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

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 _∘_
_∘_ : ∀ {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
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
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 $\emptyset$: **FinSet**, and for every A : **FinSet**, the type $A+1$: **FinSet** has one more element:

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

```
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 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

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

  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, …, B_m, and binds r_i variables in its ith argument of kind A_{ij}, producing an expression of kind C. We write this expression as

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

The subexpressions of the form $[x_{i1}, \ldots, x_{ir_i}]E_i$ shall be called *abstractions*, and the pieces of syntax of the form $(A_{i1}, \ldots, A_{ij})B_i$ 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
    Π    : ExpressionKind → Kind ExpressionKind -Abstraction → Kind ExpressionKind -Abs
    out₂ : ∀ {K} → Kind ExpressionKind (-Constructor K)
    Π₂   : ∀ {K} → Kind ExpressionKind -Abstraction → Kind ExpressionKind (-Constructor

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

ConstructorKind : ∀ {ExpressionKind} → 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    : ∀ {K : ExpressionKind} → 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 α-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)

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
  Λ   : ∀ {K} {A} → Expression' (V , K) -Abstraction A → Expression' V -Abstraction (
  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

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

```
Rep↑ : ∀ {U} {V} {K} → Rep U V → Rep (U , K) (V , K)
Rep↑ _ _ x₀ = x₀
Rep↑ ρ K (↑ x) = ↑ (ρ K x)

Rep↑-wd : ∀ {U} {V} {K} {ρ ρ' : Rep U V} → ρ ∼R ρ' → Rep↑ {K = K} ρ ∼R Rep↑ ρ'
Rep↑-wd ρ-is-ρ' x₀ = ref
Rep↑-wd ρ-is-ρ' (↑ x) = wd ↑ (ρ-is-ρ' x)

Rep↑-id : ∀ {V} {K} → Rep↑ (idRep V) ∼R idRep (V , K)
Rep↑-id x₀ = ref
Rep↑-id (↑ _) = ref

Rep↑-comp : ∀ {U} {V} {W} {K} {ρ' : Rep V W} {ρ : Rep U V} → Rep↑ {K = K} (ρ' •R ρ) ∼
Rep↑-comp x₀ = ref
Rep↑-comp (↑ _) = 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 $\mathsf{Expression}\ -\ K$ becomes a functor from the category of alphabets and replacements to the category of sets.

```
rep : ∀ {U} {V} {C} {K} → Expression' U C K → Rep U V → Expression' V C K
rep (var x) ρ = var (ρ _ x)
rep (app c EE) ρ = app c (rep EE ρ)
rep (out E) ρ = out (rep E ρ)
rep (Λ E) ρ = Λ (rep E (Rep↑ ρ))
rep out₂ _ = out₂
```

```
rep (app₂ E F) ρ = app₂ (rep E ρ) (rep F ρ)

mutual
  infix 60 _⟨_⟩
  _⟨_⟩ : ∀ {U} {V} {K} → Expression'' U K → Rep U V → Expression'' V K
  var x ⟨ ρ ⟩ = var (ρ _ x)
  (app c EE) ⟨ ρ ⟩ = app c (EE ⟨ ρ ⟩B)

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

  infix 60 _⟨_⟩A
  _⟨_⟩A : ∀ {U} {V} {A} → Expression' U -Abstraction A → Rep U V → Expression' V -Ab
  out E ⟨ ρ ⟩A = out (E ⟨ ρ ⟩)
  Λ A ⟨ ρ ⟩A = Λ (A ⟨ Rep↑ ρ ⟩A)

mutual
  rep-wd : ∀ {U} {V} {K} {E : Expression'' U K} {ρ : Rep U V} {ρ'} → ρ ∼R ρ' → 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-w

  rep-wdA : ∀ {U} {V} {A} {E : Expression' U -Abstraction A} {ρ ρ' : Rep U V} → ρ ∼R
  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} → r
```

```
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} {
rep-compB {EE = out₂} = ref
rep-compB {U} {V} {W} {K} {Π₂ L C} {ρ} {ρ'} {app₂ A EE} = wd2 app₂ rep-compA rep-comp

rep-compA : ∀ {U} {V} {W} {K} {ρ : Rep U V} {ρ' : Rep V W} {A : Expression' U -Abstr
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-w
```

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* 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.

```
Sub : Alphabet → Alphabet → Set
Sub U V = ∀ K → Var U K → Expression'' V (varKind K)

_∼_ : ∀ {U} {V} → Sub U V → Sub U V → Set
σ ∼ τ = ∀ K x → σ K x ≡ τ K x
```

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

```
idSub : ∀ {V} → Sub V V
idSub _ x = var x
```

Given $\sigma : U \to V$ 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

```
infix 75 _•₁_
_•₁_ : ∀ {U} {V} {W} → Rep V W → Sub U V → Sub U W
(ρ •₁ σ) K x = rep (σ K x) ρ

infix 75 _•₂_
_•₂_ : ∀ {U} {V} {W} → Sub V W → Rep U V → Sub U W
(σ •₂ ρ) K x = σ K (ρ K x)
```

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

```
Sub↑ : ∀ {U} {V} {K} → Sub U V → Sub (U , K) (V , K)
Sub↑ _ _ x₀ = var x₀
Sub↑ σ K (↑ x) = liftE (σ K x)

Sub↑-wd : ∀ {U} {V} {K} {σ σ' : Sub U V} → σ ∼ σ' → Sub↑ {K = K} σ ∼ Sub↑ σ'
Sub↑-wd {K = K} σ-is-σ' ._ x₀ = ref
Sub↑-wd σ-is-σ' L (↑ x) = wd liftE (σ-is-σ' L x)
```

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

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

```
Sub↑-id : ∀ {V} {K} → Sub↑ {V} {V} {K} idSub ∼ idSub
Sub↑-id {K = K} ._ x₀ = ref
Sub↑-id _ (↑ _) = ref

Sub↑-comp₁ : ∀ {U} {V} {W} {K} {ρ : Rep V W} {σ : Sub U V} → Sub↑ (ρ •₁ σ) ∼ Rep↑ ρ •
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 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-i
```

Lemma 2.

1. $M[\mathrm{id}_V] \equiv M$
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 : ∀ {V} {K} {E : Expression'' V K} → E ⟦ idSub ⟧ ≡ E
    subid {E = var _} = ref
    subid {V} {K} {app c _} = wd (app c) subidB

    subidB : ∀ {V} {K} {C : ConstructorKind K} {EE : Expression' V (-Constructor K) C} →
    subidB {EE = out₂} = ref
    subidB {EE = app₂ _ _} = wd2 app₂ subidA subidB

    subidA : ∀ {V} {K} {A : Expression' V -Abstraction K} → A ⟦ idSub ⟧A ≡ A
    subidA {A = out _} = wd out subid
    subidA {A = Λ _} = wd Λ (trans (sub-wdA Sub↑-id) subidA)

  mutual
    sub-comp₁ : ∀ {U} {V} {W} {K} {E : Expression'' U K} {ρ : Rep V W} {σ : Sub U V} →
      E ⟦ ρ •₁ σ ⟧ ≡ rep (E ⟦ σ ⟧) ρ
    sub-comp₁ {E = var _} = ref
    sub-comp₁ {E = app c _} = wd (app c) sub-comp₁B

    sub-comp₁B : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {EE : Expression' U (-Constru
      EE ⟦ ρ •₁ σ ⟧B ≡ rep (EE ⟦ σ ⟧B) ρ
    sub-comp₁B {EE = out₂} = ref
    sub-comp₁B {U} {V} {W} {K} {(Π₂ L C)} {app₂ A EE} = wd2 app₂ sub-comp₁A sub-comp₁B
```

```
  sub-comp₁A : ∀ {U} {V} {W} {K} {A : Expression' U -Abstraction K} {ρ : Rep V W} {σ :
    A ⟦ ρ •₁ σ ⟧A ≡ rep (A ⟦ σ ⟧A) ρ
  sub-comp₁A {A = out E} = wd out (sub-comp₁ {E = E})
  sub-comp₁A {U} {V} {W} .{(Π (varKind K) L)} {Λ {K} {L} A} = wd Λ (trans (sub-wdA Sub↑

mutual
  sub-comp₂ : ∀ {U} {V} {W} {K} {E : Expression'' U K} {σ : Sub V W} {ρ : Rep U V} → E
  sub-comp₂ {E = var _} = ref
  sub-comp₂ {U} {V} {W} {K} {app c EE} = wd (app c) sub-comp₂B

  sub-comp₂B : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {EE : Expression' U (-Constru
    {σ : Sub V W} {ρ : Rep U V} → EE ⟦ σ •₂ ρ ⟧B ≡ (rep EE ρ) ⟦ σ ⟧B
  sub-comp₂B {EE = out₂} = ref
  sub-comp₂B {U} {V} {W} {K} {Π₂ L C} {app₂ A EE} = wd2 app₂ sub-comp₂A sub-comp₂B

  sub-comp₂A : ∀ {U} {V} {W} {K} {A : Expression' U -Abstraction K} {σ : Sub V W} {ρ :
  sub-comp₂A {A = out E} = wd out (sub-comp₂ {E = E})
  sub-comp₂A {U} {V} {W} .{Π (varKind K) L} {Λ {K} {L} A} = wd Λ (trans (sub-wdA Sub↑-c
```

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 : Expression' U -Abstraction K} {σ : Sub V W} {ρ : S
    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 Equa
    ∵ 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 (-Construc
    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 $U \to V$.*

1. *The replacement (ρ, K) and the substitution (ρ, K) are equal.*
2. $$E\langle\rho\rangle \equiv E[\rho]$$

```
Rep↑-is-Sub↑ : ∀ {U} {V} {ρ : Rep U V} {K} → (λ L x → var (Rep↑ {K = K} ρ L x)) ∼ Su
Rep↑-is-Sub↑ K x₀ = ref
Rep↑-is-Sub↑ K₁ (↑ x) = ref

mutual
  rep-is-sub : ∀ {U} {V} {K} {E : Expression'' U K} {ρ : Rep U V} →
           E ⟨ ρ ⟩ ≡ E ⟦ (λ K x → var (ρ K x)) ⟧
  rep-is-sub {E = var _} = ref
  rep-is-sub {U} {V} {K} {app c EE} = wd (app c) rep-is-subB

  rep-is-subB : ∀ {U} {V} {K} {C : ConstructorKind K} {EE : Expression' U (-Constructo
    EE ⟨ ρ ⟩B ≡ EE ⟦ (λ K x → var (ρ K x)) ⟧B
  rep-is-subB {EE = out₂} = ref
  rep-is-subB {EE = app₂ _ _} = wd2 app₂ rep-is-subA rep-is-subB
```

```
rep-is-subA : ∀ {U} {V} {K} {A : Expression' U -Abstraction K} {ρ : Rep U V} →
  A 〈 ρ 〉A ≡ A ⟦ (λ K x → var (ρ K x)) ⟧A
rep-is-subA {A = out E} = wd out rep-is-sub
rep-is-subA {U} {V} .{Π (varKind K) L} {Λ {K} {L} A} {ρ} = wd Λ (let open Equational-
  ∵ A 〈 Rep↑ ρ 〉A
  ≡ A ⟦ (λ M x → var (Rep↑ ρ M x)) ⟧A [ rep-is-subA ]
  ≡ A ⟦ Sub↑ (λ M x → var (ρ M x)) ⟧A [ sub-wdA Rep↑-is-Sub↑ ])
```

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

```
x₀:= : ∀ {V} {K} → Expression'' V (varKind K) → Sub (V , K) V
x₀:= E _ x₀ = E
x₀:= E K₁ (↑ x) = var x
```

Lemma 6. *1.*

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

2.

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

```
comp₁-botsub : ∀ {U} {V} {K} {E : Expression'' U (varKind K)} {ρ : Rep U V} →
  ρ •₁ (x₀:= E) ∼ (x₀:= (rep E ρ)) •₂ Rep↑ ρ
comp₁-botsub _ x₀ = 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 $x_1 : A_1, \ldots, x_n : A_n$ where, for each i:

- x_i is a variable of kind K_i distinct from $x_1, \ldots, x_{i-1}$;
- 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 $\{x_1, \ldots, x_n\}$.

```
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

  typeof' : ∀ {A} {K} {F} → El F → Context' A K F → Expression'' (extend A K F) (paren
  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.

$$\begin{array}{llcl}
\text{Proof} & \delta & ::= & p \mid \delta\delta \mid \lambda p : \phi.\delta \\
\text{Proposition} & f & ::= & \bot \mid \phi \to \phi \\
\text{Context} & \Gamma & ::= & \langle\rangle \mid \Gamma, p : \phi \\
\text{Judgement} & \mathcal{J} & ::= & \Gamma \vdash \delta : \phi
\end{array}$$

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 : ∀ {K : ExpressionKind} → ConstructorKind K → Set where
    app : PLCon (Π₂ (out (varKind -Proof)) (Π₂ (out (varKind -Proof)) (out₂ {K = varKind
    lam : PLCon (Π₂ (out (nonVarKind -Prp)) (Π₂ (Π (varKind -Proof) (out (varKind -Proof)
    bot : PLCon (out₂ {K = nonVarKind -Prp})
    imp : PLCon (Π₂ (out (nonVarKind -Prp)) (Π₂ (out (nonVarKind -Prp)) (out₂ {K = nonVar

  PLparent : VarKind → 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'' ∅ (nonVarKind -Prp)

⊥P : Prp
⊥P = app bot out₂

_⇒_ : ∀ {P} → Expression'' P (nonVarKind -Prp) → Expression'' P (nonVarKind -Prp) → E
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) → Exp
appP δ ε = app app (app₂ (out δ) (app₂ (out ε) out₂))

ΛP : ∀ {P} → Expression'' P (nonVarKind -Prp) → Expression'' (P , -Proof) (varKind -Pro
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
  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)))) (ap
  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))))
  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.

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

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

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

```
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 -Pro
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 = ρ

infix 10 _⊢_::_
data _⊢_::_ : ∀ {P} → PContext P → Proof (Palphabet P) → Expression'' (Palphabet P) (n
  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
  Λ : ∀ {P} {Γ : PContext P} {φ} {δ} {ψ} → (_,_ {K = -Proof} Γ φ) ⊢ δ :: liftE ψ → Γ ⊢
```

A *replacement* ρ from a context Γ to a context Δ, $\rho : \Gamma \to \Delta$, is a replacement on the syntax such that, for every $x : \phi$ in Γ, 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 ToGrammar.x₀ = embed (f ⊥)
toRep {Lift P} {Q} f K (ToGrammar.↑ x) = toRep {P} {Q} (f ∘ ↑) K x

toRep-embed : ∀ {P} {Q} {f : El P → El Q} {x : El P} → toRep f -Proof (embed x) ≡ embe
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 (Li

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} (ToGrammar.↑ (ToGrammar.↑ 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'' (Palphabe
  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 _ x))      [[ rep-comp {E = φ} ]]
  ≡ rep φ (toRep (lift ρ) •R (λ _ → ↑))  [ rep-wd (λ x → trans (sym (toRep-↑ {Q} (toRe
  ≡ 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} ]]
  ≡ rep (typeof' x Γ) (toRep {Q} ↑ •R toRep ρ)                          [[ rep-wd (λ
  ≡ rep (typeof' x Γ) (toRep (lift ρ) •R (λ _ → ↑)) [ rep-wd (toRep-comp {g = ↑} {f = ρ
  ≡ rep (liftE (typeof' x Γ)) (toRep (lift ρ)) [ rep-comp {E = typeof' x Γ} ]
```

The replacements between contexts are closed under composition.

```
•R-typed : ∀ {P} {Q} {R} {σ : El Q → El R} {ρ : El P → El Q} {Γ} {Δ} {Θ} → ρ :: Γ ⇒R Δ
  σ ∘ ρ :: Γ ⇒R Θ
•R-typed {R = R} {σ} {ρ} {Γ} {Δ} {Θ} ρ::Γ→Δ σ::Δ→Θ x = let open Equational-Reasoning (Ex
  ∵ typeof' (σ (ρ x)) Θ
  ≡ rep (typeof' (ρ x) Δ) (toRep σ)     [ σ::Δ→Θ (ρ x) ]
  ≡ rep (rep (typeof' x Γ) (toRep ρ)) (toRep σ)        [ wd (λ x₁ → rep x₁ (toRep σ)) (
  ≡ rep (typeof' x Γ) (toRep σ •R toRep ρ)    [[ rep-comp {E = typeof' x Γ} ]]
  ≡ rep (typeof' x Γ) (toRep (σ ∘ ρ))         [ rep-wd (toRep-comp {g = σ} {f = ρ}) ]
```

Weakening Lemma

```
Weakening : ∀ {P} {Q} {Γ : PContext P} {Δ : PContext Q} {ρ} {δ} {φ} → Γ ⊢ δ :: φ → ρ ::
Weakening {P} {Q} {Γ} {Δ} {ρ} (var {p = p}) ρ::Γ→Δ = subst2 (λ x y → Δ ⊢ var x :: y)
  (sym (toRep-embed {f = ρ} {x = p}))
```

```
  (ρ::Γ→Δ p)
  (var {p = ρ p})
Weakening (app Γ⊢δ::φ→ψ Γ⊢ε::φ) ρ::Γ→Δ = app (Weakening Γ⊢δ::φ→ψ ρ::Γ→Δ) (Weakening Γ⊢ε::φ
Weakening .{P} {Q} .{Γ} {Δ} {ρ} (Λ {P} {Γ} {φ} {δ} {ψ} Γ,φ⊢δ::ψ) ρ::Γ→Δ = Λ
  (subst (λ P → (Δ , rep φ (toRep ρ)) ⊢ rep δ (Rep↑ (toRep ρ)) :: P)
  (let open Equational-Reasoning (Expression'' (Palphabet Q , -Proof) (nonVarKind -Prp))
  ∵ rep (rep ψ (λ _ → ↑)) (Rep↑ (toRep ρ))
  ≡ rep ψ (λ _ x → ↑ (toRep ρ _ x))      [[ rep-comp {E = ψ} ]]
  ≡ rep (rep ψ (toRep ρ)) (λ _ → ↑)          [ rep-comp {E = ψ} ] )
  (subst2 (λ x y → Δ , rep φ (toRep ρ) ⊢ x :: y)
    (rep-wd (toRep-lift {f = ρ}))
    (rep-wd (toRep-lift {f = ρ}))
    (Weakening {Lift P} {Lift Q} {Γ , φ} {Δ , rep φ (toRep ρ)} {lift ρ} {δ} {liftE ψ}
      Γ,φ⊢δ::ψ
      claim))) where
  claim : ∀ (x : El (Lift P)) → typeof' (lift ρ x) (Δ , rep φ (toRep ρ)) ≡ rep (typeof'
  claim ⊥ = let open Equational-Reasoning (Expression'' (Palphabet (Lift Q)) (nonVarKind
    ∵ liftE (rep φ (toRep ρ))
    ≡ rep φ ((λ _ → ↑) •R toRep ρ)        [[ rep-comp ]]
    ≡ rep (liftE φ) (Rep↑ (toRep ρ))      [ rep-comp ]
    ≡ rep (liftE φ) (toRep (lift ρ))      [[ rep-wd (toRep-lift {f = ρ}) ]]
  claim (↑ x) = let open Equational-Reasoning (Expression'' (Palphabet (Lift Q)) (nonVar
    ∵ liftE (typeof' (ρ x) Δ)
    ≡ liftE (rep (typeof' x Γ) (toRep ρ))        [ wd liftE (ρ::Γ→Δ x) ]
    ≡ rep (typeof' x Γ) ((λ _ → ↑) •R toRep ρ)   [[ rep-comp ]]
    ≡ rep (liftE (typeof' x Γ)) (toRep (lift ρ)) [ trans rep-comp (sym (rep-wd (toRep-li
```

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

```
_::_⇒_ : ∀ {P} {Q} → Sub (Palphabet P) (Palphabet Q) → PContext P → PContext Q → Set
σ :: Γ ⇒ Δ = ∀ x → Δ ⊢ σ _ (embed x) :: typeof' x Γ ⟦ σ ⟧

Sub↑-typed : ∀ {P} {Q} {σ} {Γ : PContext P} {Δ : PContext Q} {φ : Expression'' (Palphabet
Sub↑-typed {P} {Q} {σ} {Γ} {Δ} {φ} σ::Γ→Δ ⊥ = subst (λ p → (Δ , φ ⟦ σ ⟧) ⊢ var x₀ :: p)
  (let open Equational-Reasoning (Expression'' (Palphabet Q , -Proof) (nonVarKind -Prp))
  ∵ rep (φ ⟦ σ ⟧) (λ _ → ↑)
  ≡ φ ⟦ (λ _ → ↑) •₁ σ ⟧       [[ sub-comp₁ {E = φ} ]]
  ≡ rep φ (λ _ → ↑) ⟦ Sub↑ σ ⟧ [ sub-comp₂ {E = φ} ])
  var
Sub↑-typed {Q = Q} {σ = σ} {Γ = Γ} {Δ = Δ} {φ = φ} σ::Γ→Δ (↑ x) =
  subst
  (λ P → Δ , φ ⟦ σ ⟧ ⊢ Sub↑ σ -Proof (↑ (embed x)) :: P)
  (let open Equational-Reasoning (Expression'' (Palphabet Q , -Proof) (nonVarKind -Prp))
  ∵ rep (typeof' x Γ ⟦ σ ⟧) (λ _ → ↑)
  ≡ typeof' x Γ ⟦ (λ _ → ↑) •₁ σ ⟧      [[ sub-comp₁ {E = typeof' x Γ} ]]
```

```
  ≡ rep (typeof' x Γ) (λ _ → ↑) ⟦ Sub↑ σ ⟧ [ sub-comp₂ {E = typeof' x Γ} ])
  (subst2 (λ x y → Δ , φ ⟦ σ ⟧ ⊢ x :: y)
    (rep-wd (toRep-↑ {Q}))
    (rep-wd (toRep-↑ {Q}))
    (Weakening (σ::Γ→Δ x) (↑-typed {φ = φ ⟦ σ ⟧})))

botsub-typed : ∀ {P} {Γ : PContext P} {φ : Expression'' (Palphabet P) (nonVarKind -Prp)}
  Γ ⊢ δ :: φ → x₀:= δ :: (Γ , φ) ⇒ Γ
botsub-typed {P} {Γ} {φ} {δ} Γ⊢δ::φ ⊥ = subst (λ P₁ → Γ ⊢ δ :: P₁)
  (let open Equational-Reasoning (Expression'' (Palphabet P) (nonVarKind -Prp)) in
  ∵ φ
  ≡ φ ⟦ idSub ⟧                      [[ subid ]]
  ≡ rep φ (λ _ → ↑) ⟦ x₀:= δ ⟧      [ sub-comp₂ {E = φ} ])
  Γ⊢δ::φ
botsub-typed {P} {Γ} {φ} {δ} _ (↑ x) = subst (λ P₁ → Γ ⊢ var (embed x) :: P₁)
  (let open Equational-Reasoning (Expression'' (Palphabet P) (nonVarKind -Prp)) in
  ∵ typeof' x Γ
  ≡ typeof' x Γ ⟦ idSub ⟧                 [[ subid ]]
  ≡ rep (typeof' x Γ) (λ _ → ↑) ⟦ x₀:= δ ⟧ [ sub-comp₂ {E = typeof' x Γ} ])
  var
```

Substitution Lemma

```
Substitution : ∀ {P} {Q} {Γ : PContext P} {Δ : PContext Q} {δ} {φ} {σ} → Γ ⊢ δ :: φ → σ
Substitution var σ::Γ→Δ = σ::Γ→Δ _
Substitution (app Γ⊢δ::φ→ψ Γ⊢ε::φ) σ::Γ→Δ = app (Substitution Γ⊢δ::φ→ψ σ::Γ→Δ) (Substitut
Substitution {Q = Q} {Δ = Δ} {σ = σ} (Λ {P} {Γ} {φ} {δ} {ψ} Γ,φ⊢δ::ψ) σ::Γ→Δ = Λ
  (subst (λ p → Δ , φ ⟦ σ ⟧ ⊢ δ ⟦ Sub↑ σ ⟧ :: p)
  (let open Equational-Reasoning (Expression'' (Palphabet Q , -Proof) (nonVarKind -Prp))
  ∵ rep ψ (λ _ → ↑) ⟦ Sub↑ σ ⟧
  ≡ ψ ⟦ Sub↑ σ •₂ (λ _ → ↑) ⟧  [[ sub-comp₂ {E = ψ} ]]
  ≡ rep (ψ ⟦ σ ⟧) (λ _ → ↑)    [ sub-comp₁ {E = ψ} ])
  (Substitution Γ,φ⊢δ::ψ (Sub↑-typed σ::Γ→Δ)))
```

Subject Reduction

```
prop-triv-red : ∀ {P} {φ ψ : Expression'' (Palphabet P) (nonVarKind -Prp)} → φ →⟨ β ⟩ ψ
prop-triv-red {_} {app bot out₂} (redex ())
prop-triv-red {P} {app bot out₂} (app ())
prop-triv-red {P} {app imp (app₂ _ (app₂ _ out₂))} (redex ())
prop-triv-red {P} {app imp (app₂ (out φ) (app₂ ψ out₂))} (app (appl (out φ→φ'))) = prop-
prop-triv-red {P} {app imp (app₂ φ (app₂ (out ψ) out₂))} (app (appr (appl (out ψ→ψ'))))
prop-triv-red {P} {app imp (app₂ _ (app₂ (out _) out₂))} (app (appr (appr ())))

SR : ∀ {P} {Γ : PContext P} {δ ε : Proof (Palphabet P)} {φ} → Γ ⊢ δ :: φ → δ →⟨ β ⟩ ε -
SR var ()
SR (app {ε = ε} (Λ {P} {Γ} {φ} {δ} {ψ} Γ,φ⊢δ::ψ) Γ⊢ε::φ) (redex βI) =
```

```
  subst (λ P₁ → Γ ⊢ δ ⟦ x₀:= ε ⟧ :: P₁)
  (let open Equational-Reasoning (Expression'' (Palphabet P) (nonVarKind -Prp)) in
  ∵ rep ψ (λ _ → ↑) ⟦ x₀:= ε ⟧
  ≡ ψ ⟦ idSub ⟧                   [[ sub-comp₂ {E = ψ} ]]
  ≡ ψ                             [ subid ])
  (Substitution Γ,φ⊢δ::ψ (botsub-typed Γ⊢ε::φ))
SR (app Γ⊢δ::φ→ψ Γ⊢ε::φ) (app (appl (out δ→δ'))) = app (SR Γ⊢δ::φ→ψ δ→δ') Γ⊢ε::φ
SR (app Γ⊢δ::φ→ψ Γ⊢ε::φ) (app (appr (appl (out ε→ε')))) = app Γ⊢δ::φ→ψ (SR Γ⊢ε::φ ε→ε')
SR (app Γ⊢δ::φ→ψ Γ⊢ε::φ) (app (appr (appr ())))
SR (Λ Γ⊢δ::φ) (redex ())
SR {P} (Λ Γ⊢δ::φ) (app (appl (out φ→φ'))) with prop-triv-red {P} φ→φ'
... | ()
SR (Λ Γ⊢δ::φ) (app (appr (appl (Λ (out δ→δ'))))) = Λ (SR Γ⊢δ::φ δ→δ')
SR (Λ Γ⊢δ::φ) (app (appr (appr ())))
```

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

$$C_\Gamma(\bot) = \{\delta \mid \Gamma \vdash \delta : \bot, \delta \in SN\}$$
$$C_\Gamma(\phi \to \psi) = \{\delta \mid \Gamma : \delta : \phi \to \psi, \forall \epsilon \in C_\Gamma(\phi).\delta\epsilon \in C_\Gamma(\psi)\}$$

```
C : ∀ {P} → PContext P → Prp → Proof (Palphabet P) → Set
C Γ (app bot out₂) δ = (Γ ⊢ δ :: rep ⊥P (λ _ ()) ) ∧ SN β δ
C Γ (app imp (app₂ (out φ) (app₂ (out ψ) out₂))) δ = (Γ ⊢ δ :: rep (φ ⇒ ψ) (λ _ ())) ∧
  (∀ Q {Δ : PContext Q} ρ ε → ρ :: Γ ⇒R Δ → C Δ φ ε → C Δ ψ (appP (rep δ (toRep ρ)) ε)

C-typed : ∀ {P} {Γ : PContext P} {φ} {δ} → C Γ φ δ → Γ ⊢ δ :: rep φ (λ _ ())
C-typed {φ = app bot out₂} = π₁
C-typed {Γ = Γ} {φ = app imp (app₂ (out φ) (app₂ (out ψ) out₂))} {δ = δ} = λ x → subst (
  (wd2 _⇒_ (rep-wd {E = φ} (λ ())) (rep-wd {E = ψ} (λ ())))
  (π₁ x)

C-rep : ∀ {P} {Q} {Γ : PContext P} {Δ : PContext Q} {φ} {δ} {ρ} → C Γ φ δ → ρ :: Γ ⇒R Δ
C-rep {φ = app bot out₂} (Γ⊢δ::⊥ , SNδ) ρ::Γ→Δ = (Weakening Γ⊢δ::⊥ ρ::Γ→Δ) , SNrep β-crea
C-rep {P} {Q} {Γ} {Δ} {app imp (app₂ (out φ) (app₂ (out ψ) out₂))} {δ} {ρ} (Γ⊢δ::φ⇒ψ , Cδ
  (let open Equational-Reasoning (Expression'' (Palphabet Q) (nonVarKind -Prp)) in
    ∵ rep (rep φ _) (toRep ρ)
    ≡ rep φ _             [[ rep-comp ]]
    ≡ rep φ _             [ rep-wd (λ ()) ])
  (trans (sym rep-comp) (rep-wd (λ ())))) (Weakening Γ⊢δ::φ⇒ψ ρ::Γ→Δ) ,
  (λ R σ ε σ::Δ→Θ ε∈Cφ → subst (C _ ψ) (wd (λ x → appP x ε)
    (trans (sym (rep-wd (toRep-comp {g = σ} {f = ρ}))) rep-comp)) --(wd (λ x → appP x ε)
    (Cδ R (σ ∘ ρ) ε (•R-typed {σ = σ} {ρ = ρ} ρ::Γ→Δ σ::Δ→Θ) ε∈Cφ))

C-red : ∀ {P} {Γ : PContext P} {φ} {δ} {ε} → C Γ φ δ → δ →⟨ β ⟩ ε → C Γ φ ε
```

```
C-red {φ = app bot out₂} (Γ⊢δ::⊥ , SNδ) δ→ε = (SR Γ⊢δ::⊥ δ→ε) , (SNred SNδ (osr-red δ→
C-red {Γ = Γ} {φ = app imp (app₂ (out φ) (app₂ (out ψ) out₂))} {δ = δ} (Γ⊢δ::φ⇒ψ , Cδ) δ-
  (wd2 _⇒_ (rep-wd (λ ())) (rep-wd (λ ())))
  Γ⊢δ::φ⇒ψ) δ→δ') ,
  (λ Q ρ ε ρ::Γ→Δ ε∈Cφ → C-red {φ = ψ} (Cδ Q ρ ε ρ::Γ→Δ ε∈Cφ) (app (appl (out (reposr β
```

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 $\delta \to_\beta \epsilon$ 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 $\delta \to_\beta \epsilon$, 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 : 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)) i
    ∵ 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} {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'' (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-r
    (SNsubbodyl (SNsubexp (CsubSN {Γ = Γ'} {φ = ψ}
    (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	δ	$::=$	$p \mid \delta\delta \mid \lambda p : \phi.\delta$
Term	M, ϕ	$::=$	$x \mid \bot \mid MM \mid \lambda x : A.M \mid \phi \to \phi$
Type	A	$::=$	$\Omega \mid A \to A$
Term Context	Γ	$::=$	$\langle\rangle \mid \Gamma, x : A$
Proof Context	Δ	$::=$	$\langle\rangle \mid \Delta, p : \phi$
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 $\mathbf{Term}\,(V)$ 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 (Π₂ (out (varKind -Proof)) (Π₂ (out (varKind -Proof)) (out₂ {K =
```

```
    -lamProof : PHOPLcon (Π₂ (out (varKind -Term)) (Π₂ (Π (varKind -Proof) (out (varKind
    -bot : PHOPLcon (out₂ {K = varKind -Term})
    -imp : PHOPLcon (Π₂ (out (varKind -Term)) (Π₂ (out (varKind -Term)) (out₂ {K = varKin
    -appTerm : PHOPLcon (Π₂ (out (varKind -Term)) (Π₂ (out (varKind -Term)) (out₂ {K = va
    -lamTerm : PHOPLcon (Π₂ (out (nonVarKind -Type)) (Π₂ (Π (varKind -Term) (out (varKind
    -Omega : PHOPLcon (out₂ {K = nonVarKind -Type})
    -func  : PHOPLcon (Π₂ (out (nonVarKind -Type)) (Π₂ (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'' ∅ (nonVarKind -Type)

  liftType : ∀ {V} → Type → 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 (liftTyp

  Ω : Type
  Ω = app -Omega out₂

  infix 75 _⇒_
  _⇒_ : Type → Type → Type
  φ ⇒ ψ = app -func (app₂ (out φ) (app₂ (out ψ) out₂))

  lowerType : ∀ {V} → Expression'' 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 : Alphabet → Set where
    〈〉 : TContext ∅
    _,_ : ∀ {V} → TContext V → Type → TContext (V , -Term) -}
```

```
TContext : Alphabet → Set
TContext = Context -Term

Term : Alphabet → Set
Term V = Expression'' 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 (V , -Term) → 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'' 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 (embedl {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.

$$\frac{}{\langle\rangle \text{ valid}} \qquad \frac{\Gamma \text{ valid}}{\Gamma, x : A \text{ valid}} \qquad \frac{\Gamma \vdash \phi : \Omega}{\Gamma, p : \phi \text{ valid}}$$

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

$$\frac{\Gamma \text{ valid}}{\Gamma \vdash \bot : \Omega} \qquad \frac{\Gamma \vdash \phi : \Omega \quad \Gamma \vdash \psi : \Omega}{\Gamma \vdash \phi \to \psi : \Omega}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma \vdash \delta : \phi \to \psi \quad \Gamma \vdash \epsilon : \phi}{\Gamma \vdash \delta\epsilon : \psi}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma, p : \phi \vdash \delta : \psi}{\Gamma \vdash \lambda p : \phi.\delta : \phi \to \psi}$$

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

```
infix 10 _⊢_:_
data _⊢_:_ : ∀ {V} → TContext V → Term V → Expression'' V (nonVarKind -Type) → Set₁
```

```
  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 ⊢ va
  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
```