQUALITY

We formalize a core calculus λ^{RE} with *R*efinement types and type-indexed propositional *E*quality. First, we define the syntax and dynamic semantics of the language (§3.1). Next, we define the typing judgement and a logical relation characterizing equivalence of λ^{RE} expressions (§3.2.1). Finally, we prove that λ^{RE} is semantically sound, and that both the logical relation and the propositional equality satisfy the three equality axioms (§3.3).

3.1 Syntax and Semantics of λ^{RE}

λ^{RE} is a core calculus with *R*efinement types extended with typed *E*quality primitives (Figure 3).

Expressions. Expressions of λ^{RE} include constants for booleans, unit, and equality on base types, variables, lambda abstraction, and application. The expressions also include two primitives to prove propositional equality: we use $\texttt{bEq}_b$ to construct proofs of equality at base types and $\texttt{xEq}_{x:\tau_x \to \tau}$ to construct proofs of equality at function types. Equality proofs take three arguments: the two expressions equated and a proof of their equality; proofs at base type are trivial, of type `()`, but higher types use functional extensionality.

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

Types. The base types of λ^{RE} are booleans and unit. These types aren't used directly; we always refine them with boolean expressions r in *refinement types* $\{x{:}b \mid r\}$, which denote all expressions of base type b that satisfy the refinement r. Types of λ^{RE} 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 $\texttt{PEq}_\tau\ \{e_1\}\ \{e_2\}$, 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 $\{x{:}b \mid \texttt{true}\}$. To keep our formalism and metatheory simple, we omit polymorphic types; we could add them following Sekiyama et al. [2017].

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

Reduction $\boxed{e \hookrightarrow e}$

$$
\begin{array}{rcll}
\mathcal{E}[e] & \hookrightarrow & \mathcal{E}[e'], & \text{if } e \hookrightarrow e' \quad [\text{ctx}] \\
(\lambda x{:}\tau.\ e)\ v & \hookrightarrow & e[v/x] & [\beta] \\
(\mathsf{==}_b)\ c_1 & \hookrightarrow & (\mathsf{==}_{(c_1,b)}) & [\text{eq1}] \\
(\mathsf{==}_{(c_1,b)})\ c_2 & \hookrightarrow & c_1 = c_2, & \textit{syntactic equality on two constants} \quad [\text{eq2}]
\end{array}
$$

Fig. 3. Syntax and Dynamic Semantics of λ^{RE}.

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

Fig. 4. Semantic typing: a unary syntactic logical relation interprets types.

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

Runtime Semantics. The relation $\cdot \hookrightarrow \cdot$ evaluates λ^{RE} expressions using contextual, small step, call-by-value semantics (Figure 3, 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 (Figure 4). We extend it to open terms using closing substitutions (Figure 5).

The interpretation of the base type $\{x{:}b \mid r\}$ includes all expressions which yield b-constants c that satisfy the refinement, i.e., r evaluates to true on c. To decide the unrefined type of an expression we use the relation $\vdash_B e :: b$ (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 (carefully tracking dependency). 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 (noted as xEq_, since the type index does not need to be syntactically equal to the index of the type) whose first two arguments are functions of type $x{:}\tau_x \to \tau$ and the third proof argument takes τ_x arguments to a equality proofs of type $\texttt{PEq}_{\tau[e_x/x]}\ \{e_l\ e_x\}\ \{e_r\ e_x\}$.

Constants. For simplicity in λ^{RE} the constants are only the two boolean values, unit, and equality operators for the two base types. For each base 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 is assigned the type TyCons(c). We assign selfified types to true, false, and unit (e.g., $\{x{:}\texttt{Bool} \mid x ==_{\texttt{Bool}} \texttt{true}\}$) [Ou et al. 2004]. Equality is given a similarly reflective type:

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

Our system could be extended with any constant c, such that $c \in [\![\texttt{TyCons}(c)]\!]$ (Theorem B.1).

3.2 Static Semantics of λ^{RE}

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

3.2.1 Typing of λ^{RE}. We define three mutually recursive judgements for λ^{RE} (Figure 5):

Typing: $\Gamma \vdash e :: \tau$ when the expression e has type τ in the typing environment Γ.
Well formedness: $\Gamma \vdash \tau$ when the type τ is well formed in the typing environment Γ.
Subtyping: $\Gamma \vdash \tau_l \preceq \tau_r$ when an expression with type τ_l can be safely used at type τ_r.

Type Checking. Most of the type checking rules are standard [Knowles and Flanagan 2010; Ou et al. 2004; Rondon et al. 2008]; the T-Eq-Base and T-Eq-Fun rules assign types to proofs of equality.

The rule T-Eq-Base assigns to the expression $\texttt{bEq}_b\ e_l\ e_r\ e$ the type $\texttt{PEq}_b\ \{e_l\}\ \{e_r\}$. To do so, there must be *invariant types* τ_l and τ_r that fit e_l and e_r, respectively. Both these types should be subtypes of b that are strong enough to derive that if $l : \tau_l$ and $r : \tau_r$, then the proof argument e has type $\{\_{:}() \mid l ==_b r\}$. One might expect the proof of equality to be in terms of e_l and e_r themselves rather than general values l and r at invariant types. While we allow selfified types (rule T-Self), our formal model leaves it to the programmer to give strong, meaningful types to terms in proofs of equality. In an implementation like Liquid Haskell, type inference [Rondon et al. 2008] and reflection [Vazou et al. 2018b] automatically derive such strong types.

The rule T-Eq-Fun gives the expression $\texttt{xEq}_{x:\tau_x\to\tau}\ e_l\ e_r\ e$ type $\texttt{PEq}_{x:\tau_x\to\tau}\ \{e_l\}\ \{e_r\}$. As for T-Eq-Base, we use invariant types τ_l and τ_r to stand for e_l and e_r such that with $l : \tau_l$ and $r : \tau_r$, the proof argument e should have type $x{:}\tau_x \to \texttt{PEq}_\tau\ \{l\ x\}\ \{r\ x\}$, 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. The well formedness rule WF-Base checks that the refinement of a base type is a boolean expression. The rule WF-Fun checks that the argument of a function type is well formed and the result is well formed and uses the argument correctly. Finally, the rule WF-Eq checks that the equality type $\texttt{PEq}_\tau\ \{e_l\}\ \{e_r\}$ is well formed, by checking that the index type τ is well formed and that both expressions e_l and e_r have type τ.

Subtyping. The rule S-Base reduces subtyping of basic types to set inclusion on the interpretation of these types (Figure 4). Concretely, for all closing substitutions (as inductively defined by rules C-Empty and C-Subst) the interpretation of the left hand side type should be a subset of the right hand side type. The rule S-Fun implements the usual (dependent) contravariant function subtyping. Finally, S-Eq reduces subtyping of equality types to subtyping of the type indexes, while

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

$$\frac{\Gamma \vdash e :: \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e :: \tau'}\ \textsc{T-Sub} \qquad \frac{\Gamma \vdash e :: \{z{:}b \mid r\}}{\Gamma \vdash e :: \{z{:}b \mid z ==_b e\}}\ \textsc{T-Self} \qquad \frac{}{\Gamma \vdash c :: \mathsf{TyCons}(c)}\ \textsc{T-Con}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau}\ \textsc{T-Var} \qquad \frac{\Gamma, x : \tau_x \vdash e :: \tau \qquad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x{:}\tau_x.\ e :: x{:}\tau_x \to \tau}\ \textsc{T-Lam} \qquad \frac{\Gamma \vdash e :: x{:}\tau_x \to \tau \qquad \Gamma \vdash e_x :: \tau_x}{\Gamma \vdash e\ e_x :: \tau[e_x/x]}\ \textsc{T-App}$$

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

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

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

$$\frac{}{\vdash \emptyset}\ \textsc{WF-Empty} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}\ \textsc{WF-Bind} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash e_l :: \tau \qquad \Gamma \vdash e_r :: \tau}{\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\}}\ \textsc{WF-Eq}$$

$$\frac{\lfloor \Gamma \rfloor, x : b \vdash_B r :: \mathsf{Bool}}{\Gamma \vdash \{x{:}b \mid r\}}\ \textsc{WF-Base} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau}\ \textsc{WF-Fun}$$

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

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

$$\frac{\Gamma \vdash \tau_x' \preceq \tau_x \qquad \Gamma, x : \tau_x' \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau_x' \to \tau'}\ \textsc{S-Fun} \qquad \frac{\Gamma \vdash \tau \preceq \tau' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash \mathsf{PEq}_\tau\ \{e_l\}\ \{e_r\} \preceq \mathsf{PEq}_{\tau'}\ \{e_l\}\ \{e_r\}}\ \textsc{S-Eq}$$

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

$$\frac{}{\emptyset \in [\![\emptyset]\!]}\ \textsc{C-Empty} \qquad \frac{v \in [\![\tau]\!] \qquad \theta \in [\![\Gamma[v/x]]\!]}{x \mapsto v, \theta \in [\![x : \tau, \Gamma]\!]}\ \textsc{C-Subst} \qquad \begin{array}{c}\Gamma \models e \in \tau \\ \Leftrightarrow \\ \forall \theta \in [\![\Gamma]\!],\ \theta \cdot e \in [\![\theta \cdot \tau]\!]\end{array}$$

Fig. 5. Typing of λ^{RE}.

the expressions to be decided equal remain unchanged. Even though covariant treatment of the type index would suffice for our metatheory, we treat the type index bivariantly to be consistent with the implementation (§4) where the GADT encoding of `PEq` is bivariant. Our subtyping rule allows equality proofs between functions with convertible domains and ranges (§5.2).

3.2.2 Equivalence Logical Relation for λ^{RE}. We define characterize equivalence with a term model binary logical, lifting relations on closed values and expressions to an open relation (Figure 6).

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

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

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 $\boxed{\delta \in \Gamma}$ $\boxed{\Gamma \vdash e \sim e :: \tau}$

$$
\begin{array}{lcl}
\delta \in \Gamma & \doteq & \forall x : \tau \in \Gamma,\ \delta_1(x) \sim \delta_2(x) :: \tau;\ \delta \\
\Gamma \vdash e_1 \sim e_2 :: \tau & \doteq & \forall \delta \in \Gamma,\ \delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau;\ \delta
\end{array}
$$

Fig. 6. Definition of equivalence logical relation.

Instead of directly substituting in type indices, all three relations use *pending substitutions* δ, which map variables to pairs of equivalent values.

Closed Value and Expression Equivalence Relations. The relation $v_1 \sim v_2 :: \tau;\ \delta$ states that the values v_1 and v_2 are related under the type τ with and pending substitutions δ. It is defined as a fixpoint on types, noting that $\mathsf{PEq}_\tau\ \{e_1\}\ \{e_2\}$ is structurally larger than τ.

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

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

Open Expression Equivalence Relation. A pending substitution δ satisfies a typing environment Γ when its bindings are related pairs of values. Two open expressions, with variables from a typing environment Γ are equivalent on type τ, written $\Gamma \vdash e_1 \sim e_2 :: \tau$, *iff* for each environment δ that satisfies Γ, $\delta_1 \cdot e_1 \sim \delta_2 \cdot e_2 :: \tau;\ \delta$ holds. The expressions e_1 and e_2 and the type τ might refer to variables in the environment Γ. We use δ to close the expressions eagerly, while we close the type lazily: we apply δ in the refinement and equality cases of the closed value equivalence relation.

3.3 Metaproperties: PEq is an Equivalence Relation

Finally, we show various metaproperties of our system. Theorem 3.1 proves soundness of syntactic typing with respect to semantic typing. Theorem 3.2 proves that propositional equality implies equivalence in the term model. Theorems 3.3 and 3.4 prove that both the equivalence relation and propositional equality define equivalences i.e., satisfy the three equality axioms. All the proofs can be found in Appendix B.

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

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

The proof can be found in Theorem B.2; it 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, S-Base 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, S-Eq allows us to change the type index of propositional equality, but not the term index. Why? Consider $\lambda x{:}\{x{:}\texttt{Bool} \mid \texttt{true}\}.\ \texttt{bEq}_{\texttt{Bool}}\ x\ x\ ()\ e$ such that $e \hookrightarrow e'$ for some e'. The whole application has type $\texttt{PEq}_{\texttt{Bool}}\ \{e\}\ \{e\}$; after we take a step, it has type $\texttt{PEq}_{\texttt{Bool}}\ \{e'\}\ \{e'\}$. Subject reduction demands that the latter is a subtype of the former. We have $\texttt{PEq}_{\texttt{Bool}}\ \{e\}\ \{e\} \rightrightarrows \texttt{PEq}_{\texttt{Bool}}\ \{e'\}\ \{e'\}$, 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.

Theorem 3.2 (PEq is Sound). *If* $\Gamma \vdash e :: \texttt{PEq}_\tau\ \{e_1\}\ \{e_2\}$, *then* $\Gamma \vdash e_1 \sim e_2 :: \tau$.

The proof (see Theorem B.13) is a corollary of the Fundamental Property (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 assumed derivation tree.

Theorem 3.3 (The logical relation is an Equality). $\Gamma \vdash e_1 \sim e_2 :: \tau$ *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 essentially the Fundamental Property. The other proofs proceed by structural induction on the type τ (Theorem B.23). Transitivity requires reflexivity on e_2, so we assume that $\Gamma \vdash e_2 :: \tau$.

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

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

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.

4 IMPLEMENTATION: A GADT FOR TYPED PROPOSITIONAL EQUALITY

We defined propositional equality primitives for base and function types in Liquid Haskell as a GADT (§4.1, Figure 7). Refinements on the GADT enforce the typing rules in our formal model (§3). We used Liquid Haskell itself to establish some of our metatheory (§4.2).

```
-- (1) Plain GADT
data PBEq :: * → * where
     BEq  :: Eq a => a → a → () → PBEq a
     XEq  :: (a → b) → (a → b) → (a → PEq b) → PBEq (a → b)
     CEq  :: a → a → PBEq a → (a → b) → PBEq b

-- (2) Proofs of uninterpreted equality between terms E1 and E2 of type a
{-@ type PEq a E1 E2 = {v:PBEq a | E1 ≏ E2} @-}
{-@ measure (≏) :: a → a → Bool @-}

-- (3) Type refinement of the GADT
{-@ data PBEq  :: * → *  where
         BEq  :: Eq 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. 7. Implementation of the propositional equality PEq as a refinement of Haskell's GADT PBEq.

4.1 The PBEq GADT, its PEq Refinement, and the ≏ Measure

We define our type-indexed propositional equality `PEq a {e1} {e2}` in three steps (Figure 7): (1) structure (à la λ^{RE}) as a GADT, (2) definition of the refined type PEq, and (3) proof construction via a refinement of the GADT.

First, we define the structure of our proofs of equality as PBEq, an unrefined, i.e., Haskell, GADT (Figure 7, (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 Eq instance from Haskell (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 7, (2)). We say that two terms E1 and E2 of type a are propositionally equal when there (a) is a well formed and valid PBEq proof and (b) we have `E1 ≏ E2`, where (≏) is an SMT, uninterpreted function symbol. PEq is defined as a Liquid Haskell type alias that uses capital letters to indicate which formal type parameters in type definitions are expressions, e.g., in `type PEq a E1 E2 = ...`, both E1 and E2 are expressions, but a is a type. Liquid Haskell uses curly braces to indicate which actual arguments in type applications are expressions, e.g., in `PEq a {x} {y}`, both x and y are expressions, but a is a type. Since (≏) is uninterpreted, we can only get `E1 ≏ E2` from axioms or assumptions.

Finally, we refine the type constructors of PBEq to axiomatize the behaviour of (≏) and generate proofs of PEq (Figure 7, (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 == y` according to the Eq instance for a. The metatheory of Liquid Haskell has always assumed that Eq instances correspond to SMT equality.[4] XEq is the `funext` axiom. Given functions f and g of type `a → b`, a proof of equality

[4] This assumption is encoded as the refinement type for (==) of §4.4 and is not actually checked at instance definitions, thus unsoundness might occur when Haskell's Eq instances do not respect the equality axioms.

```
-- (1) Refined typeclass                -- (2) Base case (Eq types)
{-@ class Reflexivity a where            instance Eq a => Reflexivity a where
      refl :: x:a → PEq a {x} {x} @-}      refl a = BEq a a ()

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

Fig. 8. A proof of reflexivity using classy induction.

via extensionality also needs an `PEq`-proof that `f x` and `g x` are equal for all `x` of type `a`. Such a proof has (unrefined) type `a → PBEq b`, with refined type `x:a → PEq b {f x} {g x}`. Critically, we don't lose any type information about `f` or `g`! `CEq` implements congruence closure (§ 4.3) `x` and `y` of type `a` that are equal—i.e., `PEq a {x} {y}`—and an arbitrary context with an `a`-shaped hole (`ctx :: a → b`), filling the context with `x` and `y` yields equal results, i.e., `PEq b {ctx x} {ctx y}`.

4.2 Equivalence Properties and Classy Induction

The metatheory in §3 establishes a variety of meaningful properties of our propositional equality. We were surprised that we could prove some of these properties—reflexivity, symmetry, and transitivity (Theorem 3.4)—within Liquid Haskell itself.

Just as our paper metatheory uses proofs that go by induction on types, our proofs in Liquid Haskell also go by induction on types. But "induction" in Liquid Haskell means writing a recursive function, which necessarily has a single, fixed type. We want a Liquid Haskell theorem `refl :: x:a → PEq a {x} {x}` that corresponds to Theorem 3.4 (a), but the proof goes by induction on the type `a`, which is not a thing an ordinary Haskell function could do.[5]

The essence of our proofs is a folklore method we name *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 `Eq` 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 exactly.

Our proof of reflexivity is exemplary (Figure 8). For (1), the typeclass `Reflexivity` simply states the desired theorem type, `refl :: x:a → PEq a {x} {x}`. For (2), `BEq` suffices to define the `refl` method for those `a` with an `Eq` instance.[6] 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 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 `Eq` instance, the proof of `refl x` will simply be `BEq x x ()`, which SMT checking can trivially discharge. If `a` is a function of type `b → c`, then the compiler will try to find a `Reflexivity` instance for the codomain `c`—and if it finds one, generate a proof using `XEq` and `c`'s proof. The compiler's constraint resolver does the constructive proof for us, assembling a `refl` for our chosen type. Just as our paper metatheory works only for a fixed model, our `refl` proofs only work for types where the codomain bottoms out with an `Eq` instance.

[5] A variety of GHC extensions provide ways to do case analysis on types: type families, `TypeInType`, `Dynamic`, and generics, to name a few. Unfortunately, Liquid Haskell doesn't support these extensions.

[6] To define such a general instance, we enabled two GHC extensions: `FlexibleInstances` and `UndecidableInstances`.

Our proofs of symmetry and transitivity follow this pattern; both use congruence closure. The proofs can be found in supplementary material [2020]. 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.

4.3 Congruence Closure

The standard definition of contextual equivalence says that putting equivalent terms into a context doesn't affect the observable results. Not only do our equivalence-property proofs use `CEq` (e.g., `Symmetry` above), but so do other proofs about function equalities (e.g., the `map` function in §5.3).

Congruence closure is typically proved by induction on the expressions, i.e., following the cases of the fundamental theorem of the logical relation. While classy induction allows us to perform induction on types to prove meta-properties within the language, we have no way to perform induction on terms in Liquid Haskell (Coq can; see discussion of Sozeau's work in §7). Instead, we axiomatize congruence closure with `CEq`, using a function to represent the enclosing context.

4.4 Adequacy with Respect to SMT

Liquid Haskell's soundness depends on closely aligning Haskell and SMT concepts: numbers and data structures port from Haskell to SMT more or less wholesale, while functions are encoded as SMT integers and application is an axiomatized uninterpreted function. Equality is a particularly important point of agreement: SMT and Liquid Haskell should believe the same things are equal! We must be careful to ensure that `PEq` aligns correctly with the SMT solver.

Liquid Haskell now has three notions of equality (§2.1): primitive SMT equality `(=)`, Haskell `Eq`-equality `(==)`, and our new propositional equality, `PEq`. Liquid Haskell conflates `(=)` and `(==)`:

```
{-@ assume (==) :: Eq a => x:a → y:a → {v:Bool | v ⇔ x = y } @-}
```

For base types like `Bool` or `Int`, SMT and Haskell equality really do coincide (up to concerns about, e.g., numerical overflow). Both hand-written and derived structural `Eq` instances on data types coincide with SMT equality, too. From the metatheoretical formal perspective, the connection between Haskell's and SMT's equality comes by the assumption that equality, as well as any Haskell function that corresponds to an SMT-interpreted symbol, belongs to the semantic interpretation of its very precise or selfified type [Knowles and Flanagan 2010; Ou et al. 2004]. That is, to prove a refinement type system with equality sound, we assume $(==) \in [\![x : a \rightarrow y : a \rightarrow \{v{:}\mathrm{Bool} \mid v \Leftrightarrow x = y\}]\!]$.

Unfortunately, custom notions of equality in Haskell can subvert the alignment. For example, an AST might ignore location information for term equality. Or one might define a non-structural `Eq` on a tree-based implementation of sets. Such notions of equality are benignly non-structural, but won't agree with the SMT solver's equality. As a more extreme example, consider the following `Eq` instance on functions that takes the principle of function extensionality a little too seriously:

```
instance (Bounded a, Enum a, Eq b) => Eq (a → b) where
  f1 == f2 = all (\x → f1 x == f2 x) $ enumFromTo minBound maxBound
```

Liquid Haskell's assumed type for `(==)` is unsound for these `Eq` instances.

Our equivalence relation `PEq` is built on `Eq`, so it suffers from these same sources of inadequacy. The edge-case inadequacy of `(==)` has been acceptable so far, but `PEq` complicates the situation by allowing equivalences between functions. Since Liquid Haskell encodes higher-order functions in a numbering scheme, where each function translates to a unique number, the meaning of application for each such numbered function is axiomatized. If we have `PEq (a → b) {f} {g}`, it would be outright unsound to assume `f = g` in SMT: we encode `f` and `g` as different numbers! At the same time, it ought to be the case that if `Eq a` and `PEq a {e1} {e2}`, then `e1 == e2` and so `e1 = e2`.

In the long run, Haskell's `Eq` class should not be assumed to coincide with SMT equality. For now, Liquid Haskell continues to assume that `PEq` at `Eq` types implies SMT equality. Rather than simply adding an axiom, though, we make the axiom a typeclass itself, called `EqAdequate`:

```
{-@ class Eq a => EqAdequate a where
  toSMT :: x:a → y:a → PEq a {x} {y} → {x = y} @-}

instance Eq a => EqAdequate a where
  toSMT _x _y _pf = undefined
```

The `EqAdequate` typeclass constraint lets us know exactly which proofs depend on `Eq` instances being adequate. We use it in the base cases of symmetry and transitivity. For example:

```
instance EqAdequate a => Symmetry a where
  -- sym :: l:a → r:a → PEq a {l} {r} → PEq a {r} {l}
     sym l r pf = BEq r l (toSMT l r pf)
```

The call to `toSMT` transports the proof that `l` and `r` are equal into an SMT equality: `toSMT l r pf :: {l = r}`. The SMT solver easily discharges `BEq`'s `{r = l}` obligation using `{l = r}`.

5 EXAMPLES

We demonstrate our propositional equality in a series of examples. We start by moving from simple first-order equalities to equalities between functions (`reverse`, §5.1). Next, we show how `PEq`'s type indices reason about refined domains and dependent ranges of functions (`succ`, §5.2). Proofs about higher-order functions exhibit the `CEq` contextual equivalence axiom (`map`, §5.3). Next, we see that our type-indexed equality plays well with multi-argument functions (`foldl`, §5.4). Finally, we present how an equality proof can lead to more efficient code (`spec`, §5.5). To save space, we omit the `reflect` annotations from the following code.

5.1 Reverse: from First-Order to Higher-Order Equality

Consider three candidate definitions of the list-reverse function (Figure 9, top): a 'fast' one in accumulator-passing style (`fastReverse`), a 'slow' one in direct style (`slowReverse`), and a 'bad' one that returns the original list (`badReverse`).

First-Order Proofs. It is a relatively easy exercise in Liquid Haskell to prove a theorem relating the two list reversals (Figure 9, bottom; Vazou et al. [2018a]). 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 `slowReverse`. The `lemma` itself uses the inductive lemma `assoc` to show associativity of `(++)`.

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

```
{-@ reverseHO :: Eq a => PEq ([a] → [a]) {fastReverse} {slowReverse} @-}
```

Two implementations (and one non-implementation) of reverse

```
fastReverse :: [a] → [a]                 badReverse :: [a] → [a]
fastReverse xs = fastGo [] xs            badReverse xs = xs

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

Proofs relating `fastReverse` *and* `slowReverse`

```
{-@ reverseEq :: Eq a => xs:[a] → { fastReverse xs == slowReverse xs } @-}
{-@ lemma     :: Eq a => xs:[a] → ys:[a] → {fastGo ys xs == slowReverse xs ++ ys} @-}
{-@ assoc     :: Eq a => xs:[a] → ys:[a] → zs:[a]
              → { (xs ++ ys) ++ zs == xs ++ (ys ++ zs) } @-}
{-@ rightId   :: Eq a => xs:[a] → { xs ++ [] == xs } @-}

reverseEq xs                             lemma []     _  = ()
  = lemma xs []                          lemma (x:xs) ys = lemma xs (x:ys)
  ? rightId (slowReverse xs)               ? assoc (slowReverse xs) [x] ys

assoc []     _  _  = ()                  rightId []     = ()
assoc (_:xs) ys zs = assoc xs ys zs      rightId (_:xs) = rightId xs
```

Fig. 9. Reasoning about list reversal.

```
  reverseHO     = XEq fastReverse slowReverse reversePf
```

The inner `reversePf` shows `fastReverse xs` is propositionally equal to `slowReverse xs` for all `xs`:

```
  {-@ reversePf :: Eq a => xs:[a] → PEq [a] {fastReverse xs} {slowReverse xs} @-}
```

There are several different styles to construct such a proof.

Style 1: Lifting First-Order Proofs. The first order equality proof `reverseEq` can be directly lifted to propositional equality, using the `BEq` constructor.

```
  {-@ reversePf1 :: Eq a => xs:[a] → PEq [a] {fastReverse xs} {slowReverse xs} @-}
  reversePf1 xs = BEq (fastReverse xs) (slowReverse xs) (reverseEq xs)
```

Such proofs are unsatisfying, since `BEq` relies on SMT equality and imposes an `Eq` constraint.

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

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

The proof goes by induction and uses the `Reflexivity` and `Transitivity` properties of `PEq` encoded as typeclasses (§4.2) along with `assocP` and `rightIdP`, the propositional versions of `assoc` and `rightId`. These typeclass constraints propagate to the `reverseHO` proof, via `reversePf2`.

```
{-@ reversePf2 :: (Reflexivity [a], Transitivity [a])
               => xs:[a] → PEq [a] {fastReverse xs} {slowReverse xs} @-}
reversePf2 xs = trans (fastReverse xs) (slowReverse xs ++ []) (slowReverse xs)
                      (lemmaP [] xs) (rightIdP (slowReverse xs))
```

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

```
{-@ reversePf3 :: (Reflexivity [a], Eq a)
               => xs:[a] → PEq [a] {fastReverse xs} {slowReverse xs} @-}
reversePf3 xs = refl (fastReverse xs) ? lemma xs [] ? rightId (slowReverse xs)
```

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

5.2 Succ: Refined Domains and Dependent Ranges

Our propositional equality `PEq`, with standard refinement type checking, naturally reasons about functions with refined domains and dependent ranges. For example, consider the functions `succNat` and `succInt` that respectively return the successor of a natural and integer number.

```
succNat, succInt :: Integer → Integer
succNat x = if x >= 0 then x + 1 else 0
succInt x = x + 1
```

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

```
{-@ type Natural = {x:Integer | 0 <= x } @-}

{-@ natDom :: PEq (Natural → Integer) {succInt} {succNat} @-}
natDom = XEq succInt succNat (\x → BEq (succInt x) (succNat x) ())
```

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

```
{-@ natRng :: PEq (Natural → Natural) {succInt} {succNat} @-}
natRng = XEq succInt succNat natRng'

{-@ natRng' :: Natural →  PEq Natural (succInt x) (succNat x) @-}
natRng' x = BEq (succInt x)  (succNat x) ()
```

Liquid Haskell's type inference forces us to write `natRng'` as a separate, manually annotated term. While `natDom` does not type check with the type of `natRng`, the above definition of `natRng` type checks without refinement annotations on the range of `succNat` and `succInt` themselves.

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

```
{-@ depRng :: PEq (x:Natural → {v:Natural | v == x + 1}) {succInt} {succNat} @-}
depRng = dXEq succInt succNat depRng'

{-@ depRng' :: x:Natural → PEq {v:Natural | v == x + 1} (succInt x) (succNat x) @-}
depRng' x = BEq (succInt x)  (succNat x) ()
```

The proof uses `dXEq`, a dependent version of `XEq` that explicitly captures the range of functions in an indexed, abstract refinement `p` and curries it in the result `PEq` type [Vazou et al. 2013].

```
{-@ assume dXEq :: ∀<p :: a → b → Bool>. f:(a → b) → g:(a → b)
            → (x:a → PEq b<p x> {f x} {g x}) → PEq (x:a → b<p x>) {f} {g}  @-}
dXEq = XEq
```

Note that the above specification is assumed by our library, since Liquid Haskell can't yet parameterize the definition of the GADT `PEq` with abstract refinements.

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

```
{-@ badDom  :: PEq (  Integer → Integer)                  {succInt} {succNat} @-}
{-@ badRng  :: PEq (  Natural → {v:Integer | v < 0 })     {succInt} {succNat} @-}
{-@ badDRng :: PEq (x:Natural → {v:Integer | v == x + 2}) {succInt} {succNat} @-}
```

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

5.3 Map: Putting Equality in Context

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

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

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

Equivalence on an Arbitrary Argument. To show that `map f xs` and `map g xs` are equal for all `xs`, we use `CEq` with a context that puts `f` and `g` in a 'flipped' context. We name this context `flipMap`:

```
{-@ mapEq :: Eq a => 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) ? mapFlipMap f xs ? mapFlipMap g xs

{-@ mapFlipMap :: Eq a => f:(a → b) → xs:[a] → { map f xs == flipMap xs f } @-}
mapFlipMap f xs = ()

flipMap xs f = map f xs
```

The `mapEq` proof relies on `CEq` using the flipped context; SMT will need to know that `map f xs == flipMap xs f`, which is explicitly proved by `mapFlipMap`. Liquid Haskell cannot infer this equality in the higher order setting of the proof, where neither the function `map` nor `flipMap` are fully applied. In supplementary material [2020] we provide an alternative proof of `mapEq` using the typeclass-encoded properties of equivalence.

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

```
{-@ client :: xs:[Natural] → PEq [Integer] {map succInt xs} {map succNat xs} @-}
client = mapEq succInt succNat natDom
{-@ clientP ::  PEq ([Natural] → [Integer]) {map succInt} {map succNat} @-}
clientP = mapEqP succInt succNat natDom
```

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

5.4 Fold: Equality of Multi-Argument Functions

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

```
theorem :: Eq b => (b → a → b) → b → [a] → ()
{-@ theorem :: Eq b => f:_ → b:b → xs:[a] → { foldl f b xs == foldl' f b xs } @-}
```

The proof relies on some outer reasoning and an inductive lemma. The outer reasoning turns `foldl'` into `foldr`; the inductive lemma characterizes the actual invariant in play.

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 :: Eq 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) (theorem f b xs)
```

Interestingly, one can avoid the first-order proof, the `Eq` constraint, and the subsequent conversion via `BEq`. We used the typeclass-encoded properties to directly prove `foldl` equivalence in the propositional setting (à la *Style 2* of §5.1), as expressed by `theoremP` below.

```
{-@ theoremP :: (Reflexivity b, Transitivity b) => f:(b → a → b) → b:b → xs:[a]
             → PEq b {foldl f b xs} {foldr (construct f) id xs b} @-}
```

The proof goes by induction and can be found in supplementary material [2020]. Here, we use `theoremP` to directly prove the equivalence of `foldl` and `foldl'` in the propositional setting.

```
{-@ foldEqP :: (Reflexivity b, Transitivity b)
            => PEq ((b → a → b) → b → [a] → b) {foldl} {foldl'} @-}
foldEqP = XEq foldl foldl' $ \f →
            XEq (foldl f) (foldl' f) $ \b → XEq (foldl f b) (foldl' f b) $ \xs →
              trans (foldl f b xs) (foldr (construct f) id xs b) (foldl' f b xs)
                    (theoremP f b xs) (refl (foldl' f b xs))
```

Just like `foldEq`, the proof calls `XEq` for each but the last argument, replacing `BEq` with transitivity, reflexivity, and the inner theorem in its propositional-equality form.

5.5 Spec: Function Equality for Program Efficiency

Finally, we present an example where function equality is used to soundly optimize runtimes. Consider a `critical` function that, for soundness, can only run on inputs that satisfy a boolean, verification friendly specification, `spec`, and a `fastSpec` as an alternative way to test `spec`.

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

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

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

If the `toSMT` call above was omitted, then the call in the `then` branch would generate a type error: there is not enough information that `critical`'s precondition holds. The `toSMT` call generates the SMT equality that `fastSpec x == spec x`. Combined with the efficient runtime check `fastSpec x`, the type checker sees that in the call to `critical x` is safe in the `then` branch.

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

6 CASE STUDIES

We present two case studies of our propositional equality in action: proving the monoid laws for endofunctions and proving the monad laws for reader monads. These two examples are very much higher order; both are well known and practically important among typed functional programmers. In both case studies, we use classy induction (§4.2) to make our proofs generic over the types returned by the higher-order functions in play (i.e., Style 2 from §5.1).

6.1 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, i.e., $f : \tau \to \tau$ for some τ. 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.

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

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

```
{-@ mLeftIdentity  :: _ => x:Endo a → PEq (Endo a) {mappend mempty x} {x} @-}
{-@ mRightIdentity :: _ => x:Endo a → PEq (Endo a) {x} {mappend x mempty} @-}
{-@ mAssociativity :: _ => x:(Endo a) → y:(Endo a) → z:(Endo a) →
          PEq (Endo a) {mappend (mappend x y) z} {mappend x (mappend y z)} @-}
```

We elide the `Reflexivity` and `Transitivity` constraints required by the proofs as _.

Proving the monoid laws for endofunctions demands `funext`. 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 Haskell's `Eq` or SMT equality. With `funext`, 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.

```
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)
```

The `(=~=)` operator allows for equational style proofs. It is defined as `_ =~= y = y`, unrefined. Liquid Haskell's refinement reflection [Vazou et al. 2018b] unfolds the function definition each time a function is called. For example, in the `mLeftIdentity` proof, the term `mappend mempty x a =~= mempty (x a) =~= x a` unfolds the definitions of `mappend` and `mempty` for the given arguments, which is enough for the SMT solver. The postfix just `*** QED` casts the proof into a Haskell unit. The Liquid Haskell standard library gives `(=~=)` a refined type:

```
{-@ (===) :: Eq a => x:a → y:{a | y == x} → {v:a | v == x && v == y} @-}
```

Refining === checks the intermediate equational steps using SMT equality. In our higher order setting, we cannot use SMT equality on functions, so we use the unrefined =~= in our proofs. We lose the intermediate checks, but the unfolding is sound at all types. Liquid Haskell still conflates `(==)` and `(=)`; in the future, we will further disentangle assumptions about equality (§4.4).

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.

6.2 Monad Laws for Reader Monads

A *reader* is a function with a fixed domain `r`, i.e., the partially applied type `Reader r` (Figure 10, 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 10, 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 10, bottom). Along the way, we also prove that they satisfy the functor and applicative laws in supplementary material [2020]. The reader monad laws are expressed as refinement type specifications using `PEq`. We prove the left and right identities following the pattern of §6.1, i.e., `XEq`, followed by reflexivity with `(=~=)` for function unfolding (Figure 10, middle). We use transitivity to conduct the more complicated proof of associativity (Figure 10, 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 that el = em; local error here: needs trans
  (trans em emr er -- proof that em = er
```

Monad Instance for Readers

```
type Reader r a = r → a                pure :: a → Reader r a
                                       pure a _r = a
kleisli :: (a → Reader r b)
        → (b → Reader r c)             bind :: Reader r a → (a → Reader r b)
        → a → Reader r c                    → Reader r b
kleisli f g x = bind (f x) g           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 =                    monadRightIdentity m =
  XEq (bind (pure a) f) (f a) $ \r →         XEq (bind m pure) m $ \r →
   refl (bind (pure a) f r) ?                  refl (bind m pure r) ?
    (bind (pure a) f r =~= f (pure a r) r       (bind m pure r =~= pure (m r) r
             =~= f a r *** QED)                       =~= 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. 10. Case study: Reader Monad Proofs.

```
    (refl em) {- proof that em = emr -} (refl emr) {- proof that emr = er -})
```

The `refl el` proof will produce a type error; replacing that proof with an appropriate `trans` completes the `monadAssociativity` proof (Figure 10, bottom). Such an approach to writing proofs in this style works well: start with `refl` and where the SMT solver can't figure things out, a local refinement type error tells you to expand with `trans` (or look for a counterexample).

Our reader proofs use the `Reflexivity` and `Transitivity` typeclasses to ensure that readers are monads whatever the return type `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.

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: they state the function equality using ==, which is a 'squashed' (i.e., proof irrelevant) form of **`equals`**, a propositional Leibniz equality. They take it as an assumption that this Leibniz equality coincides with SMT equality, much like Liquid Haskell's assumption that `(==)` and `(=)` align. Liquid Haskell can't directly accommodate the F*approach, since there are no dependent, inductive type definitions nor a dedicated notion of proposition. GADTs offer a limited form of dependency without the full power of F*'s inductive definitions. Our `PEq` GADT approximates F*'s approach, but makes different compromises.

Dafny's SMT encoding axiomatizes extensionality for datatypes, but not for functions [Leino 2012]. Function equality is utterable but neither provable nor disprovable, due to their SMT encoding and how their solver (Z3) treats functions.

Ou et al. [2004] introduce *selfification*, which assigns singleton types using equality. Selfified types have the form $\text{self}(\{x{:}b \mid e_b\}, e) = \{x{:}b \mid e_b \land x = e\}$. Our T-Self rule applies selfified types to arbitrary expressions of base type and our assigned types for constants ($\texttt{TyCons}(c)$) are in selfified form. SAGE assigns selfified types to *all* variables, implying equality on functions [Knowles et al. 2006]. Dminor avoids function equality by not having 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 implements ITT [2020]. Agda's use of axiom K makes it an ETT [Norell 2008]. Extensionality axioms are independent of the rules of ITT; it is not uncommon to axiomatize extensionality. Not every model of type theory is consistent with `funext`, though: von Glehn's polynomial model refutes extensionality [2014, Proposition 4.11]. Pfenning [2001] extends LF's β-equality [Harper et al. 1993] to combine intensional and extensional flavors of type theory in a single, modal framework. Hofmann [1996] shows that ETT is a conservative extension of ITT with `funext` and UIP; introducing these non-computational axioms breaks canonicity. Observational type theory (OTT) generalizes ITT and ETT, retaining canonicity and a computational interpretation [Altenkirch and McBride 2006].

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). Our system has no way to prove equalities between equalities, though adding UIP would be easy. Since our propositional equality isn't exactly Leibniz equality, axiom K would be harder to encode but could use Theorem 3.4's proof of reflexivity as a source for canonical reflexivity proofs. F*'s squashed Leibniz equality is proof-irrelevant and there is at most one equality proof between any given pair of terms.

Zombie [Sjöberg and Weirich 2015] presents a dependently-typed programming language that uses an adaptation of a congruence closure algorithm to automatically reason about equality. Zombie does not use automatic β-reduction, thereby avoiding divergence during type conversion and type checking. Zombie can do some reasoning about equalities on functions (reflexivity; substitutivity inside of lambdas) but cannot show equalities based on bound variables, e.g., they cannot prove that $\lambda x.\ x = \lambda x.\ x + 0$. 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]. We also omit such a rule, but we have `funext`. Unlike many other dependent type theories, we don't use type conversion per se: our definition/judgmental (in)equality is *subtyping*.

The Lean theorem prover's quotient-based reasoning can *prove* `funext` [de Moura et al. 2015]. They do not, however, have a completely computational account.

We suspect that recent ideas around equality from cubical type theory offer 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. In general, univalent systems like cubical type theory get functional extensionality 'for free'—that is, for the price of the univalence axiom or of cubical foundations.

Classy Induction: Inductive Proofs Using Typeclasses. We proved inside Liquid Haskell that our equivalence relation is reflexive, symmetric, and transitive (§4.2). Our proofs are by 'classy induction', using typeclasses to do induction on type structure: we treat types with an `Eq` instance as base cases, while we use `funext` in the inductive cases (function types). Classy induction uses ad-hoc polymorphism and general instances to generate proofs that 'cover' all types. Ad-hoc polymorphism has always allowed for programming over type structure (e.g., the `Arbitrary` and `CoArbitrary` classes in QuickCheck [Claessen and Hughes 2000] cover most types); we only call it 'classy induction' when building up proofs.

We did not *invent* classy induction—it is a folklore technique that we have identified and named. We have seen five independent uses of "classy induction". First, Guillemette and Monnier [2008] speculate that they could eliminate runtime overhead by proving "lemmas over type families". It is not clear whether these lemmas would take the form of induction over types or not. Second, Weirich [2017] constructed the well formedness constraint for occurence maps by induction on lists at the type level. Third, Boulier et al. [2017] define a family of syntactic type theory models for the calculus of constructions with universes (CC_ω). They define a notion of ad-hoc polymorphism that allows for type quoting and definitions by induction-recursion on their theory's (predicative) types. They do not show any examples of its use, but it could be used to generate proofs by classy induction. Fourth, Dagand et al. [2018] use classy induction to generate instances of higher-order Galois connections in their framework for interactive proof. Fifth, and finally, Tabareau et al. [2019] use classy induction to define their univalent parametericity relation for type universes and for each type constructor in Coq. These last two uses of classy induction may require the programmer to 'complete the induction': while built-in and common types have library instances, a user of the library would need to supply instances for their custom types.

Any typeclass system that accommodates ad-hoc polymorphism and a notion of proof can accommodate 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 when proofs are possible [Wenzel 1997].

8 CONCLUSION

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

Connecting type systems with SMT brings great benefits but requires a careful encoding of your program into the logic of the SMT solver. Reconciling host-language equality with SMT equality is a particular challenge. Our propositional equality is a first step towards disentangling host-language computational equality, decidable SMT equality, and the propositional equality used in refinements.

ACKNOWLEDGMENTS

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

REFERENCES

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`.

Gilles Barthe, Marco Gaboardi, Emilio JesÃžs Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL âĂŹ15* (2015). https://doi.org/10.1145/2676726.2677000

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

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP âĂŹ00)*. Association for Computing Machinery, New York, NY, USA, 268âĂŞ279. https://doi.org/10.1145/351240.351266

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)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. 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 (ICFP âĂŹ08)*. Association for Computing Machinery, New York, NY, USA, 75âĂŞ86. https://doi.org/10.1145/1411204.1411218

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (Jan. 1993), 143âĂŞ184. https://doi.org/10.1145/138027.138060

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 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

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

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

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 (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 (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. 2020. 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:cs.PL/1909.05027

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 (Haskell 2018).* ACM, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

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 (POPL âĂŹ89).* 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 (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 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

$$\dfrac{\Gamma(\mathsf{funext}) \;=\; \forall a\, b.\mathsf{Eq}\ b \Rightarrow f : (a \to b) \to g : (a \to b) \to (x : a \to \{f\ x == g\ x\}) \to \{f \simeq g\}}{\Gamma \vdash \mathsf{funext} :: \forall a\, b.\mathsf{Eq}\ b \Rightarrow f : (a \to b) \to g : (a \to b) \to (x : a \to \{f\ x == g\ x\}) \to \{f \simeq g\}}$$

$$\dfrac{\Gamma \vdash \mathsf{funext} :: \forall a\, b.\mathsf{Eq}\ b \Rightarrow f : (a \to b) \to g : (a \to b) \to (x : a \to \{f\ x == g\ x\}) \to \{f \simeq g\}}{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\} :: \forall b.\mathsf{Eq}\ b \Rightarrow f : (\{v : \alpha \mid \kappa_\alpha\} \to b) \to g : (\{v : \alpha \mid \kappa_\alpha\} \to b) \to (x : \{v : \alpha \mid \kappa_\alpha\} \to \{f\ x == g\ x\}) \to \{f \simeq g\}}$$

$$\dfrac{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\} :: \forall b.\mathsf{Eq}\ b \Rightarrow \ldots}{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\} :: \mathsf{Eq}\ \{v : \beta \mid \kappa_\beta\} \Rightarrow f : (\{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}) \to g : (\{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}) \to (x : \{v : \alpha \mid \kappa_\alpha\} \to \{f\ x == g\ x\}) \to \{f \simeq g\}}$$

$$\dfrac{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\} :: \mathsf{Eq}\ \{v : \beta \mid \kappa_\beta\} \Rightarrow \ldots \qquad \dfrac{\Gamma(\mathsf{d}) \;=\; \mathsf{Eq}\ \alpha}{\Gamma \vdash \mathsf{d} :: \mathsf{Eq}\ \alpha} \qquad \dfrac{}{\Gamma \vdash \mathsf{Eq}\ \alpha \;\preceq\; \mathsf{Eq}\ \{v : \beta \mid \kappa_\beta\}}\ \textsc{Sub-D}}{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d} :: f : (\{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}) \to g : (\{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}) \to (x : \{v : \alpha \mid \kappa_\alpha\} \to \{f\ x == g\ x\}) \to \{f \simeq g\}}$$

$$\dfrac{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d} :: \ldots \qquad \dfrac{\Gamma(\mathsf{h}) \;=\; x : \{v : \alpha \mid d_{\mathsf{h}}\} \to \{v : \beta \mid r_{\mathsf{h}}\}}{\Gamma \vdash \mathsf{h} :: x : \{v : \alpha \mid d_{\mathsf{h}}\} \to \{v : \beta \mid r_{\mathsf{h}}\}} \qquad \dfrac{\ldots}{\Gamma \vdash x : \{v : \alpha \mid d_{\mathsf{h}}\} \to \{v : \beta \mid r_{\mathsf{h}}\} \;\preceq\; \{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}}\ \textsc{Sub-H}}{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d}\ \mathsf{h} :: g : (\{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}) \to (x : \{v : \alpha \mid \kappa_\alpha\} \to \{\mathsf{h}\ x == g\ x\}) \to \{\mathsf{h} \simeq g\}}$$

$$\dfrac{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d}\ \mathsf{h} :: \ldots \qquad \dfrac{\Gamma(\mathsf{k}) \;=\; x : \{v : \alpha \mid d_{\mathsf{k}}\} \to \{v : \beta \mid r_{\mathsf{k}}\}}{\Gamma \vdash \mathsf{k} :: x : \{v : \alpha \mid d_{\mathsf{k}}\} \to \{v : \beta \mid r_{\mathsf{k}}\}} \qquad \dfrac{\ldots}{\Gamma \vdash x : \{v : \alpha \mid d_{\mathsf{k}}\} \to \{v : \beta \mid r_{\mathsf{k}}\} \;\preceq\; \{v : \alpha \mid \kappa_\alpha\} \to \{v : \beta \mid \kappa_\beta\}}\ \textsc{Sub-K}}{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d}\ \mathsf{h}\ \mathsf{k} :: (x : \{v : \alpha \mid \kappa_\alpha\} \to \{\mathsf{h}\ x == \mathsf{k}\ x\}) \to \{\mathsf{h} \simeq \mathsf{k}\}}$$

$$\dfrac{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d}\ \mathsf{h}\ \mathsf{k} :: \ldots \qquad \dfrac{\Gamma(\mathsf{lemma}) \;=\; x : \alpha \to \{p\}}{\Gamma \vdash \mathsf{lemma} :: x : \alpha \to \{p\}} \qquad \dfrac{\ldots}{\Gamma \vdash x : \alpha \to \{p\} \;\preceq\; x : \{v : \alpha \mid \kappa_\alpha\} \to \{\mathsf{h}\ x == \mathsf{k}\ x\}}\ \textsc{Sub-L}}{\Gamma \vdash \mathsf{funext}\ @\{v : \alpha \mid \kappa_\alpha\}\ @\{v : \beta \mid \kappa_\beta\}\ \mathsf{d}\ \mathsf{h}\ \mathsf{k}\ \mathsf{lemma} :: \{h \simeq k\}}$$

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

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

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

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

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

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

$$\frac{}{G \vdash_B c :: \lfloor \mathsf{TyCons}(c) \rfloor}\ \text{BT-Con} \qquad \frac{x : t \in G}{G \vdash_B x :: t}\ \text{BT-Var}$$

$$\frac{G \vdash_B e :: t_x \to t \qquad G \vdash_B e_x :: t_x}{G \vdash_B e\ e_x :: t}\ \text{BT-App} \qquad \frac{G, x : \lfloor \tau_x \rfloor \vdash_B e :: t}{G \vdash_B \lambda x{:}\tau_x.\ e :: \lfloor \tau_x \rfloor \to t}\ \text{BT-Lam}$$

$$\frac{G \vdash_B e :: () \quad G \vdash_B e_1 :: b \qquad G \vdash_B e_2 :: b}{G \vdash_B \mathsf{bEq}_b\ e_1\ e_2\ e :: \mathsf{PBEq}_b e_1 e_2}\ \text{BT-Eq-Base} \qquad \frac{G \vdash_B e :: () \quad G \vdash_B e_1 :: \lfloor \tau_x \to \tau \rfloor \qquad G \vdash_B e_2 :: \lfloor \tau_x \to \tau \rfloor}{G \vdash_B \mathsf{xEq}_{x:\tau_x \to \tau}\ e_1\ e_2\ e :: \mathsf{PBEq}_{\lfloor \tau_x \to \tau \rfloor} e_1 e_2}\ \text{BT-Eq-Fun}$$

Fig. 12. Syntax and Typing of λ^E.

B PROOFS AND DEFINITIONS FOR METATHEORY

In this section we provide proofs and definitions ommitted from §3.

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 indexes from the typed equality.

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

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

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* $c = \mathsf{true}, \mathsf{false}, \mathsf{unit}$, *and* $==_b$, *constants are sound, i.e.,* $c \in [\![\mathsf{TyCons}(c)]\!]$.

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

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

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

- $e \equiv \texttt{false}$ 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 BT-Con; 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 $e \in [\![\{x{:}() \mid x ==_{()} \texttt{unit}\}]\!]$. 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 BT-Con; 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 $e \equiv e_x ==_b e_y \in [\![(\{z{:}\texttt{Bool} \mid z ==_{\texttt{Bool}} (x ==_b y)\})[e_x/x][e_y/y]]\!]$. 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 BT-Con 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

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

□

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.

T-Sub 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 $\Gamma \models e \in \tau$.

T-Self 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, $\Gamma \models e \in \{z{:}b \mid z ==_b e\}$.

T-Con This case holds exactly because of Property B.1.

T-Var This case holds by the definition of closing substitutions.

T-Lam Assume $\Gamma \vdash \lambda x{:}\tau_x.\ e :: x{:}\tau_x \rightarrow \tau$. 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 $\Gamma \models \lambda x{:}\tau_x.\ e \in x{:}\tau_x \rightarrow \tau$. Which, for some $\theta \in [\![\Gamma]\!]$ is equivalent to $\lambda x{:}\theta \cdot \tau_x.\ \theta \cdot e \in [\![x{:}\theta \cdot \tau_x \rightarrow \theta \cdot \tau]\!]$.

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.

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

(1) $\Gamma \vdash e :: x{:}\tau_x \rightarrow \tau$
(2) $\Gamma \vdash e_x :: \tau_x$

By IH we get

(3) $\Gamma \models e \in x{:}\tau_x \rightarrow \tau$
(4) $\Gamma \models e_x \in \tau_x$

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

(5) $\theta \cdot e \in [\![\theta \cdot x{:}\tau_x \rightarrow \tau]\!]$
(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 $\Gamma \models e\ e_x \in \tau[e_x/x]$

T-Eq-Base 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) $\theta \cdot \mathsf{bEq}_b\ e_l\ e_r\ e \in \llbracket \theta \cdot \mathsf{PEq}_b\ \{e_l\}\ \{e_r\} \rrbracket$

Which leads to the required $\Gamma \models \mathsf{bEq}_b\ e_l\ e_r\ e \in \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$.

T-Eq-Fun 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) $\theta \cdot e_l, \theta \cdot e_r \in \llbracket \theta \cdot x{:}\tau_x \to \tau \rrbracket$

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

– $\forall e_x \in \llbracket \tau_x \rrbracket . e\ e_x \in \llbracket \mathsf{PEq}_{\tau[e_x/x]}\ \{e_l\ e_x\}\ \{e_r\ e_x\} \rrbracket$

By (12), (13), (14), and (15) we get

(19) $\theta \cdot \mathsf{xEq}_{x:\tau_x \to \tau}\ e_l\ e_r\ e \in \llbracket \theta \cdot \mathsf{PEq}_{x:\tau_x \to \tau}\ \{e_l\}\ \{e_r\} \rrbracket$

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 $e \in \llbracket \tau \rrbracket$, then $e \hookrightarrow^* v$ and $v \in \llbracket \tau \rrbracket$.*

Proof. By structural induction of the type τ. □

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

Proof. We can use parallel reductions (of §C) to prove that if $e_1 \rightrightarrows e_2$, then (1) $\llbracket \tau[e_1/x] \rrbracket = \llbracket \tau[e_2/x] \rrbracket$ and (2) $e_1 \in \llbracket \tau \rrbracket$ *iff* $e_2 \in \llbracket \tau \rrbracket$. 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 $\forall \theta \in \llbracket \Gamma \rrbracket . \llbracket \theta \cdot \tau \rrbracket \subseteq \llbracket \theta \cdot \tau' \rrbracket$.

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:

S-Base 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.

S-Fun Assume $\Gamma \vdash x{:}\tau_x \rightarrow \tau \preceq x{:}\tau'_x \rightarrow \tau'$. 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 $e \in [\![\theta \cdot x{:}\tau_x \rightarrow \tau]\!]$ and we will show that $e \in [\![\theta \cdot x{:}\tau'_x \rightarrow \tau']\!]$. 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 $\forall e_x \in [\![\theta \cdot \tau'_x]\!].\ e\ e_x \in [\![\theta \cdot \tau'[e_x/x]]\!]$. We fix e_x. By (3), if $e_x \in [\![\theta \cdot \tau'_x]\!]$, 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, $e \in [\![\theta \cdot x{:}\tau'_x \rightarrow \tau']\!]$. This leads to $[\![\theta \cdot x{:}\tau_x \rightarrow \tau]\!] \subseteq [\![\theta \cdot x{:}\tau'_x \rightarrow \tau']\!]$, which by definition gives semantic subtyping: $\Gamma \vdash x{:}\tau_x \rightarrow \tau \subseteq x{:}\tau'_x \rightarrow \tau'$.

S-Eq 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 $\Gamma \vdash \mathsf{PEq}_{x:\tau_x \rightarrow \tau}\ \{e_l\}\ \{e_r\} \preceq \mathsf{PEq}_{x:\tau'_x \rightarrow \tau'}\ \{e_l\}\ \{e_r\}$

By inversion

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

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) $\Gamma \vdash x{:}\tau_x \rightarrow \tau \subseteq x{:}\tau'_x \rightarrow \tau'$
(7) $\Gamma \vdash \tau'_x \subseteq \tau_x$

We fix $\theta \in \Gamma$ and some e. If $e \in [\![\theta \cdot \mathsf{PEq}_{x:\tau_x \rightarrow \tau}\ \{e_l\}\ \{e_r\}]\!]$ we need to show that $e \in [\![\theta \cdot \mathsf{PEq}_{x:\tau'_x \rightarrow \tau'}\ \{e_l\}\ \{e_r\}]\!]$. By the assumption we have

(8) $\vdash_B e :: \mathsf{PBEq}_{\lfloor \theta \cdot (x:\tau_x \rightarrow \tau) \rfloor}$
(9) $e \hookrightarrow^* \mathsf{xEq\_}\ (\theta \cdot e_l)\ (\theta \cdot e_r)\ e_{pf}$
(10) $(\theta \cdot e_l), (\theta \cdot e_r) \in [\![\theta \cdot (x{:}\tau_x \rightarrow \tau)]\!]$
(11) $\forall e_x \in [\![\theta \cdot \tau_x]\!].e_{pf}\ e_x \in [\![\mathsf{PEq}_{\theta \cdot (\tau[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}]\!]$

Since (8) only depends on the structure of the type index, we get

(12) $\vdash_B e :: \mathsf{PBEq}_{\lfloor \theta \cdot (x:\tau'_x \rightarrow \tau') \rfloor}$

By (6) and (10) we get

(13) $(\theta \cdot e_l), (\theta \cdot e_r) \in [\![\theta \cdot (x{:}\tau'_x \rightarrow \tau')]\!]$

By (4), (5), Lemma B.7, the rule S-Eq and the IH, we get that $[\![\mathsf{PEq}_{\theta \cdot (\tau[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}]\!] \subseteq [\![\mathsf{PEq}_{\theta \cdot (\tau'[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}]\!]$. By which, (11), (7), and reasoning similar to the S-Fun case, we get

(14) $\forall e_x \in [\![\theta \cdot \tau'_x]\!].e_{pf}\ e_x \in [\![\mathsf{PEq}_{\theta \cdot (\tau'[e_x/x])}\ \{(\theta \cdot e_l)\ e_x\}\ \{(\theta \cdot e_r)\ e_x\}]\!]$

By (12), (9), (13), and (14) we conclude that $e \in [\![\theta \cdot \mathsf{PEq}_{x:\tau'_x \rightarrow \tau'}\ \{e_l\}\ \{e_r\}]\!]$, thus $\Gamma \vdash \mathsf{PEq}_{x:\tau_x \rightarrow \tau}\ \{e_l\}\ \{e_r\} \subseteq \mathsf{PEq}_{x:\tau'_x \rightarrow \tau'}\ \{e_l\}\ \{e_r\}$.

□

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 T-Var case is insteresting; we use T-Sub 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* $\mathsf{TyCons}(c) = b$ *and* $\Gamma \vdash \mathsf{TyCons}(c) \preceq \{x{:}b \mid e\}$*.*
- *If* $\tau = x{:}\tau_x \to \tau'$*, then* $v =$ *T-Lam*$x\tau'_x e$ *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.

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

S-Fun 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, $\Gamma \vdash e_1 \sim e_2 :: x{:}\tau'_x \to \tau'$.

S-Eq 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). $\Gamma \vdash c \sim c :: \mathsf{TyCons}(c)$

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 $e_1 \rightrightarrows^* e_1'$, $e_2 \rightrightarrows^* e_2'$, 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 $\delta_i, [v_{x_i}/x] \cdot r[v_i/z] \rightrightarrows^* \delta_i \cdot r[v_i/z][e_i/x]$ 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 $\delta_1 \cdot e_l[e_x/x] \rightrightarrows^* \delta_1, [v_{x_1}/x] \cdot e_l$ and $\delta_2 \cdot e_r[e_x/x] \rightrightarrows^* \delta_2, [v_{x_2}/x] \cdot e_r$

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:

T-Sub 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 $\Gamma \vdash e \sim e :: \tau$.

T-Con By Lemma B.15.

T-Self 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)

T-Var By inversion of the rule and Lemma B.17.

T-Lam 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, $\Gamma \vdash \lambda x{:}\tau_x.\ e \sim \lambda x{:}\tau_x.\ e :: x{:}\tau_x \to \tau$

T-App 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, $\Gamma \vdash e\ e_x \sim e\ e_x :: \tau[e_x/x]$.

T-Eq-Base 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) $\Gamma, r : \tau_r, l : \tau_l \vdash e :: \{x{:}() \mid l ==_b r\}$

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, $\Gamma \vdash \mathsf{bEq}_b\ e_l\ e_r\ e \sim \mathsf{bEq}_b\ e_l\ e_r\ e :: \mathsf{PEq}_b\ \{e_l\}\ \{e_r\}$.

T-Eq-Fun 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) $\Gamma \vdash \tau_r \preceq x{:}\tau_x \rightarrow \tau$
(5) $\Gamma \vdash \tau_l \preceq x{:}\tau_x \rightarrow \tau$
(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 S-Base and semantic typing.
(5) We find $\Gamma \vdash e_p :: \mathsf{PEq}_b \ \{e\} \ \{e\}$ by T-Eq-Base, 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 T-Self.
(7) We find (5b) immediately by S-Base.
(8) We find (5c) by T-Var, using T-Sub to see that if $l, r : \{x{:}b \mid x ==_b e\}$ then `unit` will be typeable at the refinement where both l and r are equal to e.

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

(1) $\Gamma, x : \tau_x \vdash e\ x :: \tau[x/x]$ by T-App and T-Var, 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, $\mathrm{pf}(l, r, x{:}\tau_x \to \tau) = x{:}\tau_x \to \mathsf{PEq}_\tau \ \{e\ x\} \ \{e\ x\}$.
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 T-Eq-Base or T-Eq-Fun, respectively.
(4) Let $e_{\mathrm{pf}} = x{:}\tau_x \to e''_{\mathrm{pf}}$.
(5) Let $e_p = \mathsf{xEq}_{x:\tau_x \to \tau} \ e \ e \ e_{\mathrm{pf}}$.
(6) Let $e_l = e_r = e$ and $\tau_l = x{:}\tau_x \to \tau'_l$ and $\tau_r = x{:}\tau_x \to \tau'_r$.
(7) We find subtyping by S-Fun and (2a).
(8) By T-Eq-Fun. We must show:
  (a) $\Gamma \vdash e_l :: \tau_l$ and $\Gamma \vdash e_r :: \tau_r$;
  (b) $\Gamma \vdash \tau_l \ \preceq \ x{:}\tau_x \to \tau$ and $\Gamma \vdash \tau_r \ \preceq \ x{:}\tau_x \to \tau$;
  (c) $\Gamma, r : \tau_r, l : \tau_l \vdash e_{\mathrm{pf}} :: (x{:}\tau_x \to \mathsf{PEq}_\tau \ \{l\ x\} \ \{r\ x\})$
  (d) $\Gamma \vdash x{:}\tau_x \to \tau$
(9) We find (8a) by assumption, T-Sub, and (7).
(10) We find (8b) by (7).
(11) We find (8c) by T-Lam 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 T-Eq-Base, 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 S-Base and symmetry of $(==_b)$.

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

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

(2) By canonical forms, $v_{12} = \mathsf{xEq}_{x:\tau'_x \to \tau''}\ e_l\ e_r\ v_p$ such that $\tau_x \vdash \tau'_x \ \preceq$ and $\tau'' \vdash \tau' \ \preceq$ 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 = T-LAM$x\tau'_x e_p$ such that $\Gamma \vdash \tau_x \ \preceq \ \tau'_x$ and $\Gamma, r : \tau_r, l : \tau_l, x : \tau'_x \vdash e :: \tau'''$ such that $\Gamma, r : \tau_r, l : \tau_l, x : \tau'_x \vdash \tau''' \ \preceq \ \mathsf{PEq}_{\tau''}\ \{l\ x\}\ \{r\ x\}$.

(4) By T-SUB, (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 T-LAM, and T-SUB (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 T-EQ-BASE, 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:

$$\Gamma, r : \tau_2, l : \tau_1 \vdash \{x{:}() \mid x ==_{()} \mathtt{unit}\} \ \preceq \ \{x{:}b \mid \{x{:}() \mid l ==_b r\}\}, \text{ and}$$
$$\Gamma, r : \tau_3, l : \tau'_2 \vdash \{x{:}() \mid x ==_{()} \mathtt{unit}\} \ \preceq \ \{x{:}b \mid \{x{:}() \mid l ==_b r\}\}.$$

(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 T-EQ-BASE, 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 T-SUB and S-BASE, 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

$$\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2$$
$$\Gamma \vdash e_2 :: \tau_2' \quad \Gamma \vdash e_3 :: \tau_3$$

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 T-Sub, 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 T-Lam.
(8) Let $v_{13} = \mathsf{xEq}_{x:\tau_x \to \tau'}\ e_1\ e_3\ v_p$.
(9) By T-Eq-Base, 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 4) and a binary logical relation characterizing equivalence (Figure 6). 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 $e_1' \rightrightarrows\!\!\!\!\rightsquigarrow e_2$ (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 $\tau \Rrightarrow \tau'$ and $e \Rrightarrow e'$ and $v \Rrightarrow v'$. Let $e_2' = e'[v'/x]$. We find $e_2 \hookrightarrow^* e_2'$ in one step by β. We find $e[v/x] \Rrightarrow e'[v'/x]$ by substitutivity of parallel reduction (Lemma C.2).

In the β case, we have $e_2 = e'[v'/x]$ such that $e \Rrightarrow e'$ and $v \Rrightarrow v'$. Let $e_2' = e_2$. We find $e_2 \hookrightarrow^* e_2'$ in no steps at all; we find $e_1' \Rrightarrow e_2'$ 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 $(==_b)\ c_1 \Rrightarrow e_2$: 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 $(==_{(c_1,b)})\ c_2 \Rrightarrow e_2$: 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). □

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

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

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.[7] That is, $\overset{\rightsquigarrow}{\rightsquigarrow} \subseteq \rightrightarrows$, 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* $e_1 \overset{\rightsquigarrow}{\rightsquigarrow} e_2$ *then* $e_1 \rightrightarrows e_2$.

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

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

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 $y \overset{\rightsquigarrow}{\rightsquigarrow} y$ by assumption (or var). If $x = y$, then $e_1[e_2/x] = e_2$ and we have $e_2 \overset{\rightsquigarrow}{\rightsquigarrow} e_2'$ 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 $\lambda y{:}\tau[e_2/x].\ e[e_2/x] \overset{\rightsquigarrow}{\rightsquigarrow} \lambda y{:}\tau'[e_2'/x].\ e'[e_2'/x]$, which we have immediately by lam and Lemma C.2 on our two subparts. If $y = x$, then we must show $\lambda y{:}\tau[e_2/x].\ e \overset{\rightsquigarrow}{\rightsquigarrow} \lambda y{:}\tau'[e_2'/x].\ e'$, 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. □

[7] 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 $v_1 \leftrightsquigarrow v_2$.*

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 $e_1' \leftrightsquigarrow e_2$.*

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 $e' \leftrightsquigarrow e''$. We *ignore* the IH on $v \rightrightarrows v''$, noticing instead that parallel reducing values are congruent (Lemma C.10) and so $v \leftrightsquigarrow v''$. 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 $e'[v/x] \leftrightsquigarrow e''[v''/x]$, 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 $e \leftrightsquigarrow v'$ then $e \hookrightarrow^* v$ such that $v \leftrightsquigarrow v'$.*

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 $e_p' \leftrightsquigarrow v_p'$. By the IH on v_p', we know that $e_p' \hookrightarrow^* v_p$ such that $v_p \leftrightsquigarrow v_p'$.

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 $\mathsf{bEq}_b\ e_l\ e_r\ v_p \leftrightsquigarrow \mathsf{bEq}_b\ e_l'\ e_r'\ v_p'$ 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 $e_p' \leftrightsquigarrow v_p'$. By the IH on v_p', we know that $e_p' \hookrightarrow^* v_p$ such that $v_p \leftrightsquigarrow v_p'$. 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 $\mathsf{xEq}_{\tau_x:\tau\to}\ e_l\ e_r\ v_p \leftrightsquigarrow \mathsf{xEq}_{x:\tau_x'\to\tau'}\ e_l'\ e_r'\ v_p'$ 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 $v \leftrightsquigarrow v'$.*

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

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

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 $v_1 \leftrightsquigarrow v_1'$ 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 $v_{11}' \leftrightsquigarrow \lambda x{:}\tau'.\ e'$, i.e., $v_{11} = \lambda x{:}\tau.\ e$ such that $\tau \rightrightarrows \tau'$ and $e \rightrightarrows e'$. Similarly, $e_{12} \hookrightarrow^* v$ such that $v \leftrightsquigarrow v'$.
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 $e[v/x] \leftrightsquigarrow e'[v'/x]$.

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 $v_{11} \leftrightsquigarrow (==_b)$ (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 $e_1' \leftrightsquigarrow e_2$. But congruence is a backward simulation (Lemma C.14), so $e_1' \hookrightarrow^* e_1''$ such that $e_1'' \leftrightsquigarrow e_2'$. 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. □