

# Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

March 11, 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

```
infix 75 _∘_
_∘_ : ∀ {A B C : Set} → (B → C) → (A → B) → A → C
(g ∘ 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')

```

## 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)

lift : ∀ {A} {B} → (El A → El B) → El (Lift A) → El (Lift B)
lift _ ⊥ = ⊥
lift f (↑ x) = ↑ (f x)

```

### 3 Grammars

```

module Grammar where

```

```

open import Prelims

```

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*.

```

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

data ExpressionKind : Set where

```

```

varKind : VarKind → ExpressionKind
nonVarKind : NonVarKind → ExpressionKind

data KindClass : Set where
  -Expression : KindClass
  -Abstraction : KindClass
  -Constructor : ExpressionKind → KindClass

data Kind : KindClass → Set where
  base : ExpressionKind → Kind -Expression
  out : ExpressionKind → Kind -Abstraction
  Π : VarKind → Kind -Abstraction → Kind -Abstraction
  out2 : ∀ {K} → Kind (-Constructor K)
  Π2 : ∀ {K} → Kind -Abstraction → Kind (-Constructor K) → Kind (-Constructor K)

```

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
  ∅ : 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

extend : Alphabet → VarKind → FinSet → Alphabet
extend A K ∅ = A
extend A K (Lift F) = extend A K F , K

embed : ∀ {A} {K} {F} → El F → Var (extend A K F) K
embed ⊥ = x0
embed (↑ x) = ↑ (embed x)

record ToGrammar (T : Taxonomy) : Set1 where
  open Taxonomy T
  field
    Constructor : ∀ {K : ExpressionKind} → Kind (-Constructor K) → Set

```

```

parent      : VarKind → ExpressionKind

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

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

Body' : Alphabet → ∀ K → Kind (-Constructor K) → Set
Body' V K C = Subexpression V (-Constructor K) C

Abstraction' : Alphabet → Kind -Abstraction → Set
Abstraction' V K = Subexpression V -Abstraction K

```

Given alphabets  $U$ ,  $V$ , and a function  $\rho$  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

embedl : ∀ {A} {K} {F} → Rep A (extend A K F)
embedl {F = ∅} _ x = x
embedl {F = Lift F} K x = ↑ (embedl {F = F} 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.

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

$\text{mutual}$   
 $\text{infix } 60 \ \_ \langle \_ \rangle$   
 $\_ \langle \_ \rangle : \forall \{U\} \{V\} \{K\} \rightarrow \text{Expression } U \ K \rightarrow \text{Rep } U \ V \rightarrow \text{Expression } V \ K$   
 $\text{var } x \ \langle \ \rho \ \rangle = \text{var } (\rho \ x)$   
 $(\text{app } c \ EE) \ \langle \ \rho \ \rangle = \text{app } c \ (EE \ \langle \ \rho \ \rangle B)$   
  
 $\text{infix } 60 \ \_ \langle \_ \rangle B$   
 $\_ \langle \_ \rangle B : \forall \{U\} \{V\} \{K\} \{C : \text{Kind } (-\text{Constructor } K)\} \rightarrow \text{Subexpression } U \ (-\text{Constructor } K)$   
 $\text{out}_2 \ \langle \ \rho \ \rangle B = \text{out}_2$   
 $(\text{app}_2 \ A \ EE) \ \langle \ \rho \ \rangle B = \text{app}_2 \ (A \ \langle \ \rho \ \rangle A) \ (EE \ \langle \ \rho \ \rangle B)$   
  
 $\text{infix } 60 \ \_ \langle \_ \rangle A$   
 $\_ \langle \_ \rangle A : \forall \{U\} \{V\} \{A\} \rightarrow \text{Subexpression } U \ -\text{Abstraction } A \rightarrow \text{Rep } U \ V \rightarrow \text{Subexpression } V \ A$   
 $\text{out } E \ \langle \ \rho \ \rangle A = \text{out } (E \ \langle \ \rho \ \rangle)$   
 $\Lambda \ A \ \langle \ \rho \ \rangle A = \Lambda \ (A \ \langle \ \text{Rep}\uparrow \ \rho \ \rangle A)$

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

```

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 : Kind (-Constructor K)} {EE : Subexpression 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 : Subexpression U -Abstraction A} {ρ ρ' : Rep U V} → ρ ~ ρ' ⇒
rep-wdA {U} {V} {out K} {out E} ρ-is-ρ' = wd out (rep-wd ρ-is-ρ')
rep-wdA {U} {V} .{Π _ _} {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 : Kind (-Constructor K)} {EE : Subexpression V (-Constructor K)}
rep-idB {EE = out2} = ref
rep-idB {EE = app2 _ _} = wd2 app2 rep-idA rep-idB

rep-idA : ∀ {V} {K} {A : Subexpression 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-comp ρ ρ') ≡
rep-comp {E = var _} = ref
rep-comp {E = app c _} = wd (app c) rep-compB

rep-compB : ∀ {U} {V} {W} {K} {C : Kind (-Constructor K)} {ρ : Rep U V} {ρ' : Rep V W}
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 : Subexpression U -Abstraction A}
rep-compA {A = out _} = wd out rep-comp
rep-compA {U} {V} {W} .{Π K L} {ρ} {ρ'} {Λ {K} {L} A} = wd Λ (trans (rep-wdA Rep↑-comp ρ ρ') rep-compA)

```

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

```

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

```

A *substitution*  $\sigma$  from alphabet  $U$  to alphabet  $V$ ,  $\sigma : U \Rightarrow V$ , is a function  $\sigma$  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  $\sigma$  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.

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

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

**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
  ∴ liftE (rep (σ L x) ρ)
  ≡ rep (σ L x) (λ _ x → ↑ (ρ _ x)) [[ rep-comp {E = σ L x} ]]
  ≡ rep (liftE (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  $\sigma$  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 : Kind (-Constructor K)} → Subexpression U (-Constructor K)
  out2 [_ σ]B = out2
  (app2 A EE) [_ σ]B = app2 (A [_ σ]A) (EE [_ σ]B)

  infix 60 _[_]A
  _[_]A : ∀ {U} {V} {A} → Subexpression U -Abstraction A → Sub U V → Subexpression V A
  (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 : Kind (-Constructor K)} {EE : Subexpression 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 : Subexpression U -Abstraction K} {σ σ' : Sub U V} → σ ~
  sub-wdA {A = out E} σ-is-σ' = wd out (sub-wd {E = E} σ-is-σ')
  sub-wdA {U} {V} .{Π 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 :  $\forall \{V\} \{K\} \{C : \text{Kind } (-\text{Constructor } K)\} \{EE : \text{Subexpression } V \ (-\text{Constructor } K)\}$   
subidB  $\{EE = \text{out}_2 \} = \text{ref}$   
subidB  $\{EE = \text{app}_2 \_ \_ \} = \text{wd}_2 \text{ app}_2 \text{ subidA subidB}$

subidA :  $\forall \{V\} \{K\} \{A : \text{Subexpression } V \text{-Abstraction } K\} \rightarrow A \llbracket \text{idSub} \rrbracket A \equiv A$   
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-comp<sub>1</sub> :  $\forall \{U\} \{V\} \{W\} \{K\} \{E : \text{Expression } U \ K\} \{\rho : \text{Rep } V \ W\} \{\sigma : \text{Sub } U \ V\} \rightarrow$   
 $E \llbracket \rho \bullet_1 \sigma \rrbracket \equiv \text{rep } (E \llbracket \sigma \rrbracket) \rho$   
sub-comp<sub>1</sub>  $\{E = \text{var } \_ \} = \text{ref}$   
sub-comp<sub>1</sub>  $\{E = \text{app } c \ \_ \} = \text{wd } (\text{app } c) \text{ sub-comp}_1\text{B}$

sub-comp<sub>1</sub>B :  $\forall \{U\} \{V\} \{W\} \{K\} \{C : \text{Kind } (-\text{Constructor } K)\} \{EE : \text{Subexpression } U \ (-\text{Constructor } K)\}$   
 $EE \llbracket \rho \bullet_1 \sigma \rrbracket B \equiv \text{rep } (EE \llbracket \sigma \rrbracket B) \rho$   
sub-comp<sub>1</sub>B  $\{EE = \text{out}_2 \} = \text{ref}$   
sub-comp<sub>1</sub>B  $\{U\} \{V\} \{W\} \{K\} \{(\Pi_2 \ L \ C)\} \{\text{app}_2 \ A \ EE\} = \text{wd}_2 \text{ app}_2 \text{ sub-comp}_1\text{A sub-comp}_1\text{B}$

sub-comp<sub>1</sub>A :  $\forall \{U\} \{V\} \{W\} \{K\} \{A : \text{Subexpression } U \text{-Abstraction } K\} \{\rho : \text{Rep } V \ W\} \{\sigma : \text{Sub } U \ V\} \rightarrow$   
 $A \llbracket \rho \bullet_1 \sigma \rrbracket A \equiv \text{rep } (A \llbracket \sigma \rrbracket A) \rho$   
sub-comp<sub>1</sub>A  $\{A = \text{out } E\} = \text{wd out } (\text{sub-comp}_1 \{E = E\})$   
sub-comp<sub>1</sub>A  $\{U\} \{V\} \{W\} \{(\Pi \ K \ L)\} \{\Lambda \ \{K\} \{L\} \ A\} = \text{wd } \Lambda \ (\text{trans } (\text{sub-wdA Sub}\uparrow\text{-comp}_1) \text{ sub-comp}_1\text{A})$

mutual

sub-comp<sub>2</sub> :  $\forall \{U\} \{V\} \{W\} \{K\} \{E : \text{Expression } U \ K\} \{\sigma : \text{Sub } V \ W\} \{\rho : \text{Rep } U \ V\} \rightarrow E \llbracket \rho \bullet_2 \sigma \rrbracket$   
sub-comp<sub>2</sub>  $\{E = \text{var } \_ \} = \text{ref}$   
sub-comp<sub>2</sub>  $\{U\} \{V\} \{W\} \{K\} \{\text{app } c \ EE\} = \text{wd } (\text{app } c) \text{ sub-comp}_2\text{B}$

sub-comp<sub>2</sub>B :  $\forall \{U\} \{V\} \{W\} \{K\} \{C : \text{Kind } (-\text{Constructor } K)\} \{EE : \text{Subexpression } U \ (-\text{Constructor } K)\}$   
 $\{\sigma : \text{Sub } V \ W\} \{\rho : \text{Rep } U \ V\} \rightarrow EE \llbracket \sigma \bullet_2 \rho \rrbracket B \equiv (\text{rep } EE \rho) \llbracket \sigma \rrbracket B$   
sub-comp<sub>2</sub>B  $\{EE = \text{out}_2 \} = \text{ref}$   
sub-comp<sub>2</sub>B  $\{U\} \{V\} \{W\} \{K\} \{(\Pi_2 \ L \ C)\} \{\text{app}_2 \ A \ EE\} = \text{wd}_2 \text{ app}_2 \text{ sub-comp}_2\text{A sub-comp}_2\text{B}$

sub-comp<sub>2</sub>A :  $\forall \{U\} \{V\} \{W\} \{K\} \{A : \text{Subexpression } U \text{-Abstraction } K\} \{\sigma : \text{Sub } V \ W\} \{\rho : \text{Rep } U \ V\} \rightarrow$   
 $A \llbracket \rho \bullet_2 \sigma \rrbracket A \equiv \text{rep } (A \llbracket \sigma \rrbracket A) \rho$   
sub-comp<sub>2</sub>A  $\{A = \text{out } E\} = \text{wd out } (\text{sub-comp}_2 \{E = E\})$   
sub-comp<sub>2</sub>A  $\{U\} \{V\} \{W\} \{(\Pi \ K \ L)\} \{\Lambda \ \{K\} \{L\} \ A\} = \text{wd } \Lambda \ (\text{trans } (\text{sub-wdA Sub}\uparrow\text{-comp}_2) \text{ sub-comp}_2\text{A})$

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
    ∴ liftE ((ρ L x) [[ σ ]])
    ≡ ρ L x [[ (λ _ → ↑) •1 σ ]] [[ sub-comp1 {E = ρ L x} ]]
    ≡ (liftE (ρ L x)) [[ Sub↑ σ ]] [ sub-comp2 {E = ρ L x} ]

```

mutual

```

sub-compA : ∀ {U} {V} {W} {K} {A : Subexpression U -Abstraction K} {σ : Sub V W} {ρ}
  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 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 : Kind (-Constructor K)} {EE : Subexpression U (-Constructor 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  $\rho$  be a replacement  $U \rightarrow V$ .*

1. *The replacement  $(\rho, K)$  and the substitution  $(\rho, K)$  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

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

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

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

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 (K : VarKind) : Alphabet → Set where  
 ⟨⟩ : Context K ∅  
 _,_ : ∀ {V} → Context K V → Expression V (parent K) → Context K (V , K)`

`typeof : ∀ {V} {K} (x : Var V K) (Γ : Context K V) → Expression V (parent K)  
 typeof x0 (_, A) = liftE A  
 typeof (↑ x) (Γ , _) = liftE (typeof x Γ)`

`data Context' (A : Alphabet) (K : VarKind) : FinSet → Set where  
 ⟨⟩ : Context' A K ∅  
 _,_ : ∀ {F} → Context' A K F → Expression (extend A K F) (parent K) → Context' A K F`

`typeof' : ∀ {A} {K} {F} → El F → Context' A K F → Expression (extend A K F) (parent  
 typeof' ⊥ (_, A) = liftE A  
 typeof' (↑ x) (Γ , _) = liftE (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	$\delta$	$::=$	$p \mid \delta\delta \mid \lambda p : \phi. \delta$
Proposition	$f$	$::=$	$\perp \mid \phi \rightarrow \phi$
Context	$\Gamma$	$::=$	$\langle \rangle \mid \Gamma, p : \phi$
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  $\delta$  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  $\alpha$ -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 :  $\forall \{K : \text{ExpressionKind}\} \rightarrow \text{Kind} \rightarrow \text{Set} \rightarrow \text{Set}$  where
  app : PLCon ( $\Pi_2$  (out (varKind -Proof)) ( $\Pi_2$  (out (varKind -Proof)) (out2 {K = varKind})))
  lam : PLCon ( $\Pi_2$  (out (nonVarKind -Prp)) ( $\Pi_2$  ( $\Pi$  -Proof (out (varKind -Proof))) (out2 {K = varKind})))
  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)

 $\perp$ P : Prp
 $\perp$ P = app bot out2

 $\_ \Rightarrow \_$  :  $\forall \{P\} \rightarrow \text{Expression } P \text{ (nonVarKind -Prp)} \rightarrow \text{Expression } P \text{ (nonVarKind -Prp)} \rightarrow \text{Expression } P \text{ (nonVarKind -Prp)}$ 
 $\varphi \Rightarrow \psi = \text{app imp (app}_2 \text{ (out } \varphi) \text{ (app}_2 \text{ (out } \psi) \text{ out}_2))}$ 

Proof : Alphabet  $\rightarrow$  Set
Proof P = Expression P (varKind -Proof)

appP :  $\forall \{P\} \rightarrow \text{Expression } P \text{ (varKind -Proof)} \rightarrow \text{Expression } P \text{ (varKind -Proof)} \rightarrow \text{Expression } P \text{ (varKind -Proof)}$ 
appP  $\delta \ \varepsilon = \text{app app (app}_2 \text{ (out } \delta) \text{ (app}_2 \text{ (out } \varepsilon) \text{ out}_2))}$ 

 $\Lambda$ P :  $\forall \{P\} \rightarrow \text{Expression } P \text{ (nonVarKind -Prp)} \rightarrow \text{Expression } (P, -\text{Proof}) \text{ (varKind -Proof)}$ 
 $\Lambda$ P  $\varphi \ \delta = \text{app lam (app}_2 \text{ (out } \varphi) \text{ (app}_2 \text{ (} \Lambda \text{ (out } \delta) \text{)) out}_2)$ 

data  $\beta$  : Reduction where
   $\beta$ I :  $\forall \{V\} \{\varphi\} \{\delta\} \{\varepsilon\} \rightarrow \beta \{V\} \text{app (app}_2 \text{ (out (} \Lambda$ P  $\varphi \ \delta)) \text{ (app}_2 \text{ (out } \varepsilon) \text{ out}_2))} \text{ (} \delta \llbracket x_0 :=$ 

 $\beta$ -respects-rep : respect-rep  $\beta$ 
 $\beta$ -respects-rep  $\{U\} \{V\} \{\rho = \rho\} (\beta$ I  $\{U\} \{\varphi\} \{\delta\} \{\varepsilon\}) = \text{subst } (\beta \text{ app } \_)$ 
  (let open Equational-Reasoning (Expression V (varKind -Proof)) in
     $\therefore (\text{rep } \delta \text{ (Rep}^\uparrow \rho)) \llbracket x_0 := (\text{rep } \varepsilon \rho) \rrbracket$ 
     $\equiv \delta \llbracket x_0 := (\text{rep } \varepsilon \rho) \bullet_2 \text{Rep}^\uparrow \rho \rrbracket \llbracket \text{sub-comp}_2 \{E = \delta\} \rrbracket$ 
     $\equiv \delta \llbracket \rho \bullet_1 x_0 := \varepsilon \rrbracket \llbracket \text{sub-wd } \{E = \delta\} \text{comp}_1\text{-botsub} \rrbracket$ 
     $\equiv \text{rep } (\delta \llbracket x_0 := \varepsilon \rrbracket) \rho \llbracket \text{sub-comp}_1 \{E = \delta\} \rrbracket$ 
     $\beta$ I

 $\beta$ -creates-rep : create-rep  $\beta$ 
 $\beta$ -creates-rep = record {
  created = created;
  red-created = red-created;
  rep-created = rep-created } where
  created :  $\forall \{U \ V : \text{Alphabet}\} \{K\} \{C\} \{c : \text{PLCon } C\} \{EE : \text{Subexpression } U \text{ (-Constructor } K) C\}$ 
  created  $\{c = \text{app}\} \{EE = \text{app}_2 \text{ (out (var } \_) \text{ ) } \_ \} ()$ 
  created  $\{c = \text{app}\} \{EE = \text{app}_2 \text{ (out (app app } \_) \text{ ) } \_ \} ()$ 
  created  $\{c = \text{app}\} \{EE = \text{app}_2 \text{ (out (app lam (app}_2 \text{ (out } \varphi) \text{ (app}_2 \text{ (} \Lambda \text{ (out } \delta) \text{)) out}_2)) \text{ ) } \_ \} ()$ 
  created  $\{c = \text{lam}\} ()$ 
  created  $\{c = \text{bot}\} ()$ 
  created  $\{c = \text{imp}\} ()$ 
  red-created :  $\forall \{U\} \{V\} \{K\} \{C\} \{c : \text{PLCon } C\} \{EE : \text{Subexpression } U \text{ (-Constructor } K) C\}$ 
  red-created  $\{c = \text{app}\} \{EE = \text{app}_2 \text{ (out (var } \_) \text{ ) } \_ \} ()$ 

```

```

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))) _)} ()
red-created {c = lam} ()
red-created {c = bot} ()
red-created {c = imp} ()
rep-created : ∀ {U} {V} {K} {C} {c : PLCon C} {EE : Subexpression 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.

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

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

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

```

PContext : FinSet → Set
PContext P = Context' ∅ -Proof P

```

```

Palphabet : FinSet → Alphabet
Palphabet P = extend ∅ -Proof P

```

```

Palphabet-faithful : ∀ {P} {Q} {ρ σ : Rep (Palphabet P) (Palphabet Q)} → (∀ x → ρ -Proof x = σ -Proof x) → Palphabet-faithful {P} {Q} {ρ} {σ}
Palphabet-faithful {∅} ρ-is-σ ()
Palphabet-faithful {Lift _} ρ-is-σ x0 = ρ-is-σ ⊥
Palphabet-faithful {Lift _} {Q} {ρ} {σ} ρ-is-σ (↑ x) = Palphabet-faithful {Q} {Q} {ρ} {σ}

```

```

infix 10 _|-_::_
data _|-_::_ : ∀ {P} → PContext P → Proof (Palphabet P) → Expression (Palphabet P) (non)
  var : ∀ {P} {Γ : PContext P} {p : El P} → Γ ⊢ var (embed p) :: typeof' p Γ
  app : ∀ {P} {Γ : PContext P} {δ} {ε} {φ} {ψ} → Γ ⊢ δ :: φ ⇒ ψ → Γ ⊢ ε :: φ → Γ ⊢ app δ ε :: φ → Γ ⊢ app δ ε
  Λ : ∀ {P} {Γ : PContext P} {φ} {δ} {ψ} → (Λ, _ {K = -Proof} Γ φ) ⊢ δ :: liftE ψ → Γ ⊢ Λ φ δ

```

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



```

toRep : ∀ {P} {Q} → (El P → El Q) → Rep (Palphabet P) (Palphabet Q)
toRep {} f K ()
toRep {Lift P} f .-Proof x0 = embed (f ⊥)
toRep {Lift P} {Q} f K (↑ x) = toRep {P} {Q} (f ∘ ↑) K x

toRep-embed : ∀ {P} {Q} {f : El P → El Q} {x : El P} → toRep f -Proof (embed x) ≡ embed
toRep-embed {} {} {} {} {}
toRep-embed {Lift _} {} {} {} {} {} = ref
toRep-embed {Lift P} {Q} {f} {↑ x} = toRep-embed {P} {Q} {f ∘ ↑} {x}

toRep-comp : ∀ {P} {Q} {R} {g : El Q → El R} {f : El P → El Q} → toRep g •R toRep f ~
toRep-comp {} {} {}
toRep-comp {Lift _} {g = g} x0 = toRep-embed {f = g}
toRep-comp {Lift _} {g = g} {f = f} (↑ x) = toRep-comp {g = g} {f = f ∘ ↑} x

_::_⇒R_ : ∀ {P} {Q} → (El P → El Q) → PContext P → PContext Q → Set
ρ :: Γ ⇒R Δ = ∀ x → typeof' (ρ x) Δ ≡ rep (typeof' x Γ) (toRep ρ)

toRep-↑ : ∀ {P} → toRep {P} {Lift P} ↑ ~R (λ _ → ↑)
toRep-↑ {} = λ ()
toRep-↑ {Lift P} = Palphabet-faithful {Lift P} {Lift (Lift P)} {toRep {Lift P} {Lift (Lift P)}}

toRep-lift : ∀ {P} {Q} {f : El P → El Q} → toRep (lift f) ~R Rep↑ (toRep f)
toRep-lift x0 = ref
toRep-lift {} (↑ ())
toRep-lift {Lift _} (↑ x0) = ref
toRep-lift {Lift P} {Q} {f} (↑ (↑ x)) = trans
  (sym (toRep-comp {g = ↑} {f = f ∘ ↑} x))
  (toRep-↑ {Q} (toRep (f ∘ ↑) _ x))

↑-typed : ∀ {P} {Γ : PContext P} {φ : Expression (Palphabet P) (nonVarKind -Prp)} →
  ↑ :: Γ ⇒R (Γ , φ)
↑-typed {Lift P} ⊥ = rep-wd (λ x → sym (toRep-↑ {Lift P} x))
↑-typed {Lift P} (↑ _) = rep-wd (λ x → sym (toRep-↑ {Lift P} x))

Rep↑-typed : ∀ {P} {Q} {ρ} {Γ : PContext P} {Δ : PContext Q} {φ : Expression (Palphabet P) (nonVarKind -Prp)} →
  lift ρ :: (Γ , φ) ⇒R (Δ , rep φ (toRep ρ))
Rep↑-typed {P} {Q = Q} {ρ = ρ} {φ = φ} ρ::Γ→Δ ⊥ = let open Equational-Reasoning (Express
  ∴ rep (rep φ (toRep ρ)) (λ _ → ↑)
  ≡ rep φ (λ K x → ↑ (toRep ρ K x)) [[ rep-comp {E = φ} ]]
  ≡ rep φ (toRep (lift ρ) •R (λ _ → ↑)) [ rep-wd (λ x → trans (sym (toRep-↑ {Q} (toRep
  ≡ rep (rep φ (λ _ → ↑)) (toRep (lift ρ)) [ rep-comp {E = φ} ]
Rep↑-typed {Q = Q} {ρ = ρ} {Γ = Γ} {Δ = Δ} ρ::Γ→Δ (↑ x) = let open Equational-Reasoning (
  ∴ liftE (typeof' (ρ x) Δ)
  ≡ liftE (rep (typeof' x Γ) (toRep ρ)) [ wd liftE (ρ::Γ→Δ x) ]
  ≡ rep (typeof' x Γ) (λ K x → ↑ (toRep ρ K x)) [[ rep-comp {E = typeof' x Γ} ]]

```

$\equiv \text{rep } (\text{typeof}' \ x \ \Gamma) \ (\text{toRep } \{Q\} \uparrow \bullet R \text{ toRep } \rho)$  [[ rep-wd ( $\lambda$   
 $\equiv \text{rep } (\text{typeof}' \ x \ \Gamma) \ (\text{toRep } (\text{lift } \rho) \bullet R (\lambda \_ \rightarrow \uparrow))$  [ rep-wd (toRep-comp {g =  $\uparrow$ } {f =  $\rho$   
 $\equiv \text{rep } (\text{liftE } (\text{typeof}' \ x \ \Gamma)) \ (\text{toRep } (\text{lift } \rho))$  [ rep-comp {E = typeof' x  $\Gamma$ } ]

The replacements between contexts are closed under composition.

$\bullet R\text{-typed} : \forall \{P\} \{Q\} \{R\} \{\sigma : \text{El } Q \rightarrow \text{El } R\} \{\rho : \text{El } P \rightarrow \text{El } Q\} \{\Gamma\} \{\Delta\} \{\Theta\} \rightarrow \rho :: \Gamma \Rightarrow_R \Delta$   
 $\sigma \circ \rho :: \Gamma \Rightarrow_R \Theta$   
 $\bullet R\text{-typed } \{R = R\} \{\sigma\} \{\rho\} \{\Gamma\} \{\Delta\} \{\Theta\} \rho :: \Gamma \rightarrow \Delta \ \sigma :: \Delta \rightarrow \Theta \ x = \text{let open Equational-Reasoning (Exp$   
 $\therefore \text{typeof}' \ (\sigma \ (\rho \ x)) \ \Theta$   
 $\equiv \text{rep } (\text{typeof}' \ (\rho \ x) \ \Delta) \ (\text{toRep } \sigma)$  [  $\sigma :: \Delta \rightarrow \Theta \ (\rho \ x)$  ]  
 $\equiv \text{rep } (\text{rep } (\text{typeof}' \ x \ \Gamma) \ (\text{toRep } \rho)) \ (\text{toRep } \sigma)$  [ wd ( $\lambda \ x_1 \rightarrow \text{rep } x_1 \ (\text{toRep } \sigma)$ ) ]  
 $\equiv \text{rep } (\text{typeof}' \ x \ \Gamma) \ (\text{toRep } \sigma \bullet R \text{ toRep } \rho)$  [[ rep-comp {E = typeof' x  $\Gamma$ } ]]  
 $\equiv \text{rep } (\text{typeof}' \ x \ \Gamma) \ (\text{toRep } (\sigma \circ \rho))$  [ rep-wd (toRep-comp {g =  $\sigma$ } {f =  $\rho$ }) ]

Weakening Lemma

$\text{Weakening} : \forall \{P\} \{Q\} \{\Gamma : \text{PContext } P\} \{\Delta : \text{PContext } Q\} \{\rho\} \{\delta\} \{\psi\} \rightarrow \Gamma \vdash \delta :: \psi \rightarrow \rho :: \Delta$   
 $\text{Weakening } \{P\} \{Q\} \{\Gamma\} \{\Delta\} \{\rho\} \ (\text{var } \{p = p\}) \ \rho :: \Gamma \rightarrow \Delta = \text{subst2 } (\lambda \ x \ y \rightarrow \Delta \vdash \text{var } x :: y)$   
 $(\text{sym } (\text{toRep-embed } \{f = \rho\} \{x = p\}))$   
 $(\rho :: \Gamma \rightarrow \Delta \ p)$   
 $(\text{var } \{p = \rho \ p\})$   
 $\text{Weakening } (\text{app } \Gamma \vdash \delta :: \psi \rightarrow \psi \ \Gamma \vdash \epsilon :: \psi) \ \rho :: \Gamma \rightarrow \Delta = \text{app } (\text{Weakening } \Gamma \vdash \delta :: \psi \rightarrow \psi \ \rho :: \Gamma \rightarrow \Delta) \ (\text{Weakening } \Gamma \vdash \epsilon :: \psi)$   
 $\text{Weakening } .\{P\} \{Q\} .\{\Gamma\} \{\Delta\} \{\rho\} \ (\Lambda \{P\} \{\Gamma\} \{\psi\} \{\delta\} \{\psi\} \ \Gamma, \phi \vdash \delta :: \psi) \ \rho :: \Gamma \rightarrow \Delta = \Lambda$   
 $(\text{subst } (\lambda \ P \rightarrow (\Delta, \text{rep } \psi \ (\text{toRep } \rho)) \vdash \text{rep } \delta \ (\text{Rep} \uparrow \ (\text{toRep } \rho)) :: P)$   
 $(\text{let open Equational-Reasoning (Expression (Palphabet } Q, \text{-Proof) (nonVarKind -Prp)) in$   
 $\therefore \text{rep } (\text{rep } \psi \ (\lambda \_ \rightarrow \uparrow)) \ (\text{Rep} \uparrow \ (\text{toRep } \rho))$   
 $\equiv \text{rep } \psi \ (\lambda \_ x \rightarrow \uparrow \ (\text{toRep } \rho \_ x))$  [[ rep-comp {E =  $\psi$ } ]]  
 $\equiv \text{rep } (\text{rep } \psi \ (\text{toRep } \rho)) \ (\lambda \_ \rightarrow \uparrow)$  [ rep-comp {E =  $\psi$ } ] )  
 $(\text{subst2 } (\lambda \ x \ y \rightarrow \Delta, \text{rep } \psi \ (\text{toRep } \rho) \vdash x :: y)$   
 $(\text{rep-wd } (\text{toRep-lift } \{f = \rho\}))$   
 $(\text{rep-wd } (\text{toRep-lift } \{f = \rho\}))$   
 $(\text{Weakening } \{\text{Lift } P\} \{\text{Lift } Q\} \{\Gamma, \phi\} \{\Delta, \text{rep } \psi \ (\text{toRep } \rho)\} \{\text{lift } \rho\} \{\delta\} \{\text{liftE } \psi\}$   
 $\Gamma, \phi \vdash \delta :: \psi$   
 $\text{claim}))$  where  
 $\text{claim} : \forall (x : \text{El } (\text{Lift } P)) \rightarrow \text{typeof}' \ (\text{lift } \rho \ x) \ (\Delta, \text{rep } \psi \ (\text{toRep } \rho)) \equiv \text{rep } (\text{typeof}' \ x \ \Gamma)$   
 $\text{claim } \perp = \text{let open Equational-Reasoning (Expression (Palphabet (Lift } Q)) (\text{nonVarKind -Prp})) in$   
 $\therefore \text{liftE } (\text{rep } \psi \ (\text{toRep } \rho))$   
 $\equiv \text{rep } \psi \ ((\lambda \_ \rightarrow \uparrow) \bullet R \text{ toRep } \rho)$  [[ rep-comp ]]  
 $\equiv \text{rep } (\text{liftE } \psi) \ (\text{Rep} \uparrow \ (\text{toRep } \rho))$  [ rep-comp ]  
 $\equiv \text{rep } (\text{liftE } \psi) \ (\text{toRep } (\text{lift } \rho))$  [[ rep-wd (toRep-lift {f =  $\rho$ }) ]]  
 $\text{claim } (\uparrow x) = \text{let open Equational-Reasoning (Expression (Palphabet (Lift } Q)) (\text{nonVarKind -Prp})) in$   
 $\therefore \text{liftE } (\text{typeof}' \ (\rho \ x) \ \Delta)$   
 $\equiv \text{liftE } (\text{rep } (\text{typeof}' \ x \ \Gamma) \ (\text{toRep } \rho))$  [ wd liftE ( $\rho :: \Gamma \rightarrow \Delta \ x$ ) ]  
 $\equiv \text{rep } (\text{typeof}' \ x \ \Gamma) \ ((\lambda \_ \rightarrow \uparrow) \bullet R \text{ toRep } \rho)$  [[ rep-comp ]]  
 $\equiv \text{rep } (\text{liftE } (\text{typeof}' \ x \ \Gamma)) \ (\text{toRep } (\text{lift } \rho))$  [ trans rep-comp (sym (rep-wd (toRep-lift {f =  $\rho$ }) )) ]

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

$\_::\_ \Rightarrow \_ : \forall \{P\} \{Q\} \rightarrow \text{Sub} (\text{Palphabet } P) (\text{Palphabet } Q) \rightarrow \text{PContext } P \rightarrow \text{PContext } Q \rightarrow \text{Set}$   
 $\sigma :: \Gamma \Rightarrow \Delta = \forall x \rightarrow \Delta \vdash \sigma \_ (\text{embed } x) :: \text{typeof}' x \Gamma \llbracket \sigma \rrbracket$

$\text{Sub}\uparrow\text{-typed} : \forall \{P\} \{Q\} \{\sigma\} \{\Gamma : \text{PContext } P\} \{\Delta : \text{PContext } Q\} \{\varphi : \text{Expression} (\text{Palphabet } P)\}$   
 $\text{Sub}\uparrow\text{-typed} \{P\} \{Q\} \{\sigma\} \{\Gamma\} \{\Delta\} \{\varphi\} \sigma :: \Gamma \rightarrow \Delta \perp = \text{subst} (\lambda p \rightarrow (\Delta, \varphi \llbracket \sigma \rrbracket) \vdash \text{var } x_0 :: p)$   
 $(\text{let open Equational-Reasoning} (\text{Expression} (\text{Palphabet } Q, \text{-Proof}) (\text{nonVarKind } \text{-Prp})) \text{ in}$   
 $\therefore \text{rep } (\varphi \llbracket \sigma \rrbracket) (\lambda \_ \rightarrow \uparrow)$   
 $\equiv \varphi \llbracket (\lambda \_ \rightarrow \uparrow) \bullet_1 \sigma \rrbracket \quad [[ \text{sub-comp}_1 \{E = \varphi\} ]]$   
 $\equiv \text{rep } \varphi (\lambda \_ \rightarrow \uparrow) \llbracket \text{Sub}\uparrow \sigma \rrbracket [ \text{sub-comp}_2 \{E = \varphi\} ]]$   
 $\text{var}$   
 $\text{Sub}\uparrow\text{-typed} \{Q = Q\} \{\sigma = \sigma\} \{\Gamma = \Gamma\} \{\Delta = \Delta\} \{\varphi = \varphi\} \sigma :: \Gamma \rightarrow \Delta (\uparrow x) =$   
 $\text{subst}$   
 $(\lambda P \rightarrow \Delta, \varphi \llbracket \sigma \rrbracket \vdash \text{Sub}\uparrow \sigma \text{-Proof } (\uparrow (\text{embed } x)) :: P)$   
 $(\text{let open Equational-Reasoning} (\text{Expression} (\text{Palphabet } Q, \text{-Proof}) (\text{nonVarKind } \text{-Prp})) \text{ in}$   
 $\therefore \text{rep } (\text{typeof}' x \Gamma \llbracket \sigma \rrbracket) (\lambda \_ \rightarrow \uparrow)$   
 $\equiv \text{typeof}' x \Gamma \llbracket (\lambda \_ \rightarrow \uparrow) \bullet_1 \sigma \rrbracket \quad [[ \text{sub-comp}_1 \{E = \text{typeof}' x \Gamma\} ]]$   
 $\equiv \text{rep } (\text{typeof}' x \Gamma) (\lambda \_ \rightarrow \uparrow) \llbracket \text{Sub}\uparrow \sigma \rrbracket [ \text{sub-comp}_2 \{E = \text{typeof}' x \Gamma\} ]]$   
 $(\text{subst2 } (\lambda x y \rightarrow \Delta, \varphi \llbracket \sigma \rrbracket \vdash x :: y)$   
 $(\text{rep-wd } (\text{toRep-}\uparrow \{Q\}))$   
 $(\text{rep-wd } (\text{toRep-}\uparrow \{Q\}))$   
 $(\text{Weakening } (\sigma :: \Gamma \rightarrow \Delta x) (\uparrow\text{-typed } \{\varphi = \varphi \llbracket \sigma \rrbracket\})))$

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

Substitution Lemma

$\text{Substitution} : \forall \{P\} \{Q\} \{\Gamma : \text{PContext } P\} \{\Delta : \text{PContext } Q\} \{\delta\} \{\varphi\} \{\sigma\} \rightarrow \Gamma \vdash \delta :: \varphi \rightarrow \sigma$   
 $\text{Substitution var } \sigma :: \Gamma \rightarrow \Delta = \sigma :: \Gamma \rightarrow \Delta \_$   
 $\text{Substitution (app } \Gamma \vdash \delta :: \varphi \rightarrow \psi \Gamma \vdash \epsilon :: \varphi) \sigma :: \Gamma \rightarrow \Delta = \text{app} (\text{Substitution } \Gamma \vdash \delta :: \varphi \rightarrow \psi \sigma :: \Gamma \rightarrow \Delta) (\text{Substitution}$   
 $\text{Substitution } \{Q = Q\} \{\Delta = \Delta\} \{\sigma = \sigma\} (\Lambda \{P\} \{\Gamma\} \{\varphi\} \{\delta\} \{\psi\} \Gamma, \varphi \vdash \delta :: \psi) \sigma :: \Gamma \rightarrow \Delta = \Lambda$   
 $(\text{subst} (\lambda p \rightarrow \Delta, \varphi \llbracket \sigma \rrbracket \vdash \delta \llbracket \text{Sub}\uparrow \sigma \rrbracket :: p))$

```

(let open Equational-Reasoning (Expression (Palphabet Q , -Proof) (nonVarKind -Prp)) in
 $\therefore$  rep  $\psi$  ( $\lambda \_ \rightarrow \uparrow$ )  $\llbracket$  Sub $\uparrow$   $\sigma$   $\rrbracket$ 
 $\equiv \psi \llbracket$  Sub $\uparrow$   $\sigma \bullet_2$  ( $\lambda \_ \rightarrow \uparrow$ )  $\rrbracket$   $\llbracket$  sub-comp $_2$  {E =  $\psi$ }  $\rrbracket$ 
 $\equiv$  rep ( $\psi \llbracket \sigma \rrbracket$ ) ( $\lambda \_ \rightarrow \uparrow$ )  $\llbracket$  sub-comp $_1$  {E =  $\psi$ }  $\rrbracket$ 
(Substitution  $\Gamma, \phi \vdash \delta :: \psi$  (Sub $\uparrow$ -typed  $\sigma :: \Gamma \rightarrow \Delta$ )))

```

Subject Reduction

```

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

```

```

SR :  $\forall$  {P} { $\Gamma$  : PContext P} { $\delta \epsilon$  : Proof (Palphabet P)} { $\phi$ }  $\rightarrow \Gamma \vdash \delta :: \phi \rightarrow \delta \rightarrow \langle \beta \rangle \epsilon$ 
SR var ()
SR (app { $\epsilon$  =  $\epsilon$ } ( $\Lambda$  {P} { $\Gamma$ } { $\phi$ } { $\delta$ } { $\psi$ }  $\Gamma, \phi \vdash \delta :: \psi$ )  $\Gamma \vdash \epsilon :: \phi$ ) (redex  $\beta I$ ) =
  subst ( $\lambda P_1 \rightarrow \Gamma \vdash \delta \llbracket x_0 := \epsilon \rrbracket :: P_1$ )
  (let open Equational-Reasoning (Expression (Palphabet P) (nonVarKind -Prp)) in
 $\therefore$  rep  $\psi$  ( $\lambda \_ \rightarrow \uparrow$ )  $\llbracket x_0 := \epsilon \rrbracket$ 
 $\equiv \psi \llbracket$  idSub  $\rrbracket$   $\llbracket$  sub-comp $_2$  {E =  $\psi$ }  $\rrbracket$ 
 $\equiv \psi$   $\llbracket$  subid  $\rrbracket$ 
(Substitution  $\Gamma, \phi \vdash \delta :: \psi$  (botsub-typed  $\Gamma \vdash \epsilon :: \phi$ ))
SR (app  $\Gamma \vdash \delta :: \phi \rightarrow \psi$   $\Gamma \vdash \epsilon :: \phi$ ) (app (appl (out  $\delta \rightarrow \delta'$ ))) = app (SR  $\Gamma \vdash \delta :: \phi \rightarrow \psi$   $\delta \rightarrow \delta'$ )  $\Gamma \vdash \epsilon :: \phi$ 
SR (app  $\Gamma \vdash \delta :: \phi \rightarrow \psi$   $\Gamma \vdash \epsilon :: \phi$ ) (app (appr (appl (out  $\epsilon \rightarrow \epsilon'$ )))) = app  $\Gamma \vdash \delta :: \phi \rightarrow \psi$  (SR  $\Gamma \vdash \epsilon :: \phi$   $\epsilon \rightarrow \epsilon'$ )
SR (app  $\Gamma \vdash \delta :: \phi \rightarrow \psi$   $\Gamma \vdash \epsilon :: \phi$ ) (app (appr (appr ())))
SR ( $\Lambda$   $\Gamma \vdash \delta :: \phi$ ) (redex ())
SR {P} ( $\Lambda$   $\Gamma \vdash \delta :: \phi$ ) (app (appl (out  $\phi \rightarrow \phi'$ ))) with prop-triv-red {P}  $\phi \rightarrow \phi'$ 
... | ()
SR ( $\Lambda$   $\Gamma \vdash \delta :: \phi$ ) (app (appr (appl ( $\Lambda$  (out  $\delta \rightarrow \delta'$ )))))) =  $\Lambda$  (SR  $\Gamma \vdash \delta :: \phi$   $\delta \rightarrow \delta'$ )
SR ( $\Lambda$   $\Gamma \vdash \delta :: \phi$ ) (app (appr (appr ())))

```

We define the sets of *computable* proofs  $C_\Gamma(\phi)$  for each context  $\Gamma$  and proposition  $\phi$  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}  $\rightarrow$  PContext P  $\rightarrow$  Prp  $\rightarrow$  Proof (Palphabet P)  $\rightarrow$  Set
C  $\Gamma$  (app bot out $_2$ )  $\delta$  = ( $\Gamma \vdash \delta ::$  rep  $\perp P$  ( $\lambda \_$  ()))  $\wedge$  SN  $\beta \delta$ 
C  $\Gamma$  (app imp (app $_2$  (out  $\phi$ ) (app $_2$  (out  $\psi$ ) out $_2$ )))  $\delta$  = ( $\Gamma \vdash \delta ::$  rep ( $\phi \Rightarrow \psi$ ) ( $\lambda \_$  ()))  $\wedge$ 
  ( $\forall Q \{ \Delta : PContext Q \} \rho \epsilon \rightarrow \rho :: \Gamma \Rightarrow_R \Delta \rightarrow C \Delta \phi \epsilon \rightarrow C \Delta \psi$  (appP (rep  $\delta$  (toRep  $\rho$ ))  $\epsilon$ ))

```

```

C-typed :  $\forall$  {P} { $\Gamma$  : PContext P} { $\phi$ } { $\delta$ }  $\rightarrow C \Gamma \phi \delta \rightarrow \Gamma \vdash \delta ::$  rep  $\phi$  ( $\lambda \_$  ())

```

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

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

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

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

$\text{data Neutral } \{P\} : \text{Proof } P \rightarrow \text{Set where}$   
 $\quad \text{varNeutral} : \forall x \rightarrow \text{Neutral (var } x)$   
 $\quad \text{appNeutral} : \forall \delta \varepsilon \rightarrow \text{Neutral } \delta \rightarrow \text{Neutral (appP } \delta \ \varepsilon)$

**Lemma 7.** *If  $\delta$  is neutral and  $\delta \rightarrow_\beta \varepsilon$  then  $\varepsilon$  is neutral.*

$\text{neutral-red} : \forall \{P\} \{\delta \varepsilon : \text{Proof } P\} \rightarrow \text{Neutral } \delta \rightarrow \delta \rightarrow \langle \beta \rangle \varepsilon \rightarrow \text{Neutral } \varepsilon$   
 $\text{neutral-red (varNeutral } \_) ()$   
 $\text{neutral-red (appNeutral .(app lam (app}_2 \text{ (out } \_) \text{ (app}_2 \text{ (} \Lambda \text{ (out } \_) \text{ out}_2)) \_) () (redex } \beta \text{I$   
 $\text{neutral-red (appNeutral } \_ \ \varepsilon \text{ neutral}\delta) (\text{app (appl (out } \delta \rightarrow \delta')) = \text{appNeutral } \_ \ \varepsilon (\text{neutral-}$   
 $\text{neutral-red (appNeutral } \delta \_ \text{ neutral}\delta) (\text{app (appr (appl (out } \varepsilon \rightarrow \varepsilon')) = \text{appNeutral } \delta \_ \text{ ne}$   
 $\text{neutral-red (appNeutral } \_ \_ \_) (\text{app (appr (appr ()))$

$\text{neutral-rep} : \forall \{P\} \{Q\} \{\delta : \text{Proof } P\} \{\rho : \text{Rep } P \ Q\} \rightarrow \text{Neutral } \delta \rightarrow \text{Neutral (rep } \delta \ \rho)$   
 $\text{neutral-rep } \{\rho = \rho\} (\text{varNeutral } x) = \text{varNeutral } (\rho \text{ -Proof } x)$   
 $\text{neutral-rep } \{\rho = \rho\} (\text{appNeutral } \delta \ \varepsilon \text{ neutral}\delta) = \text{appNeutral (rep } \delta \ \rho) (\text{rep } \varepsilon \ \rho) (\text{neutral-}$

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

$\text{NeutralC-lm} : \forall \{P\} \{\delta \varepsilon : \text{Proof } P\} \{X : \text{Proof } P \rightarrow \text{Set}\} \rightarrow$   
 $\quad \text{Neutral } \delta \rightarrow$

```

(∀ 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 : PContext P} {δ : Proof (Palphabet 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 : El P → El Q} ε → SN β ε → ρ :: Γ ⇒R Δ → C Δ φ ε → C Δ ψ
  claim {Q} {Δ} {ρ} ε (SNI .ε SNε) ρ::Γ→Δ ε∈Cφ = NeutralC {Q} {Δ} {appP (rep δ (toRep
    (app (subst (λ P1 → Δ ⊢ rep δ (toRep ρ) :: P1)
      (wd2 _⇒_
        (let open Equational-Reasoning (Expression (Palphabet Q) (nonVarKind -Prp)) in
          ∴ rep (rep φ _) (toRep ρ)
            ≡ rep φ _ [[ rep-comp ]]
            ≡ rep φ _ [[ rep-wd (λ ()) ]])
        ( (let open Equational-Reasoning (Expression (Palphabet Q) (nonVarKind -Prp)) in
          ∴ rep (rep ψ _) (toRep 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 (toRep ρ)) ε (neutral-rep neutralδ))
        (NeutralC-lm {X = C Δ ψ} (neutral-rep neutralδ)
          (λ δ' δ⟨ρ⟩→δ' →
            let δ0 : Proof (Palphabet 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 (toRep 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 : PContext P} {φ} {δ} → C Γ φ δ → SN β δ
CsubSN {P} {Γ} {app bot out2} P1 = π2 P1
CsubSN {P} {Γ} {app imp (app2 (out φ) (app2 (out ψ) out2)))} {δ} P1 =
  let φ' : Expression (Palphabet P) (nonVarKind -Prp)
    φ' = rep φ (λ _ ()) in
  let Γ' : PContext (Lift P)
    Γ' = Γ , φ' in
  SNrep' {Palphabet P} {Palphabet P , -Proof} { varKind -Proof} {λ _ → ↑} β-respects-1
    (SNsubbody1 (SNsubexp (CsubSN {Γ = Γ'} {φ = φ}
      (subst (C Γ' ψ) (wd (λ x → appP x (var x0)) (rep-wd (toRep-↑ {P = P}))))
      (π2 P1 (Lift P) ↑ (var x0) (λ x → sym (rep-wd (toRep-↑ {P = P}))))
      (NeutralC {φ = φ}
        (subst (λ x → Γ' ⊢ var x0 :: x)
          (trans (sym rep-comp) (rep-wd (λ _ ())))
          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  $\delta$  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  $\alpha$ -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 :  $\forall$  {K : ExpressionKind}  $\rightarrow$  Kind (-Constructor K)  $\rightarrow$  Set where
    -appProof : PHOPLcon ( $\Pi_2$  (out (varKind -Proof)) ( $\Pi_2$  (out (varKind -Proof)) (out2 {K =
    -lamProof : PHOPLcon ( $\Pi_2$  (out (varKind -Term)) ( $\Pi_2$  ( $\Pi$  -Proof (out (varKind -Proof)))
    -bot : PHOPLcon (out2 {K = varKind -Term})
    -imp : PHOPLcon ( $\Pi_2$  (out (varKind -Term)) ( $\Pi_2$  (out (varKind -Term)) (out2 {K = varKind
    -appTerm : PHOPLcon ( $\Pi_2$  (out (varKind -Term)) ( $\Pi_2$  (out (varKind -Term)) (out2 {K = va
    -lamTerm : PHOPLcon ( $\Pi_2$  (out (nonVarKind -Type)) ( $\Pi_2$  ( $\Pi$  -Term (out (varKind -Term)))
    -Omega : PHOPLcon (out2 {K = nonVarKind -Type})
    -func : PHOPLcon ( $\Pi_2$  (out (nonVarKind -Type)) ( $\Pi_2$  (out (nonVarKind -Type)) (out2 {K

  PHOPLparent : PHOPLVarKind  $\rightarrow$  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;-Omega)
    open Grammar.Grammar PHOPLGrammar.PHOPL

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

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

```



```

 $\Omega$  = app -Omega out2

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

lowerType :  $\forall \{V\} \rightarrow$  Expression V (nonVarKind -Type) → Type
lowerType (app -Omega out2) =  $\Omega$ 
lowerType (app -func (app2 (out  $\varphi$ ) (app2 (out  $\psi$ ) out2))) = lowerType  $\varphi \Rightarrow$  lowerType  $\psi$ 

{- infix 80  $\_,\_$ 
data TContext : Alphabet → Set where
   $\langle \rangle$  : TContext  $\emptyset$ 
   $\_,\_ : \forall \{V\} \rightarrow$  TContext V → Type → TContext (V , -Term) -}

TContext : Alphabet → Set
TContext = Context -Term

Term : Alphabet → Set
Term V = Expression V (varKind -Term)

 $\perp$  :  $\forall \{V\} \rightarrow$  Term V
 $\perp$  = app -bot out2

appTerm :  $\forall \{V\} \rightarrow$  Term V → Term V → Term V
appTerm M N = app -appTerm (app2 (out M) (app2 (out N) out2))

 $\Lambda$ Term :  $\forall \{V\} \rightarrow$  Type → Term (V , -Term) → Term V
 $\Lambda$ Term A M = app -lamTerm (app2 (out (liftType A)) (app2 ( $\Lambda$  (out M)) out2))

 $\_ \supset \_$  :  $\forall \{V\} \rightarrow$  Term V → Term V → Term V
 $\varphi \supset \psi$  = app -imp (app2 (out  $\varphi$ ) (app2 (out  $\psi$ ) out2))

PAlphabet : FinSet → Alphabet → Alphabet
PAlphabet  $\emptyset$  A = A
PAlphabet (Lift P) A = PAlphabet P A , -Proof

liftVar :  $\forall \{A\} \{K\} P \rightarrow$  Var A K → Var (PAlphabet P A) K
liftVar  $\emptyset$  x = x
liftVar (Lift P) x =  $\uparrow$  (liftVar P x)

liftVar' :  $\forall \{A\} P \rightarrow$  El P → Var (PAlphabet P A) -Proof
liftVar' (Lift P) Prelims. $\perp$  = x0
liftVar' (Lift P) ( $\uparrow$  x) =  $\uparrow$  (liftVar' P x)

liftExp :  $\forall \{V\} \{K\} P \rightarrow$  Expression V K → Expression (PAlphabet P V) K

```

```

liftExp P E = E (λ _ → liftVar P)

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

PContext : Alphabet → FinSet → Set
PContext V = Context' V -Proof

P⟨⟩ : ∀ {V} → PContext V ∅
P⟨⟩ = ⟨⟩

_P,_ : ∀ {V} {P} → PContext V P → Term V → PContext V (Lift P)
_P,_ {V} {P} Δ φ = Δ , rep φ (embed1 {V} { -Proof} {P})

Proof : Alphabet → FinSet → Set
Proof V P = Expression (PAlphabet P V) (varKind -Proof)

varP : ∀ {V} {P} → El P → Proof V P
varP {P = P} x = var (liftVar' P x)

appP : ∀ {V} {P} → Proof V P → Proof V P → Proof V P
appP δ ε = app -appProof (app2 (out δ) (app2 (out ε) out2))

ΛP : ∀ {V} {P} → Term V → Proof V (Lift P) → Proof V P
ΛP {P = P} φ δ = app -lamProof (app2 (out (liftExp P φ)) (app2 (Λ (out δ)) out2))

-- typeof' : ∀ {V} → Var V -Term → TContext V → Type
-- typeof' x0 (_ , A) = A
-- typeof' (↑ x) (Γ , _) = typeof' x Γ

propof : ∀ {V} {P} → El P → PContext' V P → Term V
propof Prelims.⊥ (_ , φ) = φ
propof (↑ x) (Γ , _) = propof x Γ

data β : Reduction PHOPLGrammar.PHOPL where
  βI : ∀ {V} A (M : Term (V , -Term)) N → β -appTerm (app2 (out (ΛTerm A M)) (app2 (ou

```

The rules of deduction of the system are as follows.

$$\begin{array}{c}
\frac{}{\langle \rangle \text{ valid}} \quad \frac{\Gamma \text{ valid}}{\Gamma, x : A \text{ valid}} \quad \frac{\Gamma \vdash \phi : \Omega}{\Gamma, p : \phi \text{ valid}} \\
\\
\frac{\Gamma \text{ valid}}{\Gamma \vdash x : A} (x : A \in \Gamma) \quad \frac{\Gamma \text{ valid}}{\Gamma \vdash p : \phi} (p : \phi \in \Gamma)
\end{array}$$

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

```

infix 10 _⊢_:_
data _⊢_:_ : ∀ {V} → TContext V → Term V → Expression V (nonVarKind -Type) → Set₁ where
  var : ∀ {V} {Γ : TContext V} {x} → Γ ⊢ var x : typeof x Γ
  ⊥R : ∀ {V} {Γ : TContext V} → Γ ⊢ ⊥ : rep Ω (λ _ ())
  imp : ∀ {V} {Γ : TContext V} {φ} {ψ} → Γ ⊢ φ : rep Ω (λ _ ()) → Γ ⊢ ψ : rep Ω (λ _ ())
  app : ∀ {V} {Γ : TContext V} {M} {N} {A} {B} → Γ ⊢ M : app -func (app₂ (out A) (app₂
  Λ : ∀ {V} {Γ : TContext V} {A} {M} {B} → Γ , A ⊢ M : liftE B → Γ ⊢ app -lamTerm (ap

data Pvalid : ∀ {V} {P} → TContext V → PContext' V P → Set₁ where
  ⟨⟩ : ∀ {V} {Γ : TContext V} → 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 :: φ
  convR : ∀ {V} {P} {Γ : TContext V} {Δ : PContext' V P} {δ} {φ} {ψ} → Γ , , Δ ⊢ δ :: φ

```