Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

May 4, 2016

1 Preliminaries

```
module Prelims where

open import Relation.Binary public hiding (_⇒_)
import Relation.Binary.EqReasoning
open import Relation.Binary.PropositionalEquality public using (_≡_;refl;sym;trans;cong;

module EqReasoning {s₁ s₂} (S : Setoid s₁ s₂) where
  open Setoid S using (_≈_)
  open Relation.Binary.EqReasoning S public

  infixr 2 _≡⟨⟨_⟩⟩_
  _≡⟨⟨_⟩⟩_ : ∀ x {y z} → y ≈ x → y ≈ z → x ≈ z
  _ ≡⟨⟨ y≈x ⟩⟩ y≈z = Setoid.trans S (Setoid.sym S y≈x) y≈z

module ≡-Reasoning {a} {A : Set a} where
  open Relation.Binary.PropositionalEquality
  open ≡-Reasoning {a} {A} public

  infixr 2 _≡⟨⟨_⟩⟩_
  _≡⟨⟨_⟩⟩_ : ∀ (x : A) {y z} → y ≡ x → y ≡ z → x ≡ z
  _ ≡⟨⟨ y≡x ⟩⟩ y≡z = trans (sym y≡x) y≡z
--TODO Add this to standard library

open import Function
open import Data.List
open import Prelims
open import Taxonomy

module Grammar where

record ToGrammar (T : Taxonomy) : Set₁ where
```

```
open Taxonomy.Taxonomy T
field
  Constructor    : ∀ {K} → Kind (-Constructor K) → Set
  parent         : VarKind → ExpressionKind

data Subexpression : Alphabet → ∀ C → Kind C → Set
Expression : Alphabet → ExpressionKind → Set
Body : Alphabet → ∀ {K} → Kind (-Constructor K) → Set

Expression V K = Subexpression V -Expression (base K)
Body V {K} C = Subexpression V (-Constructor K) C

infixr 50 _,,_
data Subexpression where
  var : ∀ {V} {K} → Var V K → Expression V (varKind K)
  app : ∀ {V} {K} {C} → Constructor C → Body V {K} C → Expression V K
  out : ∀ {V} {K} → Body V {K} out
  _,,_ : ∀ {V} {K} {A} {L} {C} → Expression (extend' V A) L → Body V {K} C → Body V

var-inj : ∀ {V} {K} {x y : Var V K} → var x ≡ var y → x ≡ y
var-inj refl = refl

record PreOpFamily : Set₂ where
  field
    Op : Alphabet → Alphabet → Set
    apV : ∀ {U} {V} {K} → Op U V → Var U K → Expression V (varKind K)
    up : ∀ {V} {K} → Op V (V , K)
    apV-up : ∀ {V} {K} {L} {x : Var V K} → apV (up {K = L}) x ≡ var (↑ x)
    idOp : ∀ V → Op V V
    apV-idOp : ∀ {V} {K} (x : Var V K) → apV (idOp V) x ≡ var x

  _∼op_ : ∀ {U} {V} → Op U V → Op U V → Set
  _∼op_ {U} {V} ρ σ = ∀ {K} (x : Var U K) → apV ρ x ≡ apV σ x

  ∼-refl : ∀ {U} {V} {σ : Op U V} → σ ∼op σ
  ∼-refl _ = refl

  ∼-sym : ∀ {U} {V} {σ τ : Op U V} → σ ∼op τ → τ ∼op σ
  ∼-sym σ-is-τ x = sym (σ-is-τ x)

  ∼-trans : ∀ {U} {V} {ρ σ τ : Op U V} → ρ ∼op σ → σ ∼op τ → ρ ∼op τ
  ∼-trans ρ-is-σ σ-is-τ x = trans (ρ-is-σ x) (σ-is-τ x)

  OP : Alphabet → Alphabet →  Setoid _ _
  OP U V = record {
    Carrier = Op U V ;
```

```
  _≈_ = _∼op_ ;
  isEquivalence = record {
    refl = ∼-refl ;
    sym = ∼-sym ;
    trans = ∼-trans } }

record Lifting : Set₁ where
  field
    liftOp : ∀ {U} {V} K → Op U V → Op (U , K) (V , K)
    liftOp-cong : ∀ {V} {W} {K} {ρ σ : Op V W} → ρ ∼op σ → liftOp K ρ ∼op liftOp K
```

Given an operation $\sigma : U \to V$ and an abstraction kind $(x_1 : A_1, \ldots, x_n : A_n)B$, define the *repeated lifting* σ^A to be $((\cdots(\sigma, A_1), A_2), \cdots), A_n)$.

```
liftOp' : ∀ {U} {V} A → Op U V → Op (extend' U A) (extend' V A)
liftOp' [] σ = σ
liftOp' (K :: A) σ = liftOp' A (liftOp K σ)

liftOp'-cong : ∀ {U} {V} A {ρ σ : Op U V} → ρ ∼op σ → liftOp' A ρ ∼op liftOp' A
liftOp'-cong [] ρ-is-σ = ρ-is-σ
liftOp'-cong (_ :: A) ρ-is-σ = liftOp'-cong A (liftOp-cong ρ-is-σ)

ap : ∀ {U} {V} {C} {K} → Op U V → Subexpression U C K → Subexpression V C K
ap ρ (var x) = apV ρ x
ap ρ (app c EE) = app c (ap ρ EE)
ap _ out = out
ap ρ (_,,_ {A = A} {L = L} E EE) = _,,_ (ap (liftOp' A ρ) E) (ap ρ EE)

ap-congl : ∀ {U} {V} {C} {K} {ρ σ : Op U V} (E : Subexpression U C K) →
  ρ ∼op σ → ap ρ E ≡ ap σ E
ap-congl (var x) ρ-is-σ = ρ-is-σ x
ap-congl (app c E) ρ-is-σ = cong (app c) (ap-congl E ρ-is-σ)
ap-congl out _ = refl
ap-congl (_,,_ {A = A} E F) ρ-is-σ = cong₂ _,,_ (ap-congl E (liftOp'-cong A ρ-is-σ)

ap-cong : ∀ {U} {V} {C} {K} {ρ σ : Op U V} {M N : Subexpression U C K} →
  ρ ∼op σ → M ≡ N → ap ρ M ≡ ap σ N
ap-cong {ρ = ρ} {σ} {M} {N} ρ∼σ M≡N = let open ≡-Reasoning in
  begin
    ap ρ M
  ≡⟨ ap-congl M ρ∼σ ⟩
    ap σ M
  ≡⟨ cong (ap σ) M≡N ⟩
    ap σ N
  □
```

```
  record IsLiftFamily : Set₁ where
    field
      liftOp-x₀ : ∀ {U} {V} {K} {σ : Op U V} → apV (liftOp K σ) x₀ ≡ var x₀
      liftOp-↑ : ∀ {U} {V} {K} {L} {σ : Op U V} (x : Var U L) →
        apV (liftOp K σ) (↑ x) ≡ ap up (apV σ x)

    liftOp-idOp : ∀ {V} {K} → liftOp K (idOp V) ∼op idOp (V , K)
    liftOp-idOp {V} {K} x₀ = let open ≡-Reasoning in
      begin
        apV (liftOp K (idOp V)) x₀
      ≡⟨ liftOp-x₀ ⟩
        var x₀
      ≡⟨⟨ apV-idOp x₀ ⟩⟩
        apV (idOp (V , K)) x₀
        □
    liftOp-idOp {V} {K} {L} (↑ x) = let open ≡-Reasoning in
      begin
        apV (liftOp K (idOp V)) (↑ x)
      ≡⟨ liftOp-↑ x ⟩
        ap up (apV (idOp V) x)
      ≡⟨ cong (ap up) (apV-idOp x) ⟩
        ap up (var x)
      ≡⟨ apV-up ⟩
        var (↑ x)
      ≡⟨⟨ apV-idOp (↑ x) ⟩⟩
        (apV (idOp (V , K)) (↑ x)
        □)

    liftOp'-idOp : ∀ {V} A → liftOp' A (idOp V) ∼op idOp (extend' V A)
    liftOp'-idOp [] = ∼-refl
    liftOp'-idOp {V} (K :: A) = let open EqReasoning (OP (extend' (V , K) A) (extend'
      begin
        liftOp' A (liftOp K (idOp V))
      ≈⟨ liftOp'-cong A liftOp-idOp ⟩
        liftOp' A (idOp (V , K))
      ≈⟨ liftOp'-idOp A ⟩
        idOp (extend' (V , K) A)
        □

    ap-idOp : ∀ {V} {C} {K} {E : Subexpression V C K} → ap (idOp V) E ≡ E
    ap-idOp {E = var x} = apV-idOp x
    ap-idOp {E = app c EE} = cong (app c) ap-idOp
    ap-idOp {E = out} = refl
    ap-idOp {E = _,,_ {A = A} E F} = cong₂ _,,_ (trans (ap-congl E (liftOp'-idOp A))

record LiftFamily : Set₂ where
```

```
    field
      preOpFamily : PreOpFamily
      lifting : PreOpFamily.Lifting preOpFamily
      isLiftFamily : PreOpFamily.Lifting.IsLiftFamily lifting
    open PreOpFamily preOpFamily public
    open Lifting lifting public
    open IsLiftFamily isLiftFamily public

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

```
module PL where
```

```
open import Function
open import Data.Empty
open import Data.Product
open import Data.Nat
open import Data.Fin
open import Data.List
open import Prelims
open import Taxonomy
open import Grammar
import Grammar.Context
import Grammar.Substitution
import Grammar.Substitution.Botsub
import Reduction
```

2 Propositional Logic

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

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

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

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 Taxonomy.Taxonomy PLtaxonomy

  data PLCon : ∀ {K : ExpressionKind} → Kind (-Constructor K) → Set where
    app : PLCon (Π [] (varKind -Proof) (Π [] (varKind -Proof) (out {K = varKind -Proof}))
    lam : PLCon (Π [] (nonVarKind -Prp) (Π [ -Proof ] (varKind -Proof) (out {K = varKind
    bot : PLCon (out {K = nonVarKind -Prp})
    imp : PLCon (Π [] (nonVarKind -Prp) (Π [] (nonVarKind -Prp) (out {K = nonVarKind -Prp

  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 Grammar.Context Propositional-Logic
open import Grammar.OpFamily Propositional-Logic
open import Grammar.Replacement Propositional-Logic
open Grammar.Substitution Propositional-Logic
open Grammar.Substitution.Botsub Propositional-Logic

Prp : Set
Prp = Expression ∅ (nonVarKind -Prp)

⊥P : Prp
⊥P = app bot out

_⇛_ : ∀ {P} → Expression P (nonVarKind -Prp) → Expression P (nonVarKind -Prp) → Expre
φ ⇛ ψ = app imp (φ ,, ψ ,, out)
```

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

appP : ∀ {P} → Expression P (varKind -Proof) → Expression P (varKind -Proof) → Express
appP δ ε = app app (δ ,, ε ,, out)

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

data β : ∀ {V} {K} {C : Kind (-Constructor K)} → Constructor C → Subexpression V (-Cons
  βI : ∀ {V} {φ} {δ} {ε} → β {V} app (ΛP φ δ ,, ε ,, out) (δ ⟦ x₀:= ε ⟧)

open Reduction Propositional-Logic β

β-respects-rep : Respects-Creates.respects' replacement
β-respects-rep {U} {V} {σ = ρ} (βI .{U} {φ} {δ} {ε}) = subst (β app _)
  (let open ≡-Reasoning {A = Expression V (varKind -Proof)} in
  begin
    δ 〈 Rep↑ -Proof ρ 〉 ⟦ x₀:= (ε 〈 ρ 〉) ⟧
  ≡⟨⟨ sub-comp₂ {E = δ} ⟩⟩
    δ ⟦ x₀:= (ε 〈 ρ 〉) •₂ Rep↑ -Proof ρ ⟧
  ≡⟨⟨ sub-cong δ comp₁-botsub ⟩⟩
    δ ⟦ ρ •₁ x₀:= ε ⟧
  ≡⟨ sub-comp₁ {E = δ} ⟩
    δ ⟦ x₀:= ε ⟧ 〈 ρ 〉
    ∎)
  βI

β-creates-rep : Respects-Creates.creates' replacement
β-creates-rep {c = app} (_,,_ (var _) _) ()
β-creates-rep {c = app} (_,,_ (app app _) _) ()
β-creates-rep {c = app} (_,,_ (app lam (_,,_ A (_,,_ δ out))) (_,,_ ε out)) {σ = σ} βI =
  created = δ ⟦ x₀:= ε ⟧ ;
  red-created = βI ;
  ap-created = let open ≡-Reasoning {A = Expression _ (varKind -Proof)} in
    begin
      δ ⟦ x₀:= ε ⟧ 〈 σ 〉
    ≡⟨⟨ sub-comp₁ {E = δ} ⟩⟩
      δ ⟦ σ •₁ x₀:= ε ⟧
    ≡⟨ sub-cong δ comp₁-botsub ⟩
      δ ⟦ x₀:= (ε 〈 σ 〉) •₂ Rep↑ -Proof σ ⟧
    ≡⟨ sub-comp₂ {E = δ} ⟩
      δ 〈 Rep↑ -Proof σ 〉 ⟦ x₀:= (ε 〈 σ 〉) ⟧
      ∎ }
β-creates-rep {c = lam} _ ()
β-creates-rep {c = bot} _ ()
```

```
β-creates-rep {c = imp} _ ()
--TODO Refactor common pattern
```

The rules of deduction of the system are as follows.

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

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

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

```
PContext : ℕ → Set
PContext P = Context' ∅ -Proof P

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

Palphabet-faithful : ∀ {P} {Q} {ρ σ : Rep (Palphabet P) (Palphabet Q)} → (∀ x → ρ -Proo
Palphabet-faithful {zero} _ ()
Palphabet-faithful {suc _} ρ-is-σ x₀ = cong var (ρ-is-σ zero)
Palphabet-faithful {suc _} {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) (nonV
  var : ∀ {P} {Γ : PContext P} {p : Fin P} → Γ ⊢ var (embedr p) : typeof' p Γ
  app : ∀ {P} {Γ : PContext P} {δ} {ε} {φ} {ψ} → Γ ⊢ δ : φ ⇛ ψ → Γ ⊢ ε : φ → Γ ⊢ appP
  Λ : ∀ {P} {Γ : PContext P} {φ} {δ} {ψ} → (_,_ {K = -Proof} Γ φ) ⊢ δ : liftE ψ → Γ ⊢ Λ
```

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} → (Fin P → Fin Q) → Rep (Palphabet P) (Palphabet Q)
toRep {zero} f K ()
toRep {suc P} f .-Proof x₀ = embedr (f zero)
toRep {suc P} {Q} f K (↑ x) = toRep {P} {Q} (f ∘ suc) K x

toRep-embedr : ∀ {P} {Q} {f : Fin P → Fin Q} {x : Fin P} → toRep f -Proof (embedr x) ≡
toRep-embedr {zero} {_} {_} {()}
toRep-embedr {suc _} {_} {_} {zero} = refl
toRep-embedr {suc P} {Q} {f} {suc x} = toRep-embedr {P} {Q} {f ∘ suc} {x}

toRep-comp : ∀ {P} {Q} {R} {g : Fin Q → Fin R} {f : Fin P → Fin Q} → toRep g •R toRep
toRep-comp {zero} ()
toRep-comp {suc _} {g = g} x₀ = cong var (toRep-embedr {f = g})
```

```
toRep-comp {suc _} {g = g} {f = f} (↑ x) = toRep-comp {g = g} {f = f ∘ suc} x

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

toRep-↑ : ∀ {P} → toRep {P} {suc P} suc ∼R (λ _ → ↑)
toRep-↑ {zero} = λ ()
toRep-↑ {suc P} = Palphabet-faithful {suc P} {suc (suc P)} {toRep {suc P} {suc (suc P)} s

toRep-lift : ∀ {P} {Q} {f : Fin P → Fin Q} → toRep (lift (suc zero) f) ∼R Rep↑ -Proof
toRep-lift x₀ = refl
toRep-lift {zero} (↑ ())
toRep-lift {suc _} (↑ x₀) = refl
toRep-lift {suc P} {Q} {f} (↑ (↑ x)) = trans
  (sym (toRep-comp {g = suc} {f = f ∘ suc} x))
  (toRep-↑ {Q} (toRep (f ∘ suc) _ x))

↑-typed : ∀ {P} {Γ : PContext P} {φ : Expression (Palphabet P) (nonVarKind -Prp)} →
  suc : Γ ⇒R (Γ , φ)
↑-typed {P} {Γ} {φ} x = rep-cong {E = typeof' x Γ} (λ x → sym (toRep-↑ {P} x))

Rep↑-typed : ∀ {P} {Q} {ρ} {Γ : PContext P} {Δ : PContext Q} {φ : Expression (Palphabet P
  lift 1 ρ : (Γ , φ) ⇒R (Δ , φ ⟨ toRep ρ ⟩)
Rep↑-typed {P} {Q = Q} {ρ = ρ} {φ = φ} ρ:Γ→Δ zero =
  let open ≡-Reasoning {A = Expression (Palphabet Q , -Proof) (nonVarKind -Prp)} in
  begin
    liftE (φ ⟨ toRep ρ ⟩)
  ≡⟨⟨ rep-comp {E = φ} ⟩⟩
    φ ⟨ upRep •R toRep ρ ⟩
  ≡⟨⟨ rep-cong {E = φ} (OpFamily.liftOp-up replacement {σ = toRep ρ}) ⟩⟩
    φ ⟨ Rep↑ -Proof (toRep ρ) •R upRep ⟩
  ≡⟨⟨ rep-cong {E = φ} (OpFamily.comp-cong replacement {σ = toRep (lift 1 ρ)} toRep-lift
    φ ⟨ toRep (lift 1 ρ) •R upRep ⟩
  ≡⟨ rep-comp {E = φ} ⟩
    (liftE φ) ⟨ toRep (lift 1 ρ) ⟩
    □
Rep↑-typed {Q = Q} {ρ = ρ} {Γ = Γ} {Δ = Δ} ρ:Γ→Δ (suc x) = let open ≡-Reasoning {A = Exp
  begin
    liftE (typeof' (ρ x) Δ)
  ≡⟨ cong liftE (ρ:Γ→Δ x) ⟩
    liftE ((typeof' x Γ) ⟨ toRep ρ ⟩)
  ≡⟨⟨ rep-comp {E = typeof' x Γ} ⟩⟩
    (typeof' x Γ) ⟨ (λ K x → ↑ (toRep ρ K x)) ⟩
  ≡⟨⟨ rep-cong {E = typeof' x Γ} (λ x → toRep-↑ {Q} (toRep ρ _ x)) ⟩⟩
    (typeof' x Γ) ⟨ toRep {Q} suc •R toRep ρ ⟩
  ≡⟨ rep-cong {E = typeof' x Γ} (toRep-comp {g = suc} {f = ρ}) ⟩
```

```
    (typeof' x Γ) 〈 toRep (lift 1 ρ) •R (λ _ → ↑) 〉
  ≡〈 rep-comp {E = typeof' x Γ} 〉
    (liftE (typeof' x Γ)) 〈 toRep (lift 1 ρ) 〉
    □
```

The replacements between contexts are closed under composition.

```
•R-typed : ∀ {P} {Q} {R} {σ : Fin Q → Fin R} {ρ : Fin P → Fin Q} {Γ} {Δ} {Θ} → ρ : Γ =
  (σ ∘ ρ) : Γ ⇒R Θ
•R-typed {R = R} {σ} {ρ} {Γ} {Δ} {Θ} ρ:Γ→Δ σ:Δ→Θ x = let open ≡-Reasoning {A = Expressi
  begin
    typeof' (σ (ρ x)) Θ
  ≡〈 σ:Δ→Θ (ρ x) 〉
    (typeof' (ρ x) Δ) 〈 toRep σ 〉
  ≡〈 cong (λ x₁ → x₁ 〈 toRep σ 〉) (ρ:Γ→Δ x) 〉
    typeof' x Γ 〈 toRep ρ 〉 〈 toRep σ 〉
  ≡〈〈 rep-comp {E = typeof' x Γ} 〉〉
    typeof' x Γ 〈 toRep σ •R toRep ρ 〉
  ≡〈 rep-cong {E = typeof' x Γ} (toRep-comp {g = σ} {f = ρ}) 〉
    typeof' x Γ 〈 toRep (σ ∘ ρ) 〉
    □
```

Weakening Lemma

```
Weakening : ∀ {P} {Q} {Γ : PContext P} {Δ : PContext Q} {ρ} {δ} {φ} → Γ ⊢ δ : φ → ρ : Γ
Weakening {P} {Q} {Γ} {Δ} {ρ} (var {p = p}) ρ:Γ→Δ = subst₂ (λ x y → Δ ⊢ var x : y)
  (sym (toRep-embedr {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 ρ:
Weakening .{P} {Q} .{Γ} {Δ} {ρ} (Λ {P} {Γ} {φ} {δ} {ψ} Γ,φ⊢δ:ψ) ρ:Γ→Δ = Λ
  (subst (λ P → (Δ , φ 〈 toRep ρ 〉) ⊢ δ 〈 Rep↑ -Proof (toRep ρ) 〉 : P)
  (let open ≡-Reasoning {A = Expression (Palphabet Q , -Proof) (nonVarKind -Prp)} in
  begin
    liftE ψ 〈 Rep↑ -Proof (toRep ρ) 〉
  ≡〈〈 rep-comp {E = ψ} 〉〉
    ψ 〈 (λ _ x → ↑ (toRep ρ _ x)) 〉
  ≡〈 rep-comp {E = ψ} 〉
    liftE (ψ 〈 toRep ρ 〉)
    □)
  (subst₂ (λ x y → (Δ , φ 〈 toRep ρ 〉) ⊢ x : y)
    (rep-cong {E = δ} (toRep-lift {f = ρ}))
    (rep-cong {E = liftE ψ} (toRep-lift {f = ρ}))
    (Weakening {suc P} {suc Q} {Γ , φ} {Δ , φ 〈 toRep ρ 〉} {lift 1 ρ} {δ} {liftE ψ}
      Γ,φ⊢δ:ψ
      claim))) where
  claim : ∀ (x : Fin (suc P)) → typeof' (lift 1 ρ x) (Δ , φ 〈 toRep ρ 〉) ≡ typeof' x (Γ
```

```
  claim zero = let open ≡-Reasoning {A = Expression (Palphabet (suc Q)) (nonVarKind -Prp
    begin
      liftE (φ 〈 toRep ρ 〉)
    ≡〈〈 rep-comp {E = φ} 〉〉
      φ 〈 (λ _ → ↑) •R toRep ρ 〉
    ≡〈 rep-comp {E = φ} 〉
      liftE φ 〈 Rep↑ -Proof (toRep ρ) 〉
    ≡〈〈 rep-cong {E = liftE φ} (toRep-lift {f = ρ}) 〉〉
      liftE φ 〈 toRep (lift 1 ρ) 〉
      □
  claim (suc x) = let open ≡-Reasoning {A = Expression (Palphabet (suc Q)) (nonVarKind -
    begin
      liftE (typeof' (ρ x) Δ)
    ≡〈 cong liftE (ρ:Γ→Δ x) 〉
      liftE (typeof' x Γ 〈 toRep ρ 〉)
    ≡〈〈 rep-comp {E = typeof' x Γ} 〉〉
      typeof' x Γ 〈 (λ _ → ↑) •R toRep ρ 〉
    ≡〈 rep-comp {E = typeof' x Γ} 〉
      liftE (typeof' x Γ) 〈 Rep↑ -Proof (toRep ρ) 〉
    ≡〈〈 rep-cong {E = liftE (typeof' x Γ)} (toRep-lift {f = ρ}) 〉〉
      liftE (typeof' x Γ) 〈 toRep (lift 1 ρ) 〉
      □
```

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 _ (embedr 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 zero = subst (λ p → (Δ , φ ⟦ σ ⟧) ⊢ var x₀ : p)
  (let open ≡-Reasoning {A = Expression (Palphabet Q , -Proof) (nonVarKind -Prp)} in
  begin
    liftE (φ ⟦ σ ⟧)
  ≡〈〈 sub-comp₁ {E = φ} 〉〉
    φ ⟦ (λ _ → ↑) •₁ σ ⟧
  ≡〈 sub-comp₂ {E = φ} 〉
    liftE φ ⟦ Sub↑ -Proof σ ⟧
    □)
  (var {p = zero})
Sub↑-typed {Q = Q} {σ = σ} {Γ = Γ} {Δ = Δ} {φ = φ} σ:Γ→Δ (suc x) =
  subst
  (λ P → (Δ , φ ⟦ σ ⟧) ⊢ Sub↑ -Proof σ -Proof (↑ (embedr x)) : P)
  (let open ≡-Reasoning {A = Expression (Palphabet Q , -Proof) (nonVarKind -Prp)} in
  begin
    liftE (typeof' x Γ ⟦ σ ⟧)
```

```
  ≡⟨⟨ sub-comp₁ {E = typeof' x Γ} ⟩⟩
    typeof' x Γ ⟦ (λ _ → ↑) •₁ σ ⟧
  ≡⟨ sub-comp₂ {E = typeof' x Γ} ⟩
    liftE (typeof' x Γ) ⟦ Sub↑ -Proof σ ⟧
    □)
  (subst₂ (λ x y → (Δ , φ ⟦ σ ⟧) ⊢ x : y)
    (rep-cong {E = σ -Proof (embedr x)} (toRep-↑ {Q}))
    (rep-cong {E = typeof' x Γ ⟦ σ ⟧} (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 zero = subst (λ P₁ → Γ ⊢ δ : P₁)
  (let open ≡-Reasoning {A = Expression (Palphabet P) (nonVarKind -Prp)} in
  begin
    φ
  ≡⟨⟨ sub-idOp ⟩⟩
    φ ⟦ idOpSub _ ⟧
  ≡⟨ sub-comp₂ {E = φ} ⟩
    liftE φ ⟦ x₀:= δ ⟧
    □)
  Γ⊢δ:φ
botsub-typed {P} {Γ} {φ} {δ} _ (suc x) = subst (λ P₁ → Γ ⊢ var (embedr x) : P₁)
  (let open ≡-Reasoning {A = Expression (Palphabet P) (nonVarKind -Prp)} in
  begin
    typeof' x Γ
  ≡⟨⟨ sub-idOp ⟩⟩
    typeof' x Γ ⟦ idOpSub _ ⟧
  ≡⟨ sub-comp₂ {E = typeof' x Γ} ⟩
    liftE (typeof' x Γ) ⟦ x₀:= δ ⟧
    □)
  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) (Substitution
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↑ -Proof σ ⟧ : p)
  (let open ≡-Reasoning {A = Expression (Palphabet Q , -Proof) (nonVarKind -Prp)} in
  begin
    liftE ψ ⟦ Sub↑ -Proof σ ⟧
  ≡⟨⟨ sub-comp₂ {E = ψ} ⟩⟩
    ψ ⟦ Sub↑ -Proof σ •₂ (λ _ → ↑) ⟧
  ≡⟨ sub-comp₁ {E = ψ} ⟩
```

```
    liftE (ψ [ σ ])
    □)
  (Substitution Γ,φ⊢δ:ψ (Sub↑-typed σ:Γ→Δ)))
```

Subject Reduction

```
prop-triv-red : ∀ {P} {φ ψ : Expression (Palphabet P) (nonVarKind -Prp)} → φ ⇒ ψ → ⊥
prop-triv-red {_} {app bot out} (redex ())
prop-triv-red {P} {app bot out} (app ())
prop-triv-red {P} {app imp (_,,_ _ (_,,_ _ out))} (redex ())
prop-triv-red {P} {app imp (_,,_ φ (_,,_ ψ out))} (app (appl φ→φ')) = prop-triv-red {P}
prop-triv-red {P} {app imp (_,,_ φ (_,,_ ψ out))} (app (appr (appl ψ→ψ'))) = prop-triv-r
prop-triv-red {P} {app imp (_,,_ _ (_,,_ _ out))} (app (appr (appr ())))

SR : ∀ {P} {Γ : PContext P} {δ ε : Proof (Palphabet P)} {φ} → Γ ⊢ δ : φ → δ ⇒ ε → Γ ⊢
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 ≡-Reasoning {A = Expression (Palphabet P) (nonVarKind -Prp)} in
  begin
    liftE ψ [ x₀:= ε ]
  ≡⟨⟨ sub-comp₂ {E = ψ} ⟩⟩
    ψ [ idOpSub _ ]
  ≡⟨ sub-idOp ⟩
    ψ
    □)
  (Substitution Γ,φ⊢δ:ψ (botsub-typed Γ⊢ε:φ))
SR (app Γ⊢δ:φ→ψ Γ⊢ε:φ) (app (appl 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 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 _) (redex ())
SR (Λ {P = P} {φ = φ} {δ = δ} {ψ = ψ} Γ⊢δ:φ) (app (appl {N = φ'} δ→ε)) = ⊥-elim (prop-t
SR (Λ Γ⊢δ:φ) (app (appr (appl δ→ε))) = Λ (SR Γ⊢δ:φ δ→ε)
SR (Λ _) (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 out) δ = (Γ ⊢ δ : ⊥P ⟨ (λ _ ()) ⟩ ) × SN δ
C Γ (app imp (_,,_ φ (_,,_ ψ out))) δ = (Γ ⊢ δ : (φ ⇛ ψ) ⟨ (λ _ ()) ⟩) ×
  (∀ Q {Δ : PContext Q} ρ ε → ρ : Γ ⇒R Δ → C Δ φ ε → C Δ ψ (appP (δ ⟨ toRep ρ ⟩) ε))
```

```
C-typed : ∀ {P} {Γ : PContext P} {φ} {δ} → C Γ φ δ → Γ ⊢ δ : φ 〈 (λ _ ()) 〉
C-typed {φ = app bot out} = proj₁
C-typed {Γ = Γ} {φ = app imp (_,,_ φ (_,,_ ψ out))} {δ = δ} = λ x → subst (λ P → Γ ⊢ δ
  (cong₂ _⇛_ (rep-cong {E = φ} (λ ())) (rep-cong {E = ψ} (λ ())))
  (proj₁ 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 out} (Γ⊢δ:x₀ , SNδ) ρ:Γ→Δ = (Weakening Γ⊢δ:x₀ ρ:Γ→Δ) , SNap β-creates
C-rep {P} {Q} {Γ} {Δ} {app imp (_,,_ φ (_,,_ ψ out))} {δ} {ρ} (Γ⊢δ:φ⇛ψ , Cδ) ρ:Γ→Δ = (s
  (λ x → Δ ⊢ δ 〈 toRep ρ 〉 : x)
  (cong₂ _⇛_
  (let open ≡-Reasoning {A = Expression (Palphabet Q) (nonVarKind -Prp)} in
    begin
      (φ 〈 _ 〉) 〈 toRep ρ 〉
    ≡⟨⟨ rep-comp {E = φ} ⟩⟩
      φ 〈 _ 〉
    ≡⟨ rep-cong {E = φ} (λ ()) ⟩
      φ 〈 _ 〉
      ∎)
--TODO Refactor common pattern
  (let open ≡-Reasoning {A = Expression (Palphabet Q) (nonVarKind -Prp)} in
    begin
      ψ 〈 _ 〉 〈 toRep ρ 〉
    ≡⟨⟨ rep-comp {E = ψ} ⟩⟩
      ψ 〈 _ 〉
    ≡⟨ rep-cong {E = ψ} (λ ()) ⟩
      ψ 〈 _ 〉
      ∎))
  (Weakening Γ⊢δ:φ⇛ψ ρ:Γ→Δ)) ,
  (λ R σ ε σ:Δ→Θ ε∈Cφ → subst (C _ ψ) (cong (λ x → appP x ε)
    (trans (sym (rep-cong {E = δ} (toRep-comp {g = σ} {f = ρ}))) (rep-comp {E = δ})))
    (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 ⇒ ε → C Γ φ ε
C-red {φ = app bot out} (Γ⊢δ:x₀ , SNδ) δ→ε = (SR Γ⊢δ:x₀ δ→ε) , (SNred SNδ (osr-red δ→ε)
C-red {Γ = Γ} {φ = app imp (_,,_ φ (_,,_ ψ out))} {δ = δ} (Γ⊢δ:φ⇛ψ , Cδ) δ→δ' = (SR (su
  (cong₂ _⇛_ (rep-cong {E = φ} (λ ())) (rep-cong {E = ψ} (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 (Respects-Creat
```

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 1. *If δ is neutral and $\delta \to_\beta \epsilon$ then ϵ is neutral.*

```
neutral-red : ∀ {P} {δ ε : Proof P} → Neutral δ → δ ⇒ ε → Neutral ε
neutral-red (varNeutral _) ()
neutral-red (appNeutral .(app lam (_,,_ _ (_,,_ _ out))) _ ()) (redex βI)
neutral-red (appNeutral _ ε neutralδ) (app (appl δ→δ')) = appNeutral _ ε (neutral-red ne
neutral-red (appNeutral δ _ neutralδ) (app (appr (appl ε→ε'))) = appNeutral δ _ neutralδ
neutral-red (appNeutral _ _ _) (app (appr (appr ())))

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

Lemma 2. *Let $\Gamma \vdash \delta : \phi$. If δ is neutral and, for all ϵ such that $\delta \rightarrow_\beta \epsilon$, we have $\epsilon \in C_\Gamma(\phi)$, then $\delta \in C_\Gamma(\phi)$.*

```
NeutralC-lm : ∀ {P} {δ ε : Proof P} {X : Proof P → Set} →
  Neutral δ →
  (∀ δ' → δ ⇒ δ' → X (appP δ' ε)) →
  (∀ ε' → ε ⇒ ε' → X (appP δ ε')) →
  ∀ χ → appP δ ε ⇒ χ → X χ
NeutralC-lm () _ _ ._ (redex βI)
NeutralC-lm _ hyp1 _ .(app app (_,,_ _ (_,,_ _ out))) (app (appl δ→δ')) = hyp1 _ δ→δ'
NeutralC-lm _ _ hyp2 .(app app (_,,_ _ (_,,_ _ out))) (app (appr (appl ε→ε'))) = hyp2 _
NeutralC-lm _ _ _ .(app app (_,,_ _ (_,,_ _ _))) (app (appr (appr ())))

mutual
  NeutralC : ∀ {P} {Γ : PContext P} {δ : Proof (Palphabet P)} {φ : Prp} →
    Γ ⊢ δ : φ 〈 (λ _ ()) 〉 → Neutral δ →
    (∀ ε → δ ⇒ ε → C Γ φ ε) →
    C Γ φ δ
  NeutralC {P} {Γ} {δ} {app bot out} Γ⊢δ:x₀ Neutralδ hyp = Γ⊢δ:x₀ , SNI δ (λ ε δ→ε → pro
  NeutralC {P} {Γ} {δ} {app imp (_,,_ φ (_,,_ ψ out))} Γ⊢δ:φ→ψ neutralδ hyp = (subst (λ
    (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 : Fin P → Fin Q} ε → SN ε → ρ : Γ ⇒R Δ → C Δ φ ε → C Δ ψ (a
    claim {Q} {Δ} {ρ} ε (SNI .ε SNε) ρ:Γ→Δ ε∈Cφ = NeutralC {Q} {Δ} {appP (δ 〈 toRep ρ 〉)
      (app (subst (λ P₁ → Δ ⊢ δ 〈 toRep ρ 〉 : P₁)
      (cong₂ _⇒_
      (let open ≡-Reasoning {A = Expression (Palphabet Q) (nonVarKind -Prp)} in
        begin
          φ 〈 _ 〉 〈 toRep ρ 〉
        ≡〈〈 rep-comp {E = φ} 〉〉
          φ 〈 _ 〉
        ≡〈〈 rep-cong {E = φ} (λ ()) 〉〉
          φ 〈 _ 〉
          □)
      (  (let open ≡-Reasoning {A = Expression (Palphabet Q) (nonVarKind -Prp)} in
        begin
```

```
      ψ 〈 _ 〉 〈 toRep ρ 〉
    ≡〈〈 rep-comp {E = ψ} 〉〉
      ψ 〈 _ 〉
    ≡〈〈 rep-cong {E = ψ} (λ ()) 〉〉
      ψ 〈 _ 〉
      □)
    ))
  (Weakening Γ⊢δ:φ→ψ ρ:Γ→Δ))
  (C-typed {Q} {Δ} {φ} {ε} ε∈Cφ))
  (appNeutral (δ 〈 toRep ρ 〉) ε (neutral-rep neutralδ))
  (NeutralC-lm {X = C Δ ψ} (neutral-rep neutralδ)
  (λ δ' δ〈ρ〉→δ' →
    let δ-creation = create-osr β-creates-rep δ δ〈ρ〉→δ' in
    let δ₀ : Proof (Palphabet P)
        δ₀ = Respects-Creates.creation.created δ-creation in
    let δ⇒δ₀ : δ ⇒ δ₀
        δ⇒δ₀ = Respects-Creates.creation.red-created δ-creation in
    let δ₀〈ρ〉≡δ' : δ₀ 〈 toRep ρ 〉 ≡ δ'
        δ₀〈ρ〉≡δ' = Respects-Creates.creation.ap-created δ-creation 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]
    in subst (C Δ ψ) (cong (λ x → appP x ε) δ₀〈ρ〉≡δ') (proj₂ δ₀∈C[φ⇒ψ] Q ρ ε ρ:Γ→Δ
  (λ ε' ε→ε' → claim ε' (SNε ε' ε→ε') ρ:Γ→Δ (C-red {φ = φ} ε∈Cφ ε→ε')))
```

Lemma 3.

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

```
CsubSN : ∀ {P} {Γ : PContext P} {φ} {δ} → C Γ φ δ → SN δ
CsubSN {P} {Γ} {app bot out} P₁ = proj₂ P₁
CsubSN {P} {Γ} {app imp (_,,_ φ (_,,_ ψ out))} {δ} P₁ =
  let φ' : Expression (Palphabet P) (nonVarKind -Prp)
      φ' = φ 〈 (λ _ ()) 〉 in
  let Γ' : PContext (suc P)
      Γ' = Γ , φ' in
  SNap' {replacement} {Palphabet P} {Palphabet P , -Proof} {E = δ} {σ = upRep} β-respe
    (SNsubbodyl (SNsubexp (CsubSN {Γ = Γ'} {φ = ψ}
    (subst (C Γ' ψ) (cong (λ x → appP x (var x₀)) (rep-cong {E = δ} (toRep-↑ {P = P}))
    (proj₂ P₁ (suc P) suc (var x₀) (λ x → sym (rep-cong {E = typeof' x Γ} (toRep-↑ {P
    (NeutralC {φ = φ}
      (subst (λ x → Γ' ⊢ var x₀ : x)
        (trans (sym (rep-comp {E = φ})) (rep-cong {E = φ} (λ ())))
        (var {p = zero}))
      (varNeutral x₀)
      (λ _ ()))))))))
```

```
module PHOPL where

open import Data.List
open import Data.Nat
open import Data.Fin
open import Prelims
open import Taxonomy
open import Grammar
import Grammar.Context
import Reduction
```

3 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.Taxonomy PHOPLTaxonomy

  data PHOPLcon : ∀ {K : ExpressionKind} → Kind (-Constructor K) → Set where
```

```
    -appProof : PHOPLcon (Π [] (varKind -Proof) (Π [] (varKind -Proof) (out {K = varKind
    -lamProof : PHOPLcon (Π [] (varKind -Term) (Π [ -Proof ] (varKind -Proof) (out {K = 
    -bot : PHOPLcon (out {K = varKind -Term})
    -imp : PHOPLcon (Π [] (varKind -Term) (Π [] (varKind -Term) (out {K = varKind -Term}
    -appTerm : PHOPLcon (Π [] (varKind -Term) (Π [] (varKind -Term) (out {K = varKind -T
    -lamTerm : PHOPLcon (Π [] (nonVarKind -Type) (Π [ -Term ] (varKind -Term) (out {K = 
    -Omega : PHOPLcon (out {K = nonVarKind -Type})
    -func  : PHOPLcon (Π [] (nonVarKind -Type) (Π [] (nonVarKind -Type) (out {K = nonVar

  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
  open Grammar.Context PHOPLGrammar.PHOPL
  open import Grammar.Replacement PHOPLGrammar.PHOPL
  open import Grammar.Substitution PHOPLGrammar.PHOPL
  open import Grammar.Substitution.Botsub PHOPLGrammar.PHOPL

  Type : Set
  Type = Expression ∅ (nonVarKind -Type)

  liftType : ∀ {V} → Type → Expression V (nonVarKind -Type)
  liftType (app -Omega out) = app -Omega out
  liftType (app -func (A ,, B ,, out)) = app -func (liftType A ,, liftType B ,, out)

  Ω : Type
  Ω = app -Omega out

  infix 75 _⇛_
  _⇛_ : Type → Type → Type
  φ ⇛ ψ = app -func (φ ,,  ψ ,, out)

  lowerType : ∀ {V} → Expression V (nonVarKind -Type) → Type
  lowerType (app -Omega ou) = Ω
  lowerType (app -func (φ ,, ψ ,, out)) = 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 out

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

  ΛTerm : ∀ {V} → Type → Term (V , -Term) → Term V
  ΛTerm A M = app -lamTerm (liftType A ,,  M ,, out)

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

  PAlphabet : ℕ → Alphabet → Alphabet
  PAlphabet zero A = A
  PAlphabet (suc P) A = PAlphabet P A , -Proof

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

  liftVar' : ∀ {A} P → Fin P → Var (PAlphabet P A) -Proof
  liftVar' (suc P) zero = x₀
  liftVar' (suc P) (suc 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) : ℕ → Set where
    〈〉 : PContext' V zero
    _,_ : ∀ {P} → PContext' V P → Term V → PContext' V (suc P)

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

  P〈〉 : ∀ {V} → PContext V zero
```

```
  P⟨⟩ = ⟨⟩

  _P,_ : ∀ {V} {P} → PContext V P → Term V → PContext V (suc P)
  _P,_ {V} {P} Δ φ = Δ , φ ⟨ embedl {V} { -Proof} {P} ⟩

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

  varP : ∀ {V} {P} → Fin 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 (δ ,, ε ,, out)

  ΛP : ∀ {V} {P} → Term V → Proof V (suc P) → Proof V P
  ΛP {P = P} φ δ = app -lamProof (liftExp P φ ,, δ ,, out)

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

  propof : ∀ {V} {P} → Fin P → PContext' V P → Term V
  propof zero (_ , φ) = φ
  propof (suc x) (Γ , _) = propof x Γ

  data β : ∀ {V} {K} {C} → Constructor C → Subexpression V (-Constructor K) C → Expres
    βI : ∀ {V} A (M : Term (V , -Term)) N → β -appTerm (ΛTerm A M ,, N ,, out) (M ⟦ x₀:=
  open Reduction PHOPLGrammar.PHOPL β
```

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 w
  var : ∀ {V} {Γ : TContext V} {x} → Γ ⊢ var x : typeof x Γ
  ⊥R : ∀ {V} {Γ : TContext V} → Γ ⊢ ⊥ : Ω 〈 (λ _ ()) 〉
  imp : ∀ {V} {Γ : TContext V} {φ} {ψ} → Γ ⊢ φ : Ω 〈 (λ _ ()) 〉 → Γ ⊢ ψ : Ω 〈 (λ _ ())
  app : ∀ {V} {Γ : TContext V} {M} {N} {A} {B} → Γ ⊢ M : app -func (A ,, B ,, out) →
  Λ : ∀ {V} {Γ : TContext V} {A} {M} {B} → Γ , A ⊢ M : liftE B → Γ ⊢ app -lamTerm (A

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
```