

Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

March 8, 2016

1 Preliminaries

```
module Prelims where

postulate Level : Set
postulate zro : Level
postulate suc : Level → Level
{-# BUILTIN LEVEL Level #-}
{-# BUILTIN LEVELZERO zro #-}
{-# BUILTIN LEVELSUC suc #-}
```

1.1 The Empty Type

```
data False : Set where
```

1.2 Conjunction

```
data _^_ {i} (P Q : Set i) : Set i where
  _,_ : P → Q → P ^ Q

π1 : ∀ {i} {P Q : Set i} → P ^ Q → P
π1 (x , _) = x

π2 : ∀ {i} {P Q : Set i} → P ^ Q → Q
π2 (_, y) = y
```

1.3 Functions

We write id_A for the identity function on the type A , and $g \circ f$ for the composition of functions g and f .

```
id : ∀ (A : Set) → A → A
id A x = x
```

```

infix 75 _o_
_o_ : ∀ {A B C : Set} → (B → C) → (A → B) → A → C
(g o f) x = g (f x)

```

1.4 Equality

We use the inductively defined equality $=$ on every datatype.

```

infix 50 _≡_
data _≡_ {A : Set} (a : A) : A → Set where
  ref : a ≡ a

```

```

subst : ∀ {i} {A : Set} (P : A → Set i) {a} {b} → a ≡ b → P a → P b
subst P ref Pa = Pa

```

```

subst2 : ∀ {A B : Set} (P : A → B → Set) {a a' b b'} → a ≡ a' → b ≡ b' → P a b → P a' b'
subst2 P ref ref Pab = Pab

```

```

sym : ∀ {A : Set} {a b : A} → a ≡ b → b ≡ a
sym ref = ref

```

```

trans : ∀ {A : Set} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans ref ref = ref

```

```

wd : ∀ {A B : Set} (f : A → B) {a a' : A} → a ≡ a' → f a ≡ f a'
wd _ ref = ref

```

```

wd2 : ∀ {A B C : Set} (f : A → B → C) {a a' : A} {b b' : B} → a ≡ a' → b ≡ b' → f a b ≡ f a' b'
wd2 _ ref ref = ref

```

```

module Equational-Reasoning (A : Set) where
  infix 2 `'_
  `'_ : ∀ (a : A) → a ≡ a
  `'_ _ = ref

  infix 1 _≡_[_]
  _≡_[_] : ∀ {a b : A} → a ≡ b → ∀ c → b ≡ c → a ≡ c
  δ ≡ c [ δ' ] = trans δ δ'

  infix 1 _≡_[[_]]
  _≡_[[_]] : ∀ {a b : A} → a ≡ b → ∀ c → c ≡ b → a ≡ c
  δ ≡ c [[ δ' ]] = trans δ (sym δ')

```

We also write $f \sim g$ iff the functions f and g are extensionally equal, that is, $f(x) = g(x)$ for all x .

```

infix 50 _~_
_~_ : ∀ {A B : Set} → (A → B) → (A → B) → Set
f ~ g = ∀ x → f x ≡ g x

```

2 Datatypes

We introduce a universe **FinSet** of (names of) finite sets. There is an empty set $\emptyset : \mathbf{FinSet}$, and for every $A : \mathbf{FinSet}$, the type $A + 1 : \mathbf{FinSet}$ has one more element:

$$A + 1 = \{\perp\} \uplus \{\uparrow a : a \in A\}$$

```

data FinSet : Set where
  ∅ : FinSet
  Lift : FinSet → FinSet

data El : FinSet → Set where
  ⊥ : ∀ {V} → El (Lift V)
  ↑ : ∀ {V} → El V → El (Lift V)

```

3 Grammars

```

module Grammar where

open import Prelims hiding (_~_)

```

Before we begin investigating the several theories we wish to consider, we present a general theory of syntax and capture-avoiding substitution.

A *grammar* consists of:

- a set of *expression kinds*;
- a set of *constructors*, each with an associated *constructor kind* of the form

$$((A_{11}, \dots, A_{1r_1})B_1, \dots, (A_{m1}, \dots, A_{mr_m})B_m)C \quad (1)$$

where each A_{ij} , B_i and C is an expression kind.

- a binary relation of *parenthood* on the set of expression kinds.

A constructor c of kind (1) is a constructor that takes m arguments of kind B_1, \dots, B_m , and binds r_i variables in its i th argument of kind A_{ij} , producing an expression of kind C . We write this expression as

$$c([x_{11}, \dots, x_{1r_1}]E_1, \dots, [x_{m1}, \dots, x_{mr_m}]E_m) . \quad (2)$$

The subexpressions of the form $[x_{i1}, \dots, x_{ir_i}]E_i$ shall be called *abstractions*, and the pieces of syntax of the form $(A_{i1}, \dots, A_{ij})B_i$ that occur in constructor kinds shall be called *abstraction kinds*.

```

mutual
  data KindClass (ExpressionKind : Set) : Set where
    -Expression : KindClass ExpressionKind
    -Abstraction : KindClass ExpressionKind
    -Constructor : ExpressionKind → KindClass ExpressionKind

  data Kind (ExpressionKind : Set) : KindClass ExpressionKind → Set where
    base : ExpressionKind → Kind ExpressionKind -Expression
    out : ExpressionKind → Kind ExpressionKind -Abstraction
     $\Pi$  : ExpressionKind → Kind ExpressionKind -Abstraction → Kind ExpressionKind -Abstraction
    out2 :  $\forall \{K\} \rightarrow$  Kind ExpressionKind (-Constructor K)
     $\Pi_2$  :  $\forall \{K\} \rightarrow$  Kind ExpressionKind -Abstraction → Kind ExpressionKind (-Constructor K)

  AbstractionKind : Set → Set
  AbstractionKind ExpressionKind = Kind ExpressionKind -Abstraction

  ConstructorKind :  $\forall \{ExpressionKind\} \rightarrow$  ExpressionKind → Set
  ConstructorKind {ExpressionKind} K = Kind ExpressionKind (-Constructor K)

  record Taxonomy : Set1 where
    field
      VarKind : Set
      NonVarKind : Set

  data ExpressionKind : Set where
    varKind : VarKind → ExpressionKind
    nonVarKind : NonVarKind → ExpressionKind

  record ToGrammar (T : Taxonomy) : Set1 where
    open Taxonomy T
    field
      Constructor :  $\forall \{K : ExpressionKind\} \rightarrow$  ConstructorKind K → Set
      parent : VarKind → ExpressionKind

  An alphabet  $V = \{V_E\}_E$  consists of a set  $V_E$  of variables of kind  $E$  for each expression kind  $E$ . The expressions of kind  $E$  over the alphabet  $V$  are defined inductively by:
    

- Every variable of kind  $E$  is an expression of kind  $E$ .
- If  $c$  is a constructor of kind (1), each  $E_i$  is an expression of kind  $B_i$ , and each  $x_{ij}$  is a variable of kind  $A_{ij}$ , then (2) is an expression of kind  $C$ .


  Each  $x_{ij}$  is bound within  $E_i$  in the expression (2). We identify expressions up to  $\alpha$ -conversion.

  data Alphabet : Set where
     $\emptyset$  : Alphabet

```

```

_,_ : Alphabet → VarKind → Alphabet

data Var : Alphabet → VarKind → Set where
  x0 : ∀ {V} {K} → Var (V , K) K
  ↑ : ∀ {V} {K} {L} → Var V L → Var (V , K) L

data Expression' (V : Alphabet) : ∀ C → Kind ExpressionKind C → Set where
  var : ∀ {K} → Var V K → Expression' V -Expression (base (varKind K))
  app : ∀ {K} {C : ConstructorKind K} → Constructor C → Expression' V (-Constructor K)
  out : ∀ {K} → Expression' V -Expression (base K) → Expression' V -Abstraction (out K)
  Λ : ∀ {K} {A} → Expression' (V , K) -Abstraction A → Expression' V -Abstraction (out K)
  out2 : ∀ {K} → Expression' V (-Constructor K) out2
  app2 : ∀ {K} {A} {C} → Expression' V -Abstraction A → Expression' V (-Constructor K)

Expression'' : Alphabet → ExpressionKind → Set
Expression'' V K = Expression' V -Expression (base K)

Body' : Alphabet → ∀ K → ConstructorKind K → Set
Body' V K C = Expression' V (-Constructor K) C

Abstraction' : Alphabet → AbstractionKind ExpressionKind → Set
Abstraction' V K = Expression' V -Abstraction K

```

Given alphabets U , V , and a function ρ that maps every variable in U of kind K to a variable in V of kind K , we denote by $E\{\rho\}$ the result of *replacing* every variable x in E with $\rho(x)$.

```

Rep : Alphabet → Alphabet → Set
Rep U V = ∀ K → Var U K → Var V K

_~R_ : ∀ {U} {V} → Rep U V → Rep U V → Set
ρ ~R ρ' = ∀ {K} x → ρ K x ≡ ρ' K x

```

The alphabets and replacements form a category.

```

idRep : ∀ V → Rep V V
idRep _ _ x = x

infixl 75 _•R_
_•R_ : ∀ {U} {V} {W} → Rep V W → Rep U V → Rep U W
(ρ' •R ρ) K x = ρ' K (ρ K x)

```

--We choose not to prove the category axioms, as they hold up to judgemental equality.

Given a replacement $\rho : U \rightarrow V$, we can ‘lift’ this to a replacement $(\rho, K) : (U, K) \rightarrow (V, K)$. Under this operation, the mapping $(-, K)$ becomes an endofunctor on the category of alphabets and replacements.

$\text{Rep}\uparrow : \forall \{U\} \{V\} \{K\} \rightarrow \text{Rep } U \ V \rightarrow \text{Rep } (U, K) \ (V, K)$
 $\text{Rep}\uparrow _ _ x_0 = x_0$
 $\text{Rep}\uparrow \rho \ K \ (\uparrow x) = \uparrow (\rho \ K \ x)$

$\text{Rep}\uparrow\text{-wd} : \forall \{U\} \{V\} \{K\} \{\rho \ \rho' : \text{Rep } U \ V\} \rightarrow \rho \sim_R \rho' \rightarrow \text{Rep}\uparrow \{K = K\} \rho \sim_R \text{Rep}\uparrow \rho'$
 $\text{Rep}\uparrow\text{-wd } \rho\text{-is-}\rho' \ x_0 = \text{ref}$
 $\text{Rep}\uparrow\text{-wd } \rho\text{-is-}\rho' \ (\uparrow x) = \text{wd } \uparrow (\rho\text{-is-}\rho' \ x)$

$\text{Rep}\uparrow\text{-id} : \forall \{V\} \{K\} \rightarrow \text{Rep}\uparrow (\text{idRep } V) \sim_R \text{idRep } (V, K)$
 $\text{Rep}\uparrow\text{-id } x_0 = \text{ref}$
 $\text{Rep}\uparrow\text{-id } (\uparrow _) = \text{ref}$

$\text{Rep}\uparrow\text{-comp} : \forall \{U\} \{V\} \{W\} \{K\} \{\rho' : \text{Rep } V \ W\} \{\rho : \text{Rep } U \ V\} \rightarrow \text{Rep}\uparrow \{K = K\} (\rho' \bullet_R \rho) \sim$
 $\text{Rep}\uparrow\text{-comp } x_0 = \text{ref}$
 $\text{Rep}\uparrow\text{-comp } (\uparrow _) = \text{ref}$

Finally, we can define $E\langle\rho\rangle$, the result of replacing each variable x in E with $\rho(x)$. Under this operation, the mapping $\text{Expression} \rightarrow K$ becomes a functor from the category of alphabets and replacements to the category of sets.

$\text{rep} : \forall \{U\} \{V\} \{C\} \{K\} \rightarrow \text{Expression}' \ U \ C \ K \rightarrow \text{Rep } U \ V \rightarrow \text{Expression}' \ V \ C \ K$
 $\text{rep } (\text{var } x) \ \rho = \text{var } (\rho \ x)$
 $\text{rep } (\text{app } c \ EE) \ \rho = \text{app } c \ (\text{rep } EE \ \rho)$
 $\text{rep } (\text{out } E) \ \rho = \text{out } (\text{rep } E \ \rho)$
 $\text{rep } (\Lambda E) \ \rho = \Lambda (\text{rep } E \ (\text{Rep}\uparrow \ \rho))$
 $\text{rep } \text{out}_2 _ = \text{out}_2$
 $\text{rep } (\text{app}_2 \ E \ F) \ \rho = \text{app}_2 \ (\text{rep } E \ \rho) \ (\text{rep } F \ \rho)$

mutual

$\text{infix } 60 \ _ \langle _ \rangle$
 $_ \langle _ \rangle : \forall \{U\} \{V\} \{K\} \rightarrow \text{Expression}' \ U \ K \rightarrow \text{Rep } U \ V \rightarrow \text{Expression}' \ V \ K$
 $\text{var } x \ \langle \ \rho \ \rangle = \text{var } (\rho \ x)$
 $(\text{app } c \ EE) \ \langle \ \rho \ \rangle = \text{app } c \ (EE \ \langle \ \rho \ \rangle B)$

$\text{infix } 60 \ _ \langle _ \rangle B$
 $_ \langle _ \rangle B : \forall \{U\} \{V\} \{K\} \{C : \text{ConstructorKind } K\} \rightarrow \text{Expression}' \ U \ (-\text{Constructor } K) \ C \rightarrow$
 $\text{out}_2 \ \langle \ \rho \ \rangle B = \text{out}_2$
 $(\text{app}_2 \ A \ EE) \ \langle \ \rho \ \rangle B = \text{app}_2 \ (A \ \langle \ \rho \ \rangle A) \ (EE \ \langle \ \rho \ \rangle B)$

$\text{infix } 60 \ _ \langle _ \rangle A$
 $_ \langle _ \rangle A : \forall \{U\} \{V\} \{A\} \rightarrow \text{Expression}' \ U \ -\text{Abstraction } A \rightarrow \text{Rep } U \ V \rightarrow \text{Expression}' \ V \ -\text{Ab}$
 $\text{out } E \ \langle \ \rho \ \rangle A = \text{out } (E \ \langle \ \rho \ \rangle)$
 $\Lambda \ A \ \langle \ \rho \ \rangle A = \Lambda \ (A \ \langle \ \text{Rep}\uparrow \ \rho \ \rangle A)$

mutual

$\text{rep-wd} : \forall \{U\} \{V\} \{K\} \{E : \text{Expression}' \ U \ K\} \{\rho : \text{Rep } U \ V\} \{\rho'\} \rightarrow \rho \sim_R \rho' \rightarrow \text{rep } E$

```

rep-wd {E = var x} ρ-is-ρ' = wd var (ρ-is-ρ' x)
rep-wd {E = app c EE} ρ-is-ρ' = wd (app c) (rep-wdB ρ-is-ρ')

rep-wdB : ∀ {U} {V} {K} {C : ConstructorKind K} {EE : Expression' U (-Constructor K)}
rep-wdB {U} {V} .{K} {out2 {K}} {out2} ρ-is-ρ' = ref
rep-wdB {U} {V} {K} {Π2 A C} {app2 A' EE} ρ-is-ρ' = wd2 app2 (rep-wdA ρ-is-ρ') (rep-wdB ρ-is-ρ')

rep-wdA : ∀ {U} {V} {A} {E : Expression' U -Abstraction A} {ρ ρ' : Rep U V} → ρ ~R ρ'
rep-wdA {U} {V} {out K} {out E} ρ-is-ρ' = wd out (rep-wd ρ-is-ρ')
rep-wdA {U} {V} .{Π (varKind _) _} {Λ E} ρ-is-ρ' = wd Λ (rep-wdA (Rep↑-wd ρ-is-ρ'))

mutual
rep-id : ∀ {V} {K} {E : Expression'' V K} → rep E (idRep V) ≡ E
rep-id {E = var _} = ref
rep-id {E = app c _} = wd (app c) rep-idB

rep-idB : ∀ {V} {K} {C : ConstructorKind K} {EE : Expression' V (-Constructor K) C}
rep-idB {EE = out2} = ref
rep-idB {EE = app2 _ _} = wd2 app2 rep-idA rep-idB

rep-idA : ∀ {V} {K} {A : Expression' V -Abstraction K} → rep A (idRep V) ≡ A
rep-idA {A = out _} = wd out rep-id
rep-idA {A = Λ _} = wd Λ (trans (rep-wdA Rep↑-id) rep-idA)

mutual
rep-comp : ∀ {U} {V} {W} {K} {ρ : Rep U V} {ρ' : Rep V W} {E : Expression'' U K} → rep E (ρ ~R ρ')
rep-comp {E = var _} = ref
rep-comp {E = app c _} = wd (app c) rep-compB

rep-compB : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {ρ : Rep U V} {ρ' : Rep V W} {EE : Expression' V (-Constructor K) C}
rep-compB {EE = out2} = ref
rep-compB {U} {V} {W} {K} {Π2 L C} {ρ} {ρ'} {app2 A EE} = wd2 app2 rep-compA rep-compB

rep-compA : ∀ {U} {V} {W} {K} {ρ : Rep U V} {ρ' : Rep V W} {A : Expression' U -Abstraction K}
rep-compA {A = out _} = wd out rep-comp
rep-compA {U} {V} {W} .{Π (varKind K) L} {ρ} {ρ'} {Λ {K} {L} A} = wd Λ (trans (rep-wdA Rep↑-wd ρ-is-ρ') rep-compA)

```

This provides us with the canonical mapping from an expression over V to an expression over (V, K) :

```

lift : ∀ {V} {K} {L} → Expression'' V L → Expression'' (V , K) L
lift E = rep E (λ _ → ↑)

```

A *substitution* σ from alphabet U to alphabet V , $\sigma : U \Rightarrow V$, is a function σ that maps every variable x of kind K in U to an *expression* $\sigma(x)$ of kind K over V . Then, given an expression E of kind K over U , we write $E[\sigma]$ for the result of substituting $\sigma(x)$ for x for each variable in E , avoiding capture.

$\text{Sub} : \text{Alphabet} \rightarrow \text{Alphabet} \rightarrow \text{Set}$
 $\text{Sub } U \ V = \forall K \rightarrow \text{Var } U \ K \rightarrow \text{Expression}, V \ (\text{varKind } K)$

$\sim : \forall \{U\} \{V\} \rightarrow \text{Sub } U \ V \rightarrow \text{Sub } U \ V \rightarrow \text{Set}$
 $\sigma \sim \tau = \forall K \ x \rightarrow \sigma \ K \ x \equiv \tau \ K \ x$

The *identity* substitution $\text{id}_V : V \rightarrow V$ is defined as follows.

$\text{idSub} : \forall \{V\} \rightarrow \text{Sub } V \ V$
 $\text{idSub } _ \ x = \text{var } x$

Given $\sigma : U \rightarrow V$ and an expression E over U , we want to define the expression $E[\sigma]$ over V , the result of applying the substitution σ to M . Only after this will we be able to define the composition of two substitutions. However, there is some work we need to do before we are able to do this.

We can define the composition of a substitution and a replacement as follows

$\text{infix } 75 \ \bullet_1$
 $\bullet_1 : \forall \{U\} \{V\} \{W\} \rightarrow \text{Rep } V \ W \rightarrow \text{Sub } U \ V \rightarrow \text{Sub } U \ W$
 $(\rho \bullet_1 \sigma) \ K \ x = \text{rep } (\sigma \ K \ x) \ \rho$

$\text{infix } 75 \ \bullet_2$
 $\bullet_2 : \forall \{U\} \{V\} \{W\} \rightarrow \text{Sub } V \ W \rightarrow \text{Rep } U \ V \rightarrow \text{Sub } U \ W$
 $(\sigma \bullet_2 \rho) \ K \ x = \sigma \ K \ (\rho \ K \ x)$

Given a substitution $\sigma : U \Rightarrow V$, define a substitution $(\sigma, K) : (U, K) \Rightarrow (V, K)$ as follows.

$\text{Sub}\uparrow : \forall \{U\} \{V\} \{K\} \rightarrow \text{Sub } U \ V \rightarrow \text{Sub } (U, K) \ (V, K)$
 $\text{Sub}\uparrow _ _ \ x_0 = \text{var } x_0$
 $\text{Sub}\uparrow \sigma \ K \ (\uparrow x) = \text{lift } (\sigma \ K \ x)$

$\text{Sub}\uparrow\text{-wd} : \forall \{U\} \{V\} \{K\} \{\sigma \ \sigma' : \text{Sub } U \ V\} \rightarrow \sigma \sim \sigma' \rightarrow \text{Sub}\uparrow \{K = K\} \sigma \sim \text{Sub}\uparrow \sigma'$
 $\text{Sub}\uparrow\text{-wd } \{K = K\} \sigma\text{-is-}\sigma' _ _ \ x_0 = \text{ref}$
 $\text{Sub}\uparrow\text{-wd } \sigma\text{-is-}\sigma' \ L \ (\uparrow x) = \text{wd lift } (\sigma\text{-is-}\sigma' \ L \ x)$

Lemma 1. *The operations we have defined satisfy the following properties.*

1. $(\text{id}_V, K) = \text{id}_{(V, K)}$
2. $(\rho \bullet_1 \sigma, K) = (\rho, K) \bullet_1 (\sigma, K)$
3. $(\sigma \bullet_2 \rho, K) = (\sigma, K) \bullet_2 (\rho, K)$

$\text{Sub}\uparrow\text{-id} : \forall \{V\} \{K\} \rightarrow \text{Sub}\uparrow \{V\} \{V\} \{K\} \ \text{idSub} \sim \text{idSub}$
 $\text{Sub}\uparrow\text{-id } \{K = K\} _ _ \ x_0 = \text{ref}$
 $\text{Sub}\uparrow\text{-id } _ \ (\uparrow _) = \text{ref}$

$\text{Sub}\uparrow\text{-comp}_1 : \forall \{U\} \{V\} \{W\} \{K\} \{\rho : \text{Rep } V \ W\} \{\sigma : \text{Sub } U \ V\} \rightarrow \text{Sub}\uparrow (\rho \bullet_1 \sigma) \sim \text{Rep}\uparrow \rho \bullet_1 \sigma$


```

Sub↑-comp1 {K = K} . _ x0 = ref
Sub↑-comp1 {U} {V} {W} {K} {ρ} {σ} L (↑ x) = let open Equational-Reasoning (Expression)
  ∴ lift (rep (σ L x) ρ)
  ≡ rep (σ L x) (λ _ x → ↑ (ρ _ x)) [[ rep-comp {E = σ L x} ]]
  ≡ rep (lift (σ L x)) (Rep↑ ρ) [ rep-comp ]

Sub↑-comp2 : ∀ {U} {V} {W} {K} {σ : Sub V W} {ρ : Rep U V} → Sub↑ {K = K} (σ •2 ρ) ~
Sub↑-comp2 {K = K} . _ x0 = ref
Sub↑-comp2 L (↑ x) = ref

```

We can now define the result of applying a substitution σ to an expression E , which we denote $E[\sigma]$.

```

mutual
  infix 60 _[_]
  _[_] : ∀ {U} {V} {K} → Expression' U K → Sub U V → Expression' V K
  (var x) [_ σ] = σ _ x
  (app c EE) [_ σ] = app c (EE [_ σ]B)

  infix 60 _[_]B
  _[_]B : ∀ {U} {V} {K} {C : ConstructorKind K} → Expression' U (-Constructor K) C →
  out2 [_ σ]B = out2
  (app2 A EE) [_ σ]B = app2 (A [_ σ]A) (EE [_ σ]B)

  infix 60 _[_]A
  _[_]A : ∀ {U} {V} {A} → Expression' U -Abstraction A → Sub U V → Expression' V -Ab
  (out E) [_ σ]A = out (E [_ σ])
  (Λ A) [_ σ]A = Λ (A [_ Sub↑ σ]A)

mutual
  sub-wd : ∀ {U} {V} {K} {E : Expression' U K} {σ σ' : Sub U V} → σ ~ σ' → E [_ σ] ~
  sub-wd {E = var x} σ-is-σ' = σ-is-σ' _ x
  sub-wd {U} {V} {K} {app c EE} σ-is-σ' = wd (app c) (sub-wdB σ-is-σ')

  sub-wdB : ∀ {U} {V} {K} {C : ConstructorKind K} {EE : Expression' U (-Constructor K)
  sub-wdB {EE = out2} σ-is-σ' = ref
  sub-wdB {EE = app2 A EE} σ-is-σ' = wd2 app2 (sub-wdA σ-is-σ') (sub-wdB σ-is-σ')

  sub-wdA : ∀ {U} {V} {K} {A : Expression' U -Abstraction K} {σ σ' : Sub U V} → σ ~ σ'
  sub-wdA {A = out E} σ-is-σ' = wd out (sub-wd {E = E} σ-is-σ')
  sub-wdA {U} {V} .{Π (varKind K) L} {Λ {K} {L} A} σ-is-σ' = wd Λ (sub-wdA (Sub↑-wd σ-is-σ'))

```

Lemma 2.

1. $M[\text{id}_V] \equiv M$
2. $M[\rho \bullet_1 \sigma] \equiv M[\sigma]\langle \rho \rangle$

3. $M[\sigma \bullet_2 \rho] \equiv M\langle \rho \rangle[\sigma]$

mutual

subid : $\forall \{V\} \{K\} \{E : \text{Expression}' V K\} \rightarrow E \llbracket \text{idSub} \rrbracket \equiv E$
subid $\{E = \text{var } _ \} = \text{ref}$
subid $\{V\} \{K\} \{\text{app } c _ \} = \text{wd } (\text{app } c) \text{ subidB}$

subidB : $\forall \{V\} \{K\} \{C : \text{ConstructorKind } K\} \{EE : \text{Expression}' V (-\text{Constructor } K) C\} \rightarrow EE \llbracket \text{idSub} \rrbracket \equiv EE$
subidB $\{EE = \text{out}_2 \} = \text{ref}$
subidB $\{EE = \text{app}_2 _ _ \} = \text{wd}_2 \text{ app}_2 \text{ subidA subidB}$

subidA : $\forall \{V\} \{K\} \{A : \text{Expression}' V -\text{Abstraction } K\} \rightarrow A \llbracket \text{idSub} \rrbracket A \equiv A$
subidA $\{A = \text{out } _ \} = \text{wd out subid}$
subidA $\{A = \Lambda _ \} = \text{wd } \Lambda (\text{trans } (\text{sub-wdA Sub}\uparrow\text{-id}) \text{ subidA})$

mutual

sub-comp₁ : $\forall \{U\} \{V\} \{W\} \{K\} \{E : \text{Expression}' U K\} \{\rho : \text{Rep } V W\} \{\sigma : \text{Sub } U V\} \rightarrow E \llbracket \rho \bullet_1 \sigma \rrbracket \equiv \text{rep } (E \llbracket \sigma \rrbracket) \rho$
sub-comp₁ $\{E = \text{var } _ \} = \text{ref}$
sub-comp₁ $\{E = \text{app } c _ \} = \text{wd } (\text{app } c) \text{ sub-comp}_1B$

sub-comp₁B : $\forall \{U\} \{V\} \{W\} \{K\} \{C : \text{ConstructorKind } K\} \{EE : \text{Expression}' U (-\text{Constructor } K) C\} \rightarrow EE \llbracket \rho \bullet_1 \sigma \rrbracket B \equiv \text{rep } (EE \llbracket \sigma \rrbracket B) \rho$
sub-comp₁B $\{EE = \text{out}_2 \} = \text{ref}$
sub-comp₁B $\{U\} \{V\} \{W\} \{K\} \{(\Pi_2 L C)\} \{\text{app}_2 A EE\} = \text{wd}_2 \text{ app}_2 \text{ sub-comp}_1A \text{ sub-comp}_1B$

sub-comp₁A : $\forall \{U\} \{V\} \{W\} \{K\} \{A : \text{Expression}' U -\text{Abstraction } K\} \{\rho : \text{Rep } V W\} \{\sigma : \text{Sub } U V\} \rightarrow A \llbracket \rho \bullet_1 \sigma \rrbracket A \equiv \text{rep } (A \llbracket \sigma \rrbracket A) \rho$
sub-comp₁A $\{A = \text{out } E\} = \text{wd out } (\text{sub-comp}_1 \{E = E\})$
sub-comp₁A $\{U\} \{V\} \{W\} \{(\Pi (\text{varKind } K) L)\} \{\Lambda \{K\} \{L\} A\} = \text{wd } \Lambda (\text{trans } (\text{sub-wdA Sub}\uparrow\text{-id}) \text{ sub-comp}_1A)$

mutual

sub-comp₂ : $\forall \{U\} \{V\} \{W\} \{K\} \{E : \text{Expression}' U K\} \{\sigma : \text{Sub } V W\} \{\rho : \text{Rep } U V\} \rightarrow E \llbracket \sigma \bullet_2 \rho \rrbracket \equiv \text{rep } (E \llbracket \sigma \rrbracket) \rho$
sub-comp₂ $\{E = \text{var } _ \} = \text{ref}$
sub-comp₂ $\{U\} \{V\} \{W\} \{K\} \{\text{app } c EE\} = \text{wd } (\text{app } c) \text{ sub-comp}_2B$

sub-comp₂B : $\forall \{U\} \{V\} \{W\} \{K\} \{C : \text{ConstructorKind } K\} \{EE : \text{Expression}' U (-\text{Constructor } K) C\} \rightarrow EE \llbracket \sigma \bullet_2 \rho \rrbracket B \equiv (\text{rep } EE \rho) \llbracket \sigma \rrbracket B$
sub-comp₂B $\{EE = \text{out}_2 \} = \text{ref}$
sub-comp₂B $\{U\} \{V\} \{W\} \{K\} \{(\Pi_2 L C)\} \{\text{app}_2 A EE\} = \text{wd}_2 \text{ app}_2 \text{ sub-comp}_2A \text{ sub-comp}_2B$

sub-comp₂A : $\forall \{U\} \{V\} \{W\} \{K\} \{A : \text{Expression}' U -\text{Abstraction } K\} \{\sigma : \text{Sub } V W\} \{\rho : \text{Rep } U V\} \rightarrow A \llbracket \sigma \bullet_2 \rho \rrbracket A \equiv \text{rep } (A \llbracket \sigma \rrbracket A) \rho$
sub-comp₂A $\{A = \text{out } E\} = \text{wd out } (\text{sub-comp}_2 \{E = E\})$
sub-comp₂A $\{U\} \{V\} \{W\} \{(\Pi (\text{varKind } K) L)\} \{\Lambda \{K\} \{L\} A\} = \text{wd } \Lambda (\text{trans } (\text{sub-wdA Sub}\uparrow\text{-id}) \text{ sub-comp}_2A)$

We define the composition of two substitutions, as follows.

```

infix 75 _•_
_•_ : ∀ {U} {V} {W} → Sub V W → Sub U V → Sub U W
(σ • ρ) K x = ρ K x [[ σ ]]

```

Lemma 3. *Let $\sigma : V \Rightarrow W$ and $\rho : U \Rightarrow V$.*

$$1. (\sigma \bullet \rho, K) \sim (\sigma, K) \bullet (\rho, K)$$

$$2. E[\sigma \bullet \rho] \equiv E[\rho][\sigma]$$

```

Sub↑-comp : ∀ {U} {V} {W} {ρ : Sub U V} {σ : Sub V W} {K} →
  Sub↑ {K = K} (σ • ρ) ~ Sub↑ σ • Sub↑ ρ
Sub↑-comp _ x0 = ref
Sub↑-comp {W = W} {ρ = ρ} {σ = σ} {K = K} L (↑ x) =
  let open Equational-Reasoning (Expression'' (W , K) (varKind L)) in
    ∴ lift ((ρ L x) [[ σ ]])
    ≡ ρ L x [[ (λ _ → ↑) •1 σ ]] [[ sub-comp1 {E = ρ L x} ]]
    ≡ (lift (ρ L x)) [[ Sub↑ σ ]] [ sub-comp2 {E = ρ L x} ]

```

mutual

```

sub-compA : ∀ {U} {V} {W} {K} {A : Expression' U -Abstraction K} {σ : Sub V W} {ρ : Sub U V}
  A [[ σ • ρ ]] A ≡ A [[ ρ ]] A [[ σ ]] A
sub-compA {A = out E} = wd out (sub-comp {E = E})
sub-compA {U} {V} {W} .{Π (varKind K) L} {Λ {K} {L} A} {σ} {ρ} = wd Λ (let open Equational-Reasoning in
  ∴ A [[ Sub↑ (σ • ρ) ]] A
  ≡ A [[ Sub↑ σ • Sub↑ ρ ]] A [ sub-wdA Sub↑-comp ]
  ≡ A [[ Sub↑ ρ ]] A [[ Sub↑ σ ]] A [ sub-compA ])

```

```

sub-compB : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {EE : Expression' U (-ConstructorKind K)}
  EE [[ σ • ρ ]] B ≡ EE [[ ρ ]] B [[ σ ]] B
sub-compB {EE = out2} = ref
sub-compB {U} {V} {W} {K} {(Π2 L C)} {app2 A EE} = wd2 app2 sub-compA sub-compB

```

```

sub-comp : ∀ {U} {V} {W} {K} {E : Expression'' U K} {σ : Sub V W} {ρ : Sub U V} →
  E [[ σ • ρ ]] ≡ E [[ ρ ]] [[ σ ]]
sub-comp {E = var _} = ref
sub-comp {U} {V} {W} {K} {app c EE} = wd (app c) sub-compB

```

Lemma 4. *The alphabets and substitutions form a category under this composition.*

```

assoc : ∀ {U V W X} {ρ : Sub W X} {σ : Sub V W} {τ : Sub U V} → ρ • (σ • τ) ~ (ρ • σ)
assoc {τ = τ} K x = sym (sub-comp {E = τ K x})

```

```

sub-unitl : ∀ {U} {V} {σ : Sub U V} → idSub • σ ~ σ
sub-unitl _ _ = subid

```

```

sub-unitr : ∀ {U} {V} {σ : Sub U V} → σ • idSub ~ σ
sub-unitr _ _ = ref

```

Replacement is a special case of substitution:

Lemma 5. *Let ρ be a replacement $U \rightarrow V$.*

1. *The replacement (ρ, K) and the substitution (ρ, K) are equal.*

2.

$$E\langle \rho \rangle \equiv E[\rho]$$

$\text{Rep}\uparrow\text{-is-Sub}\uparrow : \forall \{U\} \{V\} \{\rho : \text{Rep } U \ V\} \{K\} \rightarrow (\lambda L \ x \rightarrow \text{var } (\text{Rep}\uparrow \{K = K\} \ \rho \ L \ x)) \sim \text{Sub}\uparrow$
 $\text{Rep}\uparrow\text{-is-Sub}\uparrow \ K \ x_0 = \text{ref}$
 $\text{Rep}\uparrow\text{-is-Sub}\uparrow \ K_1 \ (\uparrow x) = \text{ref}$

mutual

$\text{rep-is-sub} : \forall \{U\} \{V\} \{K\} \{E : \text{Expression}' \ U \ K\} \{\rho : \text{Rep } U \ V\} \rightarrow$
 $E \langle \rho \rangle \equiv E \llbracket (\lambda K \ x \rightarrow \text{var } (\rho \ K \ x)) \rrbracket$
 $\text{rep-is-sub} \{E = \text{var } _ \} = \text{ref}$
 $\text{rep-is-sub} \{U\} \{V\} \{K\} \{\text{app } c \ EE\} = \text{wd } (\text{app } c) \ \text{rep-is-subB}$

$\text{rep-is-subB} : \forall \{U\} \{V\} \{K\} \{C : \text{ConstructorKind } K\} \{EE : \text{Expression}' \ U \ (-\text{ConstructorKind } K)\}$
 $EE \langle \rho \rangle B \equiv EE \llbracket (\lambda K \ x \rightarrow \text{var } (\rho \ K \ x)) \rrbracket B$
 $\text{rep-is-subB} \{EE = \text{out}_2\} = \text{ref}$
 $\text{rep-is-subB} \{EE = \text{app}_2 _ _ \} = \text{wd}_2 \ \text{app}_2 \ \text{rep-is-subA} \ \text{rep-is-subB}$

$\text{rep-is-subA} : \forall \{U\} \{V\} \{K\} \{A : \text{Expression}' \ U \ -\text{Abstraction } K\} \{\rho : \text{Rep } U \ V\} \rightarrow$
 $A \langle \rho \rangle A \equiv A \llbracket (\lambda K \ x \rightarrow \text{var } (\rho \ K \ x)) \rrbracket A$
 $\text{rep-is-subA} \{A = \text{out } E\} = \text{wd out } \text{rep-is-sub}$
 $\text{rep-is-subA} \{U\} \{V\} \{ \Pi (\text{varKind } K) \ L \} \{A \{K\} \{L\} \ A\} \{\rho\} = \text{wd } \Lambda \ (\text{let open Equationals})$
 $\therefore A \langle \text{Rep}\uparrow \ \rho \rangle A$
 $\equiv A \llbracket (\lambda M \ x \rightarrow \text{var } (\text{Rep}\uparrow \ \rho \ M \ x)) \rrbracket A \ [\ \text{rep-is-subA} \]$
 $\equiv A \llbracket \text{Sub}\uparrow (\lambda M \ x \rightarrow \text{var } (\rho \ M \ x)) \rrbracket A \ [\ \text{sub-wdA } \text{Rep}\uparrow\text{-is-Sub}\uparrow \]$

Let E be an expression of kind K over V . Then we write $[x_0 := E]$ for the following substitution $(V, K) \Rightarrow V$:

$x_0 := : \forall \{V\} \{K\} \rightarrow \text{Expression}' \ V \ (\text{varKind } K) \rightarrow \text{Sub } (V, K) \ V$
 $x_0 := E _ \ x_0 = E$
 $x_0 := E \ K_1 \ (\uparrow x) = \text{var } x$

Lemma 6. 1.

$$\rho \bullet_1 [x_0 := E] \sim [x_0 := E\langle \rho \rangle] \bullet_2 (\rho, K)$$

2.

$$\sigma \bullet [x_0 := E] \sim [x_0 := E[\sigma]] \bullet (\sigma, K)$$

$\text{comp}_1\text{-botsub} : \forall \{U\} \{V\} \{K\} \{E : \text{Expression}' \ U \ (\text{varKind } K)\} \{\rho : \text{Rep } U \ V\} \rightarrow$
 $\rho \bullet_1 (x_0 := E) \sim (x_0 := (\text{rep } E \ \rho)) \bullet_2 \text{Rep}\uparrow \ \rho$
 $\text{comp}_1\text{-botsub} _ \ x_0 = \text{ref}$

`comp1-botsub _ (↑ _) = ref`

`comp-botsub : ∀ {U} {V} {K} {E : Expression'' U (varKind K)} {σ : Sub U V} →
 σ • (x0 := E) ~ (x0 := (E ∥ σ ∥)) • Sub↑ σ
 comp-botsub _ x0 = ref
 comp-botsub {σ = σ} L (↑ x) = trans (sym subid) (sub-comp2 {E = σ L x})`

4 Contexts

A *context* has the form $x_1 : A_1, \dots, x_n : A_n$ where, for each i :

- x_i is a variable of kind K_i distinct from x_1, \dots, x_{i-1} ;
- A_i is an expression of some kind L_i ;
- L_i is a parent of K_i .

The *domain* of this context is the alphabet $\{x_1, \dots, x_n\}$.

`data Context : Alphabet → Set where
 ⟨⟩ : Context ∅
 , : ∀ {V} {K} → Context V → Expression'' V (parent K) → Context (V , K)
 typeof : ∀ {V} {K} (x : Var V K) (Γ : Context V) → Expression'' V (parent K)
 typeof x0 (_, A) = lift A
 typeof (↑ x) (Γ , _) = lift (typeof x Γ)`

`record Grammar : Set1 where
 field
 taxonomy : Taxonomy
 toGrammar : ToGrammar taxonomy
 open Taxonomy taxonomy public
 open ToGrammar toGrammar public`

`module PL where`

`open import Prelims
 open import Grammar
 import Reduction`

5 Propositional Logic

Fix sets of *proof variables* and *term variables*.

The syntax of the system is given by the following grammar.

Proof	δ	$::=$	$p \mid \delta\delta \mid \lambda p : \phi.\delta$
Proposition	f	$::=$	$\perp \mid \phi \rightarrow \phi$
Context	Γ	$::=$	$\langle \rangle \mid \Gamma, p : \phi$
Judgement	\mathcal{J}	$::=$	$\Gamma \vdash \delta : \phi$

where p ranges over proof variables and x ranges over term variables. The variable p is bound within δ in the proof $\lambda p : \phi.\delta$, and the variable x is bound within M in the term $\lambda x : A.M$. We identify proofs and terms up to α -conversion.

```
data PLVarKind : Set where
  -Proof : PLVarKind
```

```
data PLNonVarKind : Set where
  -Prp : PLNonVarKind
```

```
PLtaxonomy : Taxonomy
PLtaxonomy = record {
  VarKind = PLVarKind;
  NonVarKind = PLNonVarKind }
```

```
module PLgrammar where
  open Grammar.Taxonomy PLtaxonomy
```

```
data PLCon :  $\forall$  {K : ExpressionKind}  $\rightarrow$  ConstructorKind K  $\rightarrow$  Set where
  app : PLCon ( $\Pi_2$  (out (varKind -Proof)) ( $\Pi_2$  (out (varKind -Proof)) (out2 {K = varKind})))
  lam : PLCon ( $\Pi_2$  (out (nonVarKind -Prp)) ( $\Pi_2$  ( $\Pi$  (varKind -Proof) (out (varKind -Proof)))))
  bot : PLCon (out2 {K = nonVarKind -Prp})
  imp : PLCon ( $\Pi_2$  (out (nonVarKind -Prp)) ( $\Pi_2$  (out (nonVarKind -Prp)) (out2 {K = nonVarKind})))
```

```
PLparent : VarKind  $\rightarrow$  ExpressionKind
PLparent -Proof = nonVarKind -Prp
```

```
open PLgrammar
```

```
Propositional-Logic : Grammar
Propositional-Logic = record {
  taxonomy = PLtaxonomy;
  toGrammar = record {
    Constructor = PLCon;
    parent = PLparent } }
```

```
open Grammar.Grammar Propositional-Logic
open Reduction Propositional-Logic
```

```
Prp : Set
Prp = Expression''  $\emptyset$  (nonVarKind -Prp)
```

```

⊥P : Prp
⊥P = app bot out2

_⇒_ : ∀ {P} → Expression'' P (nonVarKind -Prp) → Expression'' P (nonVarKind -Prp) → Expression'' P (nonVarKind -Prp)
φ ⇒ ψ = app imp (app2 (out φ) (app2 (out ψ) out2))

Proof : Alphabet → Set
Proof P = Expression'' P (varKind -Proof)

appP : ∀ {P} → Expression'' P (varKind -Proof) → Expression'' P (varKind -Proof) → Expression'' P (varKind -Proof)
appP δ ε = app app (app2 (out δ) (app2 (out ε) out2))

ΛP : ∀ {P} → Expression'' P (nonVarKind -Prp) → Expression'' (P , -Proof) (varKind -Proof)
ΛP φ δ = app lam (app2 (out φ) (app2 (Λ (out δ)) out2))

data β : Reduction where
  βI : ∀ {V} {φ} {δ} {ε} → β {V} app (app2 (out (ΛP φ δ)) (app2 (out ε) out2)) (δ [ x0 :=
    β-respects-rep : respect-rep β
  β-respects-rep {U} {V} {ρ = ρ} (βI .{U} {φ} {δ} {ε}) = subst (β app _)
    (let open Equational-Reasoning (Expression'' V (varKind -Proof)) in
      ∴ (rep δ (Rep↑ ρ)) [ x0 := (rep ε ρ) ]
      ≡ δ [ x0 := (rep ε ρ) •2 Rep↑ ρ ] [ [ sub-comp2 {E = δ} ] ]
      ≡ δ [ ρ •1 x0 := ε ] [ [ sub-wd {E = δ} comp1-botsub ] ]
      ≡ rep (δ [ x0 := ε ]) ρ [ sub-comp1 {E = δ} ] )
  βI

β-creates-rep : create-rep β
β-creates-rep = record {
  created = created;
  red-created = red-created;
  rep-created = rep-created } where
  created : ∀ {U V : Alphabet} {K} {C} {c : PLCon C} {EE : Expression' U (-Constructor K) C}
    created {c = app} {EE = app2 (out (var _)) _} ()
    created {c = app} {EE = app2 (out (app app _)) _} ()
    created {c = app} {EE = app2 (out (app lam (app2 (out φ) (app2 (Λ (out δ)) out2))))} (app)
    created {c = lam} ()
    created {c = bot} ()
    created {c = imp} ()
  red-created : ∀ {U} {V} {K} {C} {c : PLCon C} {EE : Expression' U (-Constructor K) C}
    red-created {c = app} {EE = app2 (out (var _)) _} ()
    red-created {c = app} {EE = app2 (out (app app _)) _} ()
    red-created {c = app} {EE = app2 (out (app lam (app2 (out φ) (app2 (Λ (out δ)) out2))))} (app)
    red-created {c = lam} ()
    red-created {c = bot} ()

```

```

red-created {c = imp} ()
rep-created : ∀ {U} {V} {K} {C} {c : PLCon C} {EE : Expression' U (-Constructor K) C}
rep-created {c = app} {EE = app2 (out (var _)) _} ()
rep-created {c = app} {EE = app2 (out (app app _)) _} ()
rep-created {c = app} {EE = app2 (out (app lam (app2 (out φ) (app2 (Λ (out δ)) out2))) ) _} ()
  ∴ rep (δ [ x0 := ε ]) ρ
  ≡ δ [ ρ •1 x0 := ε ] [[ sub-comp1 {E = δ} ]]
  ≡ δ [ x0 := (rep ε ρ) •2 Rep↑ ρ ] [ sub-wd {E = δ} comp1-botsub ]
  ≡ rep δ (Rep↑ ρ) [ x0 := (rep ε ρ) ] [ sub-comp2 {E = δ} ]
rep-created {c = lam} ()
rep-created {c = bot} ()
rep-created {c = imp} ()

```

The rules of deduction of the system are as follows.

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash p : \phi} (p : \phi \in \Gamma)$$

$$\frac{\Gamma \vdash \delta : \phi \rightarrow \psi}{\Gamma \vdash \delta \epsilon : \psi \quad \Gamma \vdash \epsilon : \phi}$$

$$\frac{\Gamma, p : \phi \vdash \delta : \psi}{\Gamma \vdash \lambda p : \phi. \delta : \phi \rightarrow \psi}$$

```

infix 10 _⊢_::_
data _⊢_::_ : ∀ {P} → Context P → Proof P → Expression'' P (nonVarKind -Prp) → Set where
  var : ∀ {P} {Γ : Context P} {p : Var P -Proof} → Γ ⊢ var p :: typeof p Γ
  app : ∀ {P} {Γ : Context P} {δ} {ε} {φ} {ψ} → Γ ⊢ δ :: φ ⇒ ψ → Γ ⊢ ε :: φ → Γ ⊢ appP
  Λ : ∀ {P} {Γ : Context P} {φ} {δ} {ψ} → (__,_ {K = -Proof} Γ φ) ⊢ δ :: lift ψ → Γ ⊢ ΛP

```

A *replacement* ρ from a context Γ to a context Δ , $\rho : \Gamma \rightarrow \Delta$, is a replacement on the syntax such that, for every $x : \phi$ in Γ , we have $\rho(x) : \phi \in \Delta$.

```

_::_⇒R_ : ∀ {P} {Q} → Rep P Q → Context P → Context Q → Set
ρ :: Γ ⇒R Δ = ∀ x → typeof (ρ -Proof x) Δ ≡ rep (typeof x Γ) ρ

```

```

↑-typed : ∀ {P} {Γ : Context P} {φ : Expression'' P (nonVarKind -Prp)} →
  (λ _ → ↑) :: Γ ⇒R (__,_ {P} { -Proof} Γ φ)

```

```

↑-typed x0 = ref
↑-typed (↑ _) = ref

```

```

Rep↑-typed : ∀ {P} {Q} {ρ} {Γ : Context P} {Δ : Context Q} {φ : Expression'' P (nonVarKind -Prp)}
  Rep↑ ρ :: (__,_ {P} { -Proof} Γ φ) ⇒R (__,_ {Q} { -Proof} Δ (rep φ ρ))
Rep↑-typed {Q = Q} {ρ = ρ} {φ = φ} ρ::Γ⇒Δ x0 = let open Equational-Reasoning (Expression' U (-Constructor K) C)
  ∴ rep (rep φ ρ) (λ _ → ↑)
  ≡ rep φ (λ K x → ↑ (ρ K x)) [[ rep-comp {E = φ} ]]
  ≡ rep (rep φ (λ _ → ↑)) (Rep↑ ρ) [ rep-comp {E = φ} ]

```


$\text{Rep}\uparrow\text{-typed } \{Q = Q\} \{ \rho = \rho \} \{ \Gamma = \Gamma \} \{ \Delta = \Delta \} \rho :: \Gamma \rightarrow \Delta \ (\uparrow x) = \text{let open Equational-Reasoning}$
 $\therefore \text{rep (typeof } (\rho \text{ -Proof } x) \Delta) (\lambda _ \rightarrow \uparrow)$
 $\equiv \text{rep (rep (typeof } x \Gamma) \rho) (\lambda _ \rightarrow \uparrow) \quad [\text{wd } (\lambda p \rightarrow \text{rep } p (\lambda _ \rightarrow \uparrow)) (\rho :: \Gamma \rightarrow \Delta \ x)]$
 $\equiv \text{rep (typeof } x \Gamma) (\lambda K x \rightarrow \uparrow (\rho K x)) \quad [[\text{rep-comp } \{E = \text{typeof } x \Gamma\}]]$
 $\equiv \text{rep (rep (typeof } x \Gamma) (\lambda _ \rightarrow \uparrow)) (\text{Rep}\uparrow \rho) [\text{rep-comp } \{E = \text{typeof } x \Gamma\}]$

The replacements between contexts are closed under composition.

$\bullet \text{R-typed} : \forall \{P\} \{Q\} \{R\} \{\sigma : \text{Rep } Q \ R\} \{\rho : \text{Rep } P \ Q\} \{\Gamma\} \{\Delta\} \{\Theta\} \rightarrow \rho :: \Gamma \Rightarrow_R \Delta \rightarrow \sigma :: \Delta \Rightarrow_R \Theta$
 $\sigma \bullet_R \rho :: \Gamma \Rightarrow_R \Theta$
 $\bullet \text{R-typed } \{R = R\} \{\sigma\} \{\rho\} \{\Gamma\} \{\Delta\} \{\Theta\} \rho :: \Gamma \rightarrow \Delta \ \sigma :: \Delta \rightarrow \Theta \ x = \text{let open Equational-Reasoning (Exp)}$
 $\therefore \text{typeof } (\sigma \text{ -Proof } (\rho \text{ -Proof } x)) \Theta$
 $\equiv \text{rep (typeof } (\rho \text{ -Proof } x) \Delta) \sigma \quad [\sigma :: \Delta \rightarrow \Theta (\rho \text{ -Proof } x)]$
 $\equiv \text{rep (rep (typeof } x \Gamma) \rho) \sigma \quad [\text{wd } (\lambda x_1 \rightarrow \text{rep } x_1 \sigma) (\rho :: \Gamma \rightarrow \Delta \ x)]$
 $\equiv \text{rep (typeof } x \Gamma) (\sigma \bullet_R \rho) \quad [[\text{rep-comp}]]$

Weakening Lemma

$\text{Weakening} : \forall \{P\} \{Q\} \{\Gamma : \text{Context } P\} \{\Delta : \text{Context } Q\} \{\rho\} \{\delta\} \{\varphi\} \rightarrow \Gamma \vdash \delta :: \varphi \rightarrow \rho :: \Gamma$
 $\text{Weakening } \{P\} \{Q\} \{\Gamma\} \{\Delta\} \{\rho\} (\text{var } \{p = p\}) \rho :: \Gamma \rightarrow \Delta = \text{subst } (\lambda P \rightarrow \Delta \vdash \text{var } (\rho \text{ -Proof } p))$
 $\text{Weakening } (\text{app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ \Gamma \vdash \varepsilon :: \varphi) \rho :: \Gamma \rightarrow \Delta = \text{app } (\text{Weakening } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ \rho :: \Gamma \rightarrow \Delta) (\text{Weakening } \Gamma \vdash \varepsilon :: \varphi)$
 $\text{Weakening } .\{P\} \{Q\} .\{\Gamma\} \{\Delta\} \{\rho\} (\Lambda \{P\} \{\Gamma\} \{\varphi\} \{\delta\} \{\psi\} \Gamma, \varphi \vdash \delta :: \psi) \rho :: \Gamma \rightarrow \Delta = \Lambda$
 $(\text{subst } (\lambda P \rightarrow (_, _ \{Q\} \{ \text{-Proof} \} \Delta (\text{rep } \varphi \rho)) \vdash \text{rep } \delta (\text{Rep}\uparrow \rho) :: P)$
 $(\text{let open Equational-Reasoning (Expression'' } (Q, \text{-Proof}) (\text{nonVarKind -Prp})) \text{ in}$
 $\therefore \text{rep (rep } \psi (\lambda _ \rightarrow \uparrow)) (\text{Rep}\uparrow \rho)$
 $\equiv \text{rep } \psi (\lambda _ x \rightarrow \uparrow (\rho _ x)) \quad [[\text{rep-comp } \{E = \psi\}]]$
 $\equiv \text{rep (rep } \psi \rho) (\lambda _ \rightarrow \uparrow) \quad [\text{rep-comp } \{E = \psi\}])$
 $(\text{Weakening } \Gamma, \varphi \vdash \delta :: \psi (\text{Rep}\uparrow\text{-typed } \rho :: \Gamma \rightarrow \Delta)))$

A *substitution* σ from a context Γ to a context Δ , $\sigma : \Gamma \rightarrow \Delta$, is a substitution σ on the syntax such that, for every $x : \phi$ in Γ , we have $\Delta \vdash \sigma(x) : \phi$.

$_ :: _ \Rightarrow _ : \forall \{P\} \{Q\} \rightarrow \text{Sub } P \ Q \rightarrow \text{Context } P \rightarrow \text{Context } Q \rightarrow \text{Set}$
 $\sigma :: \Gamma \Rightarrow \Delta = \forall x \rightarrow \Delta \vdash \sigma _ x :: \text{typeof } x \Gamma \llbracket \sigma \rrbracket$

$\text{Sub}\uparrow\text{-typed} : \forall \{P\} \{Q\} \{\sigma\} \{\Gamma : \text{Context } P\} \{\Delta : \text{Context } Q\} \{\varphi : \text{Expression'' } P (\text{nonVarKind } P)\}$
 $\text{Sub}\uparrow\text{-typed } \{P\} \{Q\} \{\sigma\} \{\Gamma\} \{\Delta\} \{\varphi\} \sigma :: \Gamma \rightarrow \Delta \ x_0 = \text{subst } (\lambda p \rightarrow (_, _ \{Q\} \{ \text{-Proof} \} \Delta (\varphi \llbracket \sigma \rrbracket$
 $(\text{let open Equational-Reasoning (Expression'' } (Q, \text{-Proof}) (\text{nonVarKind -Prp})) \text{ in}$
 $\therefore \text{rep } (\varphi \llbracket \sigma \rrbracket) (\lambda _ \rightarrow \uparrow)$
 $\equiv \varphi \llbracket (\lambda _ \rightarrow \uparrow) \bullet_1 \sigma \rrbracket \quad [[\text{sub-comp}_1 \{E = \varphi\}]]$
 $\equiv \text{rep } \varphi (\lambda _ \rightarrow \uparrow) \llbracket \text{Sub}\uparrow \sigma \rrbracket [\text{sub-comp}_2 \{E = \varphi\}])$
 var
 $\text{Sub}\uparrow\text{-typed } \{Q = Q\} \{\sigma = \sigma\} \{\Gamma = \Gamma\} \{\Delta = \Delta\} \{\varphi = \varphi\} \sigma :: \Gamma \rightarrow \Delta \ (\uparrow x) =$
 subst
 $(\lambda P \rightarrow _, _ \{Q\} \{ \text{-Proof} \} \Delta (\varphi \llbracket \sigma \rrbracket) \vdash \text{Sub}\uparrow \sigma \text{-Proof } (\uparrow x) :: P)$
 $(\text{let open Equational-Reasoning (Expression'' } (Q, \text{-Proof}) (\text{nonVarKind -Prp})) \text{ in}$
 $\therefore \text{rep (typeof } x \Gamma \llbracket \sigma \rrbracket) (\lambda _ \rightarrow \uparrow)$

$\equiv \text{typeof } x \Gamma \llbracket (\lambda _ \rightarrow \uparrow) \bullet_1 \sigma \rrbracket \quad \llbracket \text{sub-comp}_1 \{E = \text{typeof } x \Gamma\} \rrbracket$
 $\equiv \text{rep } (\text{typeof } x \Gamma) (\lambda _ \rightarrow \uparrow) \llbracket \text{Sub}\uparrow \sigma \rrbracket \llbracket \text{sub-comp}_2 \{E = \text{typeof } x \Gamma\} \rrbracket$
 $(\text{Weakening } (\sigma :: \Gamma \rightarrow \Delta \ x) (\uparrow\text{-typed } \{\varphi = \varphi \llbracket \sigma \rrbracket\}))$

$\text{botsub-typed} : \forall \{P\} \{\Gamma : \text{Context } P\} \{\varphi : \text{Expression'' } P \text{ (nonVarKind -Prp)}\} \{\delta\} \rightarrow$
 $\Gamma \vdash \delta :: \varphi \rightarrow x_0 := \delta :: (_, _ \{P\} \{ \text{-Proof} \} \Gamma \varphi) \Rightarrow \Gamma$
 $\text{botsub-typed } \{P\} \{\Gamma\} \{\varphi\} \{\delta\} \Gamma \vdash \delta :: \varphi \ x_0 = \text{subst } (\lambda P_1 \rightarrow \Gamma \vdash \delta :: P_1)$
 $(\text{let open Equational-Reasoning (Expression'' } P \text{ (nonVarKind -Prp)) in}$
 $\therefore \varphi$
 $\equiv \varphi \llbracket \text{idSub} \rrbracket \quad \llbracket \text{subid} \rrbracket$
 $\equiv \text{rep } \varphi (\lambda _ \rightarrow \uparrow) \llbracket x_0 := \delta \rrbracket \llbracket \text{sub-comp}_2 \{E = \varphi\} \rrbracket$
 $\Gamma \vdash \delta :: \varphi$
 $\text{botsub-typed } \{P\} \{\Gamma\} \{\varphi\} \{\delta\} _ (\uparrow x) = \text{subst } (\lambda P_1 \rightarrow \Gamma \vdash \text{var } x :: P_1)$
 $(\text{let open Equational-Reasoning (Expression'' } P \text{ (nonVarKind -Prp)) in}$
 $\therefore \text{typeof } x \Gamma$
 $\equiv \text{typeof } x \Gamma \llbracket \text{idSub} \rrbracket \quad \llbracket \text{subid} \rrbracket$
 $\equiv \text{rep } (\text{typeof } x \Gamma) (\lambda _ \rightarrow \uparrow) \llbracket x_0 := \delta \rrbracket \llbracket \text{sub-comp}_2 \{E = \text{typeof } x \Gamma\} \rrbracket$
 var

Substitution Lemma

$\text{Substitution} : \forall \{P\} \{Q\} \{\Gamma : \text{Context } P\} \{\Delta : \text{Context } Q\} \{\delta\} \{\varphi\} \{\sigma\} \rightarrow \Gamma \vdash \delta :: \varphi \rightarrow \sigma ::$
 $\text{Substitution var } \sigma :: \Gamma \rightarrow \Delta = \sigma :: \Gamma \rightarrow \Delta _$
 $\text{Substitution (app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \Gamma \vdash \varepsilon :: \varphi) \sigma :: \Gamma \rightarrow \Delta = \text{app (Substitution } \Gamma \vdash \delta :: \varphi \rightarrow \psi \sigma :: \Gamma \rightarrow \Delta) (\text{Substitution}$
 $\text{Substitution } \{Q = Q\} \{\Delta = \Delta\} \{\sigma = \sigma\} (\Lambda \{P\} \{\Gamma\} \{\varphi\} \{\delta\} \{\psi\} \Gamma, \varphi \vdash \delta :: \psi) \sigma :: \Gamma \rightarrow \Delta = \Lambda$
 $(\text{subst } (\lambda p \rightarrow _, _ \{Q\} \{ \text{-Proof} \} \Delta (\varphi \llbracket \sigma \rrbracket) \vdash \delta \llbracket \text{Sub}\uparrow \sigma \rrbracket :: p)$
 $(\text{let open Equational-Reasoning (Expression'' } (Q, \text{-Proof}) \text{ (nonVarKind -Prp)) in}$
 $\therefore \text{rep } \psi (\lambda _ \rightarrow \uparrow) \llbracket \text{Sub}\uparrow \sigma \rrbracket$
 $\equiv \psi \llbracket \text{Sub}\uparrow \sigma \bullet_2 (\lambda _ \rightarrow \uparrow) \rrbracket \llbracket \text{sub-comp}_2 \{E = \psi\} \rrbracket$
 $\equiv \text{rep } (\psi \llbracket \sigma \rrbracket) (\lambda _ \rightarrow \uparrow) \llbracket \text{sub-comp}_1 \{E = \psi\} \rrbracket$
 $(\text{Substitution } \Gamma, \varphi \vdash \delta :: \psi (\text{Sub}\uparrow\text{-typed } \sigma :: \Gamma \rightarrow \Delta)))$

Subject Reduction

$\text{prop-triv-red} : \forall \{P\} \{\varphi \psi : \text{Expression'' } P \text{ (nonVarKind -Prp)}\} \rightarrow \varphi \rightarrow \langle \beta \rangle \psi \rightarrow \text{False}$
 $\text{prop-triv-red } \{ _ \} \{\text{app bot out}_2\} (\text{redex } ())$
 $\text{prop-triv-red } \{P\} \{\text{app bot out}_2\} (\text{app } ())$
 $\text{prop-triv-red } \{P\} \{\text{app imp (app}_2 _ (\text{app}_2 _ \text{out}_2))\} (\text{redex } ())$
 $\text{prop-triv-red } \{P\} \{\text{app imp (app}_2 (\text{out } \varphi) (\text{app}_2 \psi \text{out}_2))\} (\text{app (appl (out } \varphi \rightarrow \varphi')))) = \text{prop-triv-red}$
 $\text{prop-triv-red } \{P\} \{\text{app imp (app}_2 \varphi (\text{app}_2 (\text{out } \psi) \text{out}_2))\} (\text{app (appr (appl (out } \psi \rightarrow \psi'))))$
 $\text{prop-triv-red } \{P\} \{\text{app imp (app}_2 _ (\text{app}_2 (\text{out } _) \text{out}_2))\} (\text{app (appr (appr ())))$

$\text{SR} : \forall \{P\} \{\Gamma : \text{Context } P\} \{\delta \varepsilon : \text{Proof } P\} \{\varphi\} \rightarrow \Gamma \vdash \delta :: \varphi \rightarrow \delta \rightarrow \langle \beta \rangle \varepsilon \rightarrow \Gamma \vdash \varepsilon :: \varphi$
 $\text{SR var } ()$
 $\text{SR (app } \{\varepsilon = \varepsilon\} (\Lambda \{P\} \{\Gamma\} \{\varphi\} \{\delta\} \{\psi\} \Gamma, \varphi \vdash \delta :: \psi) \Gamma \vdash \varepsilon :: \varphi) (\text{redex } \beta I) =$
 $\text{subst } (\lambda P_1 \rightarrow \Gamma \vdash \delta \llbracket x_0 := \varepsilon \rrbracket :: P_1)$
 $(\text{let open Equational-Reasoning (Expression'' } P \text{ (nonVarKind -Prp)) in}$

```

 $\vdash \text{rep } \psi \ (\lambda \_ \rightarrow \uparrow) \ [\![ \text{x}_0 := \varepsilon \ ]\!] \\
\equiv \psi \ [\![ \text{idSub} \ ]\!] \quad [\![ \text{sub-comp}_2 \ \{E = \psi\} \ ]\!] \\
\equiv \psi \quad [\![ \text{subid} \ ]\!] \\
(\text{Substitution } \Gamma, \varphi \vdash \delta :: \psi \ (\text{botsub-typed } \Gamma \vdash \varepsilon :: \varphi)) \\
\text{SR } (\text{app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ \Gamma \vdash \varepsilon :: \varphi) \ (\text{app } (\text{appl } (\text{out } \delta \rightarrow \delta')))) = \text{app } (\text{SR } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ \delta \rightarrow \delta') \ \Gamma \vdash \varepsilon :: \varphi \\
\text{SR } (\text{app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ \Gamma \vdash \varepsilon :: \varphi) \ (\text{app } (\text{appr } (\text{appl } (\text{out } \varepsilon \rightarrow \varepsilon'))))) = \text{app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ (\text{SR } \Gamma \vdash \varepsilon :: \varphi \ \varepsilon \rightarrow \varepsilon') \\
\text{SR } (\text{app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \ \Gamma \vdash \varepsilon :: \varphi) \ (\text{app } (\text{appr } (\text{appr } ()))) \\
\text{SR } (\Lambda \ \Gamma \vdash \delta :: \varphi) \ (\text{redex } ()) \\
\text{SR } (\Lambda \ \Gamma \vdash \delta :: \varphi) \ (\text{app } (\text{appl } (\text{out } \varphi \rightarrow \varphi')))) \text{ with prop-triv-red } \varphi \rightarrow \varphi' \\
\dots \mid () \\
\text{SR } (\Lambda \ \Gamma \vdash \delta :: \varphi) \ (\text{app } (\text{appr } (\text{appl } (\Lambda \ (\text{out } \delta \rightarrow \delta')))))) = \Lambda \ (\text{SR } \Gamma \vdash \delta :: \varphi \ \delta \rightarrow \delta') \\
\text{SR } (\Lambda \ \Gamma \vdash \delta :: \varphi) \ (\text{app } (\text{appr } (\text{appr } ())))$ 
```

We define the sets of *computable* proofs $C_\Gamma(\phi)$ for each context Γ and proposition ϕ as follows:

$$C_\Gamma(\perp) = \{\delta \mid \Gamma \vdash \delta : \perp, \delta \in SN\}$$

$$C_\Gamma(\phi \rightarrow \psi) = \{\delta \mid \Gamma : \delta : \phi \rightarrow \psi, \forall \epsilon \in C_\Gamma(\phi). \delta \epsilon \in C_\Gamma(\psi)\}$$

```

C :  $\forall \{P\} \rightarrow \text{Context } P \rightarrow \text{Prp} \rightarrow \text{Proof } P \rightarrow \text{Set}$ 
C  $\Gamma$  (app bot out2)  $\delta$  = ( $\Gamma \vdash \delta :: \text{rep } \perp P \ (\lambda \_ ())$ )  $\wedge$  SN  $\beta$   $\delta$ 
C  $\Gamma$  (app imp (app2 (out  $\varphi$ ) (app2 (out  $\psi$ ) out2)))  $\delta$  = ( $\Gamma \vdash \delta :: \text{rep } (\varphi \Rightarrow \psi) \ (\lambda \_ ())$ )  $\wedge$ 
  ( $\forall Q \ \{\Delta : \text{Context } Q\} \ \rho \ \varepsilon \rightarrow \rho :: \Gamma \Rightarrow R \ \Delta \rightarrow C \ \Delta \ \varphi \ \varepsilon \rightarrow C \ \Delta \ \psi \ (\text{appP } (\text{rep } \delta \ \rho) \ \varepsilon)$ )

C-typed :  $\forall \{P\} \ \{\Gamma : \text{Context } P\} \ \{\varphi\} \ \{\delta\} \rightarrow C \ \Gamma \ \varphi \ \delta \rightarrow \Gamma \vdash \delta :: \text{rep } \varphi \ (\lambda \_ ())$ 
C-typed  $\{\varphi = \text{app bot out}_2\} = \pi_1$ 
C-typed  $\{\Gamma = \Gamma\} \ \{\varphi = \text{app imp } (\text{app}_2 \ (\text{out } \varphi) \ (\text{app}_2 \ (\text{out } \psi) \ \text{out}_2))\} \ \{\delta = \delta\} = \lambda \ x \rightarrow \text{subst } ($ 
  (wd2  $\_ \Rightarrow \_$  (rep-wd  $\{E = \varphi\} \ (\lambda \ ())$ ) (rep-wd  $\{E = \psi\} \ (\lambda \ ())$ ))
  ( $\pi_1 \ x$ )

C-rep :  $\forall \{P\} \ \{Q\} \ \{\Gamma : \text{Context } P\} \ \{\Delta : \text{Context } Q\} \ \{\varphi\} \ \{\delta\} \ \{\rho\} \rightarrow C \ \Gamma \ \varphi \ \delta \rightarrow \rho :: \Gamma \Rightarrow R \ \Delta$ 
C-rep  $\{\varphi = \text{app bot out}_2\} \ (\Gamma \vdash \delta :: \perp, \text{SN} \delta) \ \rho :: \Gamma \rightarrow \Delta = (\text{Weakening } \Gamma \vdash \delta :: \perp \ \rho :: \Gamma \rightarrow \Delta), \text{SNrep } \beta\text{-crea}$ 
C-rep  $\{P\} \ \{Q\} \ \{\Gamma\} \ \{\Delta\} \ \{\text{app imp } (\text{app}_2 \ (\text{out } \varphi) \ (\text{app}_2 \ (\text{out } \psi) \ \text{out}_2))\} \ \{\delta\} \ \{\rho\} \ (\Gamma \vdash \delta :: \varphi \Rightarrow \psi, C\delta)$ 
  (let open Equational-Reasoning (Expression'' Q (nonVarKind -Prp)) in
     $\vdash \text{rep } (\text{rep } \varphi \_) \ \rho$ 
     $\equiv \text{rep } \varphi \_ \quad [\![ \text{rep-comp} \ ]\!]$ 
     $\equiv \text{rep } \varphi \_ \quad [\![ \text{rep-wd } (\lambda \ ()) \ ]\!]$ 
    (trans (sym rep-comp) (rep-wd  $(\lambda \ ())$ )) (Weakening  $\Gamma \vdash \delta :: \varphi \Rightarrow \psi \ \rho :: \Gamma \rightarrow \Delta$ ),
    ( $\lambda \ R \ \sigma \ \varepsilon \ \sigma :: \Delta \rightarrow \Theta \ \varepsilon \in C\varphi \rightarrow \text{subst } (C \_ \psi) \ (\text{wd } (\lambda \ x \rightarrow \text{appP } x \ \varepsilon) \ \text{rep-comp})$ 
      ( $C\delta \ R \ (\sigma \bullet R \ \rho) \ \varepsilon \ (\bullet R\text{-typed } \rho :: \Gamma \rightarrow \Delta \ \sigma :: \Delta \rightarrow \Theta) \ \varepsilon \in C\varphi$ ))

C-red :  $\forall \{P\} \ \{\Gamma : \text{Context } P\} \ \{\varphi\} \ \{\delta\} \ \{\varepsilon\} \rightarrow C \ \Gamma \ \varphi \ \delta \rightarrow \delta \rightarrow \langle \beta \rangle \ \varepsilon \rightarrow C \ \Gamma \ \varphi \ \varepsilon$ 
C-red  $\{\varphi = \text{app bot out}_2\} \ (\Gamma \vdash \delta :: \perp, \text{SN} \delta) \ \delta \rightarrow \varepsilon = (\text{SR } \Gamma \vdash \delta :: \perp \ \delta \rightarrow \varepsilon), (\text{SNred } \text{SN} \delta \ (\text{osr-red } \delta \rightarrow \varepsilon))$ 
C-red  $\{\Gamma = \Gamma\} \ \{\varphi = \text{app imp } (\text{app}_2 \ (\text{out } \varphi) \ (\text{app}_2 \ (\text{out } \psi) \ \text{out}_2))\} \ \{\delta = \delta\} \ (\Gamma \vdash \delta :: \varphi \Rightarrow \psi, C\delta) \ \delta \rightarrow \varepsilon$ 
  (wd2  $\_ \Rightarrow \_$  (rep-wd  $(\lambda \ ())$ ) (rep-wd  $(\lambda \ ())$ ))

```

$\Gamma \vdash \delta :: \varphi \Rightarrow \psi \mid \delta \rightarrow \delta' \rangle$,
 $(\lambda Q \rho \varepsilon \in \rho :: \Gamma \rightarrow \Delta \ \varepsilon \in C\varphi \rightarrow C\text{-red } \{\varphi = \psi\} \ (C\delta \ Q \ \rho \ \varepsilon \in \rho :: \Gamma \rightarrow \Delta \ \varepsilon \in C\varphi) \ (\text{app } (\text{appl } (\text{out } (\text{reposr } \beta$

The *neutral terms* are those that begin with a variable.

```
data Neutral {P} : Proof P → Set where
  varNeutral : ∀ x → Neutral (var x)
  appNeutral : ∀ δ ε → Neutral δ → Neutral (appP δ ε)
```

Lemma 7. *If δ is neutral and $\delta \rightarrow_\beta \epsilon$ then ϵ is neutral.*

```
neutral-red : ∀ {P} {δ ε : Proof P} → Neutral δ → δ →⟨ β ⟩ ε → Neutral ε
neutral-red (varNeutral _) ()
neutral-red (appNeutral .(app lam (app₂ (out _) (app₂ (λ (out _)) out₂))) _ ()) (redex βI)
neutral-red (appNeutral _ ε neutralδ) (app (appl (out δ → δ'))) = appNeutral _ ε (neutral-
neutral-red (appNeutral δ _ neutralδ) (app (appr (appl (out ε → ε')))) = appNeutral δ _ ne
neutral-red (appNeutral _ _ _) (app (appr (appr ())))
```

```
neutral-rep : ∀ {P} {Q} {δ : Proof P} {ρ : Rep P Q} → Neutral δ → Neutral (rep δ ρ)
neutral-rep {ρ = ρ} (varNeutral x) = varNeutral (ρ -Proof x)
neutral-rep {ρ = ρ} (appNeutral δ ε neutralδ) = appNeutral (rep δ ρ) (rep ε ρ) (neutral-
```

Lemma 8. *Let $\Gamma \vdash \delta : \phi$. If δ is neutral and, for all ϵ such that $\delta \rightarrow_\beta \epsilon$, we have $\epsilon \in C_\Gamma(\phi)$, then $\delta \in C_\Gamma(\phi)$.*

```
NeutralC-lm : ∀ {P} {δ ε : Proof P} {X : Proof P → Set} →
  Neutral δ →
  (∀ δ' → δ →⟨ β ⟩ δ' → X (appP δ' ε)) →
  (∀ ε' → ε →⟨ β ⟩ ε' → X (appP δ ε')) →
  ∀ χ → appP δ ε →⟨ β ⟩ χ → X χ
NeutralC-lm () _ _ . _ (redex βI)
NeutralC-lm _ hyp1 _ .(app app (app₂ (out _) (app₂ (out _) out₂))) (app (appl (out δ → δ')))
NeutralC-lm _ _ hyp2 .(app app (app₂ (out _) (app₂ (out _) out₂))) (app (appr (appl (out
NeutralC-lm _ _ _ .(app app (app₂ (out _) (app₂ (out _) _))) (app (appr (appr ())))
```

mutual

```
NeutralC : ∀ {P} {Γ : Context P} {δ : Proof P} {φ : Prp} →
  Γ ⊢ δ :: (rep φ (λ _ ())) → Neutral δ →
  (∀ ε → δ →⟨ β ⟩ ε → C Γ φ ε) →
  C Γ φ δ
NeutralC {P} {Γ} {δ} {app bot out₂} Γ ⊢ δ :: ⊥ Neutralδ hyp = Γ ⊢ δ :: ⊥ , SNI δ (λ ε δ → ε → π
NeutralC {P} {Γ} {δ} {app imp (app₂ (out φ) (app₂ (out ψ) out₂))) Γ ⊢ δ :: φ → ψ neutralδ hyp
  (λ Q ρ ε ∈ ρ :: Γ → Δ ε ∈ Cφ → claim ε (CsubSN {φ = φ} {δ = ε} ε ∈ Cφ) ρ :: Γ → Δ ε ∈ Cφ) where
  claim : ∀ {Q} {Δ} {ρ : Rep P Q} ε → SN β ε → ρ :: Γ ⇒R Δ → C Δ φ ε → C Δ ψ (appP
  claim {Q} {Δ} {ρ} ε (SNI .ε SNE) ρ :: Γ → Δ ε ∈ Cφ = NeutralC {Q} {Δ} {appP (rep δ ρ) ε} {
    (app (subst (λ P₁ → Δ ⊢ rep δ ρ :: P₁)
    (wd2 _⇒_
```

```

(
  (let open Equational-Reasoning (Expression'' Q (nonVarKind -Prp)) in
    ∴ rep (rep φ _) ρ
    ≡ rep φ _ [[ rep-comp ]]
    ≡ rep φ _ [[ rep-wd (λ ()) ]])
  (
    (let open Equational-Reasoning (Expression'' Q (nonVarKind -Prp)) in
      ∴ rep (rep ψ _) ρ
      ≡ rep ψ _ [[ rep-comp ]]
      ≡ rep ψ _ [[ rep-wd (λ ()) ]])
    ))
  (Weakening Γ ⊢ δ :: φ → ψ ρ :: Γ → Δ))
  (C-typed {Q} {Δ} {φ} {ε} ε ∈ Cφ))
  (appNeutral (rep δ ρ) ε (neutral-rep neutralδ))
  (NeutralC-lm {X = C Δ ψ} (neutral-rep neutralδ)
    (λ δ' δ⟨ρ⟩ → δ' →
      let δ₀ : Proof P
        δ₀ = create-reposr β-creates-rep δ⟨ρ⟩ → δ'
      in let δ → δ₀ : δ → (β) δ₀
        δ → δ₀ = red-create-reposr β-creates-rep δ⟨ρ⟩ → δ'
      in let δ₀⟨ρ⟩ ≡ δ' : rep δ₀ ρ ≡ δ'
        δ₀⟨ρ⟩ ≡ δ' = rep-create-reposr β-creates-rep δ⟨ρ⟩ → δ'
      in let δ₀ ∈ C[φ ⇒ ψ] : C Γ (φ ⇒ ψ) δ₀
        δ₀ ∈ C[φ ⇒ ψ] = hyp δ₀ δ → δ₀
      in let δ' ∈ C[φ ⇒ ψ] : C Δ (φ ⇒ ψ) δ'
        δ' ∈ C[φ ⇒ ψ] = subst (C Δ (φ ⇒ ψ)) δ₀⟨ρ⟩ ≡ δ' (C-rep {φ = φ ⇒ ψ} δ₀ ∈ C[φ ⇒ ψ] ρ)
      in subst (C Δ ψ) (wd (λ x → appP x ε) δ₀⟨ρ⟩ ≡ δ') (π₂ δ₀ ∈ C[φ ⇒ ψ] Q ρ ε ρ :: Γ → Δ ε ∈ Cφ))
      (λ ε' ε → ε' → claim ε' (SNε ε' ε → ε') ρ :: Γ → Δ (C-red {φ = φ} ε ∈ Cφ ε → ε'))))

```

Lemma 9.

$$C_{\Gamma}(\phi) \subseteq SN$$

```

CsubSN : ∀ {P} {Γ : Context P} {φ} {δ} → C Γ φ δ → SN β δ
CsubSN {P} {Γ} {ToGrammar.app bot ToGrammar.out₂} P₁ = π₂ P₁
CsubSN {P} {Γ} {app imp (app₂ (out φ) (app₂ (out ψ) out₂))} {δ} P₁ =
  let φ' : Expression'' P (nonVarKind -Prp)
    φ' = rep φ (λ _ ()) in
  let Γ' : Context (P , -Proof)
    Γ' = _,_ {K = -Proof} Γ φ' in
  SNrep' {P} {P , -Proof} { varKind -Proof} {λ _ → ↑} β-respects-rep
  (SNoutA
    (SNSubbody1 (SNSubexp (CsubSN {Γ = Γ'} {φ = φ}
      (π₂ P₁ (P , -Proof) (λ _ → ↑) (var x₀) (λ _ → ref)
        (NeutralC {φ = φ} (subst (λ x → Γ' ⊢ var x₀ :: x) (trans (sym rep-comp) (rep-wd
          (varNeutral x₀)
            (λ _ ())))))))))
    --(subst (λ x → (λ _ {K = -Proof} Γ (rep φ _)) ⊢ var x₀ :: x) {rep (rep φ _) _} {rep φ _}
    {-
      (π₂ P₁ (P , -Proof) (λ _ → ↑) (var x₀)

```

```

(↑-typed {Γ = Γ} {φ = rep φ (λ _ ())})
(NeutralC {φ = φ}
  (subst (λ x → _ ⊢ var x₀ :: x) {!!} var)
  (varNeutral x₀)
  (λ _ ()))))) -}

```

```

module PHOPL where
open import Prelims hiding (⊥)
open import Grammar
open import Reduction

```

6 Predicative Higher-Order Propositional Logic

Fix sets of *proof variables* and *term variables*.

The syntax of the system is given by the following grammar.

Proof	$\delta ::= p \mid \delta\delta \mid \lambda p : \phi. \delta$
Term	$M, \phi ::= x \mid \perp \mid MM \mid \lambda x : A. M \mid \phi \rightarrow \phi$
Type	$A ::= \Omega \mid A \rightarrow A$
Term Context	$\Gamma ::= \langle \rangle \mid \Gamma, x : A$
Proof Context	$\Delta ::= \langle \rangle \mid \Delta, p : \phi$
Judgement	$\mathcal{J} ::= \Gamma \text{ valid} \mid \Gamma \vdash M : A \mid \Gamma, \Delta \text{ valid} \mid \Gamma, \Delta \vdash \delta : \phi$

where p ranges over proof variables and x ranges over term variables. The variable p is bound within δ in the proof $\lambda p : \phi. \delta$, and the variable x is bound within M in the term $\lambda x : A. M$. We identify proofs and terms up to α -conversion.

In the implementation, we write **Term**(V) for the set of all terms with free variables a subset of V , where $V : \mathbf{FinSet}$.

```

data PHOPLVarKind : Set where
  -Proof : PHOPLVarKind
  -Term : PHOPLVarKind

```

```

data PHOPLNonVarKind : Set where
  -Type : PHOPLNonVarKind

```

```

PHOPLTaxonomy : Taxonomy
PHOPLTaxonomy = record {
  VarKind = PHOPLVarKind;
  NonVarKind = PHOPLNonVarKind }

```

```

module PHOPLGrammar where
open Taxonomy PHOPLTaxonomy

```

```

data PHOPLcon : ∀ {K : ExpressionKind} → ConstructorKind K → Set where

```

```

    -appProof : PHOPLcon ( $\Pi_2$  (out (varKind -Proof)) ( $\Pi_2$  (out (varKind -Proof)) (out2 {K =
    -lamProof : PHOPLcon ( $\Pi_2$  (out (varKind -Term)) ( $\Pi_2$  ( $\Pi$  (varKind -Proof) (out (varKind
    -bot : PHOPLcon (out2 {K = varKind -Term})
    -imp : PHOPLcon ( $\Pi_2$  (out (varKind -Term)) ( $\Pi_2$  (out (varKind -Term)) (out2 {K = varKin
    -appTerm : PHOPLcon ( $\Pi_2$  (out (varKind -Term)) ( $\Pi_2$  (out (varKind -Term)) (out2 {K = va
    -lamTerm : PHOPLcon ( $\Pi_2$  (out (nonVarKind -Type)) ( $\Pi_2$  ( $\Pi$  (varKind -Term) (out (varKind
    -Omega : PHOPLcon (out2 {K = nonVarKind -Type})
    -func : PHOPLcon ( $\Pi_2$  (out (nonVarKind -Type)) ( $\Pi_2$  (out (nonVarKind -Type)) (out2 {K

PHOPLparent : PHOPLVarKind → ExpressionKind
PHOPLparent -Proof = varKind -Term
PHOPLparent -Term = nonVarKind -Type

PHOPL : Grammar
PHOPL = record {
  taxonomy = PHOPLTaxonomy;
  toGrammar = record {
    Constructor = PHOPLcon;
    parent = PHOPLparent } }

module PHOPL where
  open PHOPLGrammar using (PHOPLcon;-appProof;-lamProof;-bot;-imp;-appTerm;-lamTerm;-Ome
  open Grammar.Grammar PHOPLGrammar.PHOPL

Type : Set
Type = Expression''  $\emptyset$  (nonVarKind -Type)

liftType :  $\forall \{V\} \rightarrow \text{Type} \rightarrow \text{Expression'' } V \text{ (nonVarKind -Type)}$ 
liftType (app -Omega out2) = app -Omega out2
liftType (app -func (app2 (out A) (app2 (out B) out2))) = app -func (app2 (out (liftTyp

 $\Omega$  : Type
 $\Omega$  = app -Omega out2

infix 75  $\Rightarrow$  _
 $\Rightarrow$  _ : Type → Type → Type
 $\varphi \Rightarrow \psi$  = app -func (app2 (out  $\varphi$ ) (app2 (out  $\psi$ ) out2))

VAlphabet : FinSet → Alphabet
VAlphabet  $\emptyset$  =  $\emptyset$ 
VAlphabet (Lift X) = VAlphabet X , -Term

inVar :  $\forall \{V\} \rightarrow \text{El } V \rightarrow \text{Var (VAlphabet } V) \text{ -Term}$ 
inVar Prelims. $\perp$  = x0
inVar ( $\uparrow$  x) =  $\uparrow$  (inVar x)

```

```

lowerType : ∀ {V} → Expression'' (VAlphabet V) (nonVarKind -Type) → Type
lowerType (app -Omega out2) = Ω
lowerType (app -func (app2 (out φ) (app2 (out ψ) out2))) = lowerType φ ⇒ lowerType ψ

infix 80 _,_
data TContext : FinSet → Set where
  ⟨⟩ : TContext ∅
  _,_ : ∀ {V} → TContext V → Type → TContext (Lift V)

Term : FinSet → Set
Term V = Expression'' (VAlphabet V) (varKind -Term)

⊥ : ∀ {V} → Term V
⊥ = app -bot out2

appTerm : ∀ {V} → Term V → Term V → Term V
appTerm M N = app -appTerm (app2 (out M) (app2 (out N) out2))

ΛTerm : ∀ {V} → Type → Term (Lift V) → Term V
ΛTerm A M = app -lamTerm (app2 (out (liftType A)) (app2 (Λ (out M)) out2))

_⊃_ : ∀ {V} → Term V → Term V → Term V
φ ⊃ ψ = app -imp (app2 (out φ) (app2 (out ψ) out2))

PAlphabet : FinSet → Alphabet → Alphabet
PAlphabet ∅ A = A
PAlphabet (Lift P) A = PAlphabet P A , -Proof

liftVar : ∀ {A} {K} P → Var A K → Var (PAlphabet P A) K
liftVar ∅ x = x
liftVar (Lift P) x = ↑ (liftVar P x)

liftVar' : ∀ {A} P → El P → Var (PAlphabet P A) -Proof
liftVar' (Lift P) Prelims.⊥ = x0
liftVar' (Lift P) (↑ x) = ↑ (liftVar' P x)

liftExp : ∀ {V} {K} P → Expression'' (VAlphabet V) K → Expression'' (PAlphabet P (VAlphabet V)) K
liftExp P E = E ⟨ (λ _ → liftVar P) ⟩

data PContext' (V : FinSet) : FinSet → Set where
  ⟨⟩ : PContext' V ∅
  _,_ : ∀ {P} → PContext' V P → Term V → PContext' V (Lift P)

PContext : FinSet → FinSet → Set
PContext V P = Context (VAlphabet V) → Context (PAlphabet P (VAlphabet V))

```


$P\langle \rangle : \forall \{V\} \rightarrow PContext\ V\ \emptyset$
 $P\langle \rangle\ \Gamma = \Gamma$

$_P, _ : \forall \{V\} \{P\} \rightarrow PContext\ V\ P \rightarrow Term\ V \rightarrow PContext\ V\ (Lift\ P)$
 $_P, _ \{V\} \{P\} \Delta\ \varphi\ \Gamma = _,_ \{K = \text{-Proof}\} (\Delta\ \Gamma) (liftExp\ P\ \varphi)$

$Proof : FinSet \rightarrow FinSet \rightarrow Set$
 $Proof\ V\ P = Expression''\ (PAlphabet\ P\ (VAlphabet\ V))\ (varKind\ \text{-Proof})$

$varP : \forall \{V\} \{P\} \rightarrow El\ P \rightarrow Proof\ V\ P$
 $varP\ \{P = P\}\ x = var\ (liftVar'\ P\ x)$

$appP : \forall \{V\} \{P\} \rightarrow Proof\ V\ P \rightarrow Proof\ V\ P \rightarrow Proof\ V\ P$
 $appP\ \delta\ \epsilon = app\ \text{-appProof}\ (app_2\ (out\ \delta)\ (app_2\ (out\ \epsilon)\ out_2))$

$\Lambda P : \forall \{V\} \{P\} \rightarrow Term\ V \rightarrow Proof\ V\ (Lift\ P) \rightarrow Proof\ V\ P$
 $\Lambda P\ \{P = P\}\ \varphi\ \delta = app\ \text{-lamProof}\ (app_2\ (out\ (liftExp\ P\ \varphi))\ (app_2\ (\Lambda\ (out\ \delta))\ out_2))$

$typeof' : \forall \{V\} \rightarrow El\ V \rightarrow TContext\ V \rightarrow Type$
 $typeof'\ Prelims.\perp\ (_, A) = A$
 $typeof'\ (\uparrow x)\ (\Gamma, _) = typeof'\ x\ \Gamma$

$propof : \forall \{V\} \{P\} \rightarrow El\ P \rightarrow PContext'\ V\ P \rightarrow Term\ V$
 $propof\ Prelims.\perp\ (_, \varphi) = \varphi$
 $propof\ (\uparrow x)\ (\Gamma, _) = propof\ x\ \Gamma$

$data\ \beta : Reduction\ PHOPLGrammar.PHOPL\ where$

$\beta I : \forall \{V\}\ A\ (M : Term\ (Lift\ V))\ N \rightarrow \beta\ \text{-appTerm}\ (app_2\ (out\ (\Lambda Term\ A\ M))\ (app_2\ (out\ M\ out_2)))$

The rules of deduction of the system are as follows.

$$\begin{array}{c}
\frac{}{\langle \rangle\ \text{valid}} \quad \frac{\Gamma\ \text{valid}}{\Gamma, x : A\ \text{valid}} \quad \frac{\Gamma \vdash \phi : \Omega}{\Gamma, p : \phi\ \text{valid}} \\
\\
\frac{\Gamma\ \text{valid}}{\Gamma \vdash x : A} (x : A \in \Gamma) \quad \frac{\Gamma\ \text{valid}}{\Gamma \vdash p : \phi} (p : \phi \in \Gamma) \\
\\
\frac{\Gamma\ \text{valid}}{\Gamma \vdash \perp : \Omega} \quad \frac{\Gamma \vdash \phi : \Omega \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \phi \rightarrow \psi : \Omega} \\
\\
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \quad \frac{\Gamma \vdash \delta : \phi \rightarrow \psi \quad \Gamma \vdash \epsilon : \phi}{\Gamma \vdash \delta \epsilon : \psi} \\
\\
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : A \rightarrow B} \quad \frac{\Gamma, p : \phi \vdash \delta : \psi}{\Gamma \vdash \lambda p : \phi. \delta : \phi \rightarrow \psi}
\end{array}$$

$$\frac{\Gamma \vdash \delta : \phi \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \delta : \psi} (\phi \simeq \psi)$$

mutual

```
data Tvalid : ∀ {V} → TContext V → Set1 where
  ⟨⟩ : Tvalid ⟨⟩
  _,_ : ∀ {V} {Γ : TContext V} → Tvalid Γ → ∀ A → Tvalid (Γ , A)
```

infix 10 _|-:_

```
data _|-:_ : ∀ {V} → TContext V → Term V → Type → Set1 where
  var : ∀ {V} {Γ : TContext V} {x} → Tvalid Γ → Γ ⊢ var (inVar x) : typeof' x Γ
  ⊥R : ∀ {V} {Γ : TContext V} → Tvalid Γ → Γ ⊢ ⊥ : Ω
  imp : ∀ {V} {Γ : TContext V} {φ} {ψ} → Γ ⊢ φ : Ω → Γ ⊢ ψ : Ω → Γ ⊢ φ ⊃ ψ : Ω
  app : ∀ {V} {Γ : TContext V} {M} {N} {A} {B} → Γ ⊢ M : A ⇒ B → Γ ⊢ N : A → Γ ⊢
  Λ : ∀ {V} {Γ : TContext V} {A} {M} {B} → Γ , A ⊢ M : B → Γ ⊢ ΛTerm A M : A ⇒ B
```

data Pvalid : ∀ {V} {P} → TContext V → PContext' V P → Set₁ where

```
⟨⟩ : ∀ {V} {Γ : TContext V} → Tvalid Γ → Pvalid Γ ⟨⟩
_,_ : ∀ {V} {P} {Γ : TContext V} {Δ : PContext' V P} {φ : Term V} → Pvalid Γ Δ → Γ
```

infix 10 _,_,_|-::_

```
data _,_,_|-::_ : ∀ {V} {P} → TContext V → PContext' V P → Proof V P → Term V → Set1
var : ∀ {V} {P} {Γ : TContext V} {Δ : PContext' V P} {p} → Pvalid Γ Δ → Γ , Δ ⊢ v
app : ∀ {V} {P} {Γ : TContext V} {Δ : PContext' V P} {δ} {ε} {φ} {ψ} → Γ , Δ ⊢ δ ::
Λ : ∀ {V} {P} {Γ : TContext V} {Δ : PContext' V P} {φ} {δ} {ψ} → Γ , Δ , φ ⊢ δ :: ψ
convR : ∀ {V} {P} {Γ : TContext V} {Δ : PContext' V P} {δ} {φ} {ψ} → Γ , Δ ⊢ δ :: φ
```