# Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

May 3, 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
```

## 2 Grammars

```
module Grammar where

open import Function
open import Data.Empty
open import Data.Product
```

```
open import Data.Nat public
open import Data.Fin public using (Fin;zero;suc)
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 *taxononmy* consists of:

- a set of *expression kinds*;
- a subset of expression kinds, called the *variable kinds*. We refer to the other expession kinds as *non-variable kinds*.

A *grammar* over a taxonomy consists of:

- 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 $A_{ij}$ is a variable kind, and each $B_i$ and $C$ is an expression kind.

- a function assigning, to each variable kind $K$, an expression kind, the *parent* of $K$.

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 $i$th 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*.

We formalise this as follows. First, we construct the sets of expression kinds, constructor kinds and abstraction kinds over a taxonomy:

```
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* $A$ consists of a finite set of *variables*, to each of which is assigned a variable kind $K$. Let $\emptyset$ be the empty alphabet, and $(A, K)$ be the result of extending the alphabet $A$ with one fresh variable $x_0$ of kind $K$. We write $\mathsf{Var}\ A\ K$ for the set of all variables in $A$ of kind $K$.

```
  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
```

We can now define a grammar over a taxonomy:

```
  record ToGrammar : Set₁ where
    field
      Constructor    : ∀ {K} → Kind (-Constructor K) → Set
      parent         : VarKind → ExpressionKind
```

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

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

    alpha : Alphabet → Kind -Abstraction → Alphabet
```

```
alpha V (out _) = V
alpha V (Π K A) = alpha (V , K) A

beta : Kind -Abstraction → ExpressionKind
beta (out K) = K
beta (Π _ A) = beta A

Abstraction V A = Expression (alpha V A) (beta A)

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
  out₂ : ∀ {V} {K} → Body V {K} out₂
  app₂ : ∀ {V} {K} {A} {C} → Abstraction V A → Body V {K} C → Body V (Π₂ A C)

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

## 2.1 Families of Operations

We now wish to define the operations of *replacement* (replacing one variable with another) and *substitution* of expressions for variables. To this end, we define the following.

A *family of operations* consists of the following data:

- Given alphabets $U$ and $V$, a set of *operations* $\sigma : U \to V$.
- Given an operation $\sigma : U \to V$ and a variable $x$ in $U$ of kind $K$, an expression $\sigma(x)$ over $V$ of kind $K$, the result of *applying* $\sigma$ to $x$.
- For every alphabet $V$, an operation $\mathsf{id}_V : V \to V$, the *identity* operation.
- For any operations $\rho : U \to V$ and $\sigma : V \to W$, an operation $\sigma \circ \rho : U \to W$, the *composite* of $\sigma$ and $\rho$
- For every alphabet $V$ and variable kind $K$, an operation $\uparrow: V \to (V, K)$, the *successor* operation.
- For every operation $\sigma : U \to V$, an operation $(\sigma, K) : (U, K) \to (V, K)$, the result of *lifting* $\sigma$. We write $(\sigma, K_1, K_2, \ldots, K_n)$ for $((\cdots(\sigma, K_1), K_2), \cdots), K_n)$.

such that

1. $\uparrow(x) \equiv x$
2. $\mathsf{id}_V(x) \equiv x$
3. $(\sigma \circ \rho)(x) \equiv \sigma[\rho(x)]$
4. Given $\sigma : U \to V$ and $x \in U$, we have $(\sigma, K)(x) \equiv \sigma(x)$

5. $(\sigma, K)(x_0) \equiv x_0$

where, given an operation $\sigma : U \to V$ and expression $E$ over $U$, the expression $\sigma[E]$ over $V$ is defined by

$$\sigma[x] \overset{=}{\text{def}} \sigma(x) \sigma[c([x_{11}, \ldots, x_{1r_1}]E_1, \ldots, [x_{n1}, \ldots, x_{nr_n}]E_n)] \quad \overset{=}{\text{def}} c([x_{11}, \ldots, x_{1r_1}](\sigma, K_{11}, \ldots, K_{1r_1})[E_1], \ldots, [$$

where $K_{ij}$ is the kind of $x_{ij}$.

We say two operations $\rho, \sigma : U \to V$ are *equivalent*, $\rho \sim \sigma$, iff $\rho(x) \equiv \sigma(x)$ for all $x$. Note that this is equivalent to $\rho[E] \equiv \sigma[E]$ for all $E$.

```
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
```

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* $\sigma^A$ to be $((\cdots(\sigma, A_1), A_2), \cdots), A_n)$.

```
        liftOp' : ∀ {U} {V} A → Op U V → Op (alpha U A) (alpha V A)
        liftOp' (out _) σ = σ
        liftOp' (Π K A) σ = liftOp' A (liftOp K σ)
--TODO Refactor to deal with sequences of kinds instead of abstraction kinds?

        liftOp'-cong : ∀ {U} {V} A {ρ σ : Op U V} → ρ ∼op σ → liftOp' A ρ ∼op liftOp'
        liftOp'-cong (out _) ρ-is-σ = ρ-is-σ
        liftOp'-cong (Π _ 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 _ out₂ = out₂
        ap ρ (app₂ {A = A} E EE) = app₂ (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 out₂ _ = refl
        ap-congl (app₂ {A = A} E F) ρ-is-σ = cong₂ app₂ (ap-congl E (liftOp'-cong A ρ-is-

        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 (alpha V A)
  liftOp'-idOp (out _) = ∼-refl
  liftOp'-idOp {V} (Π K A) = let open EqReasoning (OP (alpha (V , K) A) (alpha (V
    begin
      liftOp' A (liftOp K (idOp V))
    ≈⟨ liftOp'-cong A liftOp-idOp ⟩
      liftOp' A (idOp (V , K))
    ≈⟨ liftOp'-idOp A ⟩
      idOp (alpha (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 = out₂} = refl
  ap-idOp {E = app₂ {A = A} E F} = cong₂ app₂ (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
```

Let $F$, $G$ and $H$ be three families of operations. For all $U$, $V$, $W$, let $\circ$ be a function

$$\circ : FVW \times GUV \to HUW$$

**Lemma 1.** *If $\circ$ respects lifting, then it respects repeated lifting.*

```
module Composition {F G H}
  (circ : ∀ {U} {V} {W} → LiftFamily.Op F V W → LiftFamily.Op G U V → LiftFamily.O
  (liftOp-circ : ∀ {U V W K σ ρ} → LiftFamily._∼op_ H (LiftFamily.liftOp H K (circ
  (apV-circ : ∀ {U} {V} {W} {K} {σ} {ρ} {x : Var U K} → LiftFamily.apV H (circ {U}

  open LiftFamily

  liftOp'-circ : ∀ {U V W} A {σ ρ} → _∼op_ H (liftOp' H A (circ {U} {V} {W} σ ρ)) (
  liftOp'-circ (out _) = ∼-refl H
  liftOp'-circ {U} {V} {W} (Π K A) {σ} {ρ} = let open EqReasoning (OP H _ _) in
    begin
      liftOp' H A (liftOp H K (circ σ ρ))
    ≈⟨ liftOp'-cong H A liftOp-circ ⟩
      liftOp' H A (circ (liftOp F K σ) (liftOp G K ρ))
    ≈⟨ liftOp'-circ A ⟩
      circ (liftOp' F A (liftOp F K σ)) (liftOp' G A (liftOp G K ρ))
      □

  ap-circ : ∀ {U V W C K} (E : Subexpression U C K) {σ ρ} → ap H (circ {U} {V} {W} σ
  ap-circ (var _) = apV-circ
  ap-circ (app c E) = cong (app c) (ap-circ E)
  ap-circ out₂ = refl
  ap-circ (app₂ {A = A} E E') {σ} {ρ} = cong₂ app₂
    (let open ≡-Reasoning in
    begin
      ap H (liftOp' H A (circ σ ρ)) E
    ≡⟨ ap-congl H E (liftOp'-circ A) ⟩
      ap H (circ (liftOp' F A σ) (liftOp' G A ρ)) E
    ≡⟨ ap-circ E ⟩
      ap F (liftOp' F A σ) (ap G (liftOp' G A ρ) E)
      □)
    (ap-circ E')

  circ-cong : ∀ {U V W} {σ σ' : Op F V W} {ρ ρ' : Op G U V} → _∼op_ F σ σ' → _∼op_
  circ-cong {U} {V} {W} {σ} {σ'} {ρ} {ρ'} σ∼σ' ρ∼ρ' x = let open ≡-Reasoning in
    begin
      apV H (circ σ ρ) x
    ≡⟨ apV-circ ⟩
      ap F σ (apV G ρ x)
    ≡⟨ ap-cong F σ∼σ' (ρ∼ρ' x) ⟩
      ap F σ' (apV G ρ' x)
    ≡⟨⟨ apV-circ ⟩⟩
      apV H (circ σ' ρ') x
      □

record IsOpFamily (F : LiftFamily) : Set₂ where
```

```
open LiftFamily F public
field
    comp : ∀ {U} {V} {W} → Op V W → Op U V → Op U W
    apV-comp : ∀ {U} {V} {W} {K} {σ : Op V W} {ρ : Op U V} {x : Var U K} →
      apV (comp σ ρ) x ≡ ap σ (apV ρ x)
    liftOp-comp : ∀ {U} {V} {W} {K} {σ : Op V W} {ρ : Op U V} →
      liftOp K (comp σ ρ) ∼op comp (liftOp K σ) (liftOp K ρ)
```

The following results about operationsare easy to prove.

**Lemma 2.** *1.* $(\sigma, K) \circ \uparrow \sim \uparrow \circ \sigma$

*2.* $(\mathsf{id}_V, K) \sim \mathsf{id}_{V,K}$

*3.* $\mathsf{id}_V[E] \equiv E$

*4.* $(\sigma \circ \rho)[E] \equiv \sigma[\rho[E]]$

```
liftOp-up : ∀ {U} {V} {K} {σ : Op U V} → comp (liftOp K σ) up ∼op comp up σ
liftOp-up {U} {V} {K} {σ} {L} x =
    let open ≡-Reasoning {A = Expression (V , K) (varKind L)} in
      begin
        apV (comp (liftOp K σ) up) x
      ≡⟨ apV-comp ⟩
        ap (liftOp K σ) (apV up x)
      ≡⟨ cong (ap (liftOp K σ)) apV-up ⟩
        apV (liftOp K σ) (↑ x)
      ≡⟨ liftOp-↑ x ⟩
        ap up (apV σ x)
      ≡⟨⟨ apV-comp ⟩⟩
        apV (comp up σ) x
        □

open Composition {F} {F} {F} comp liftOp-comp apV-comp renaming (liftOp'-circ to li
```

The alphabets and operations up to equivalence form a category, which we denote **Op**. The action of application associates, with every operator family, a functor **Op** → **Set**, which maps an alphabet $U$ to the set of expressions over $U$, and every operation $\sigma$ to the function $\sigma[-]$. This functor is faithful and injective on objects, and so **Op** can be seen as a subcategory of **Set**.

```
assoc : ∀ {U} {V} {W} {X} {τ : Op W X} {σ : Op V W} {ρ : Op U V} → comp τ (comp σ
assoc {U} {V} {W} {X} {τ} {σ} {ρ} {K} x = let open ≡-Reasoning {A = Expression X (
    begin
      apV (comp τ (comp σ ρ)) x
    ≡⟨ apV-comp ⟩
      ap τ (apV (comp σ ρ) x)
    ≡⟨ cong (ap τ) apV-comp ⟩
```

```
        ap τ (ap σ (apV ρ x))
      ≡⟨⟨ ap-comp (apV ρ x) ⟩⟩
        ap (comp τ σ) (apV ρ x)
      ≡⟨⟨ apV-comp ⟩⟩
        apV (comp (comp τ σ) ρ) x
        □

  unitl : ∀ {U} {V} {σ : Op U V} → comp (idOp V) σ ∼op σ
  unitl {U} {V} {σ} {K} x = let open ≡-Reasoning {A = Expression V (varKind K)} in
      begin
        apV (comp (idOp V) σ) x
      ≡⟨ apV-comp ⟩
        ap (idOp V) (apV σ x)
      ≡⟨ ap-idOp ⟩
        apV σ x
        □

  unitr : ∀ {U} {V} {σ : Op U V} → comp σ (idOp U) ∼op σ
  unitr {U} {V} {σ} {K} x = let open ≡-Reasoning {A = Expression V (varKind K)} in
      begin
        apV (comp σ (idOp U)) x
      ≡⟨ apV-comp ⟩
        ap σ (apV (idOp U) x)
      ≡⟨ cong (ap σ) (apV-idOp x) ⟩
        apV σ x
        □

record OpFamily : Set₂ where
  field
    liftFamily : LiftFamily
    isOpFamily  : IsOpFamily liftFamily
  open IsOpFamily isOpFamily public
```

## 2.2 Replacement

The operation family of *replacement* is defined as follows. A replacement $\rho : U \to V$ is a function that maps every variable in $U$ to a variable in $V$ of the same kind. Application, idOpentity and composition are simply function application, the idOpentity function and function composition. The successor is the canonical injection $V \to (V, K)$, and $(\sigma, K)$ is the extension of $\sigma$ that maps $x_0$ to $x_0$.

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

Rep↑ : ∀ {U} {V} K → Rep U V → Rep (U , K) (V , K)
```

```
Rep↑ _ _ _ x₀ = x₀
Rep↑ _ ρ K (↑ x) = ↑ (ρ K x)

upRep : ∀ {V} {K} → Rep V (V , K)
upRep _ = ↑

idOpRep : ∀ V → Rep V V
idOpRep _ _ x = x

pre-replacement : PreOpFamily
pre-replacement = record {
  Op = Rep;
  apV = λ ρ x → var (ρ _ x);
  up = upRep;
  apV-up = refl;
  idOp = idOpRep;
  apV-idOp = λ _ → refl }

_∼R_ : ∀ {U} {V} → Rep U V → Rep U V → Set
_∼R_ = PreOpFamily._∼op_ pre-replacement

Rep↑-cong : ∀ {U} {V} {K} {ρ ρ' : Rep U V} → ρ ∼R ρ' → Rep↑ K ρ ∼R Rep↑ K ρ'
Rep↑-cong ρ-is-ρ' x₀ = refl
Rep↑-cong ρ-is-ρ' (↑ x) = cong (var ∘ ↑) (var-inj (ρ-is-ρ' x))

proto-replacement : LiftFamily
proto-replacement = record {
  preOpFamily = pre-replacement ;
  lifting = record {
    liftOp = Rep↑ ;
    liftOp-cong = Rep↑-cong } ;
  isLiftFamily = record {
    liftOp-x₀ = refl ;
    liftOp-↑ = λ _ → refl } }

infix 60 _⟨_⟩
_⟨_⟩ : ∀ {U} {V} {C} {K} → Subexpression U C K → Rep U V → Subexpression V C K
E ⟨ ρ ⟩ = LiftFamily.ap proto-replacement ρ E

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)

Rep↑-comp : ∀ {U} {V} {W} {K} {ρ' : Rep V W} {ρ : Rep U V} → Rep↑ K (ρ' •R ρ) ∼R Re
Rep↑-comp x₀ = refl
Rep↑-comp (↑ _) = refl
```

```
replacement : OpFamily
replacement = record {
  liftFamily = proto-replacement ;
  isOpFamily = record {
    comp = _•R_ ;
    apV-comp = refl ;
    liftOp-comp = Rep↑-comp } }

rep-cong : ∀ {U} {V} {C} {K} {E : Subexpression U C K} {ρ ρ' : Rep U V} → ρ ∼R ρ' -
rep-cong {U} {V} {C} {K} {E} {ρ} {ρ'} ρ-is-ρ' = OpFamily.ap-congl replacement E ρ-is

rep-idOp : ∀ {V} {C} {K} {E : Subexpression V C K} → E ⟨ idOpRep V ⟩ ≡ E
rep-idOp = OpFamily.ap-idOp replacement

rep-comp : ∀ {U} {V} {W} {C} {K} {E : Subexpression U C K} {ρ : Rep U V} {σ : Rep V
  E ⟨ σ •R ρ ⟩ ≡ E ⟨ ρ ⟩ ⟨ σ ⟩
rep-comp {U} {V} {W} {C} {K} {E} {ρ} {σ} = OpFamily.ap-comp replacement E

Rep↑-idOp : ∀ {V} {K} → Rep↑ K (idOpRep V) ∼R idOpRep (V , K)
Rep↑-idOp = OpFamily.liftOp-idOp replacement
--TODO Inline many of these
```

This providOpes 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 = E ⟨ upRep ⟩
--TOOD Inline this
```

## 2.3 Substitution

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 $V$. We now aim to prov that the substitutions form a family of operations, with application and idOpentity being simply function application and idOpentity.

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

pre-substitution : PreOpFamily
pre-substitution = record {
  Op = Sub;
  apV = λ σ x → σ _ x;
  up = λ _ x → var (↑ x);
  apV-up = refl;
  idOp = λ _ _ → var;
```

```
  apV-idOp = λ _ → refl }

open PreOpFamily pre-substitution using () renaming (_∼op_ to _∼_;idOp to idOpSub) p

Sub↑ : ∀ {U} {V} K → Sub U V → Sub (U , K) (V , K)
Sub↑ _ _ _ x₀ = var x₀
Sub↑ _ σ K (↑ x) = (σ K x) ⟨ upRep ⟩

Sub↑-cong : ∀ {U} {V} {K} {σ σ' : Sub U V} → σ ∼ σ' → Sub↑ K σ ∼ Sub↑ K σ'
Sub↑-cong {K = K} σ-is-σ' x₀ = refl
Sub↑-cong σ-is-σ' (↑ x) = cong (λ E → E ⟨ upRep ⟩) (σ-is-σ' x)

SUB↑ : PreOpFamily.Lifting pre-substitution
SUB↑ = record { liftOp = Sub↑ ; liftOp-cong = Sub↑-cong }
```

Then, given an expression $E$ of kind $K$ over $U$, we write $E[\sigma]$ for the application of $\sigma$ to $E$, which is the result of substituting $\sigma(x)$ for $x$ for each variable in $E$, avoidOping capture.

```
infix 60 _⟦_⟧
_⟦_⟧ : ∀ {U} {V} {C} {K} → Subexpression U C K → Sub U V → Subexpression V C K
E ⟦ σ ⟧ = PreOpFamily.Lifting.ap SUB↑ σ E

rep2sub : ∀ {U} {V} → Rep U V → Sub U V
rep2sub ρ K x = var (ρ K x)

Rep↑-is-Sub↑ : ∀ {U} {V} {ρ : Rep U V} {K} → rep2sub (Rep↑ K ρ) ∼ Sub↑ K (rep2sub ρ
Rep↑-is-Sub↑ x₀ = refl
Rep↑-is-Sub↑ (↑ _) = refl

module Substitution where
  open PreOpFamily pre-substitution
  open Lifting SUB↑

  liftOp'-is-liftOp' : ∀ {U} {V} {ρ : Rep U V} {A} → rep2sub (OpFamily.liftOp' repla
  liftOp'-is-liftOp' {ρ = ρ} {A = out _} = ∼-refl {σ = rep2sub ρ}
  liftOp'-is-liftOp' {U} {V} {ρ} {Π K A} = let open EqReasoning (OP _ _) in
    begin
      rep2sub (OpFamily.liftOp' replacement A (Rep↑ K ρ))
    ≈⟨ liftOp'-is-liftOp' {A = A} ⟩
      liftOp' A (rep2sub (Rep↑ K ρ))
    ≈⟨ liftOp'-cong A Rep↑-is-Sub↑ ⟩
      liftOp' A (Sub↑ K (rep2sub ρ))
      □

  rep-is-sub : ∀ {U} {V} {K} {C} (E : Subexpression U K C) {ρ : Rep U V} → E ⟨ ρ ⟩ ≡
```

```
  rep-is-sub (var _) = refl
  rep-is-sub (app c E) = cong (app c) (rep-is-sub E)
  rep-is-sub out₂ = refl
  rep-is-sub (app₂ {A = A} E F) {ρ} = cong₂ app₂
    (let open ≡-Reasoning {A = Expression (alpha _ A) (beta A)} in
    begin
      E 〈 OpFamily.liftOp' replacement A ρ 〉
    ≡〈 rep-is-sub E 〉
      E ⟦ (λ K x → var (OpFamily.liftOp' replacement A ρ K x)) ⟧
    ≡〈 ap-congl E (liftOp'-is-liftOp' {A = A}) 〉
      E ⟦ liftOp' A (λ K x → var (ρ K x)) ⟧
      □)
    (rep-is-sub F)

open Substitution public

proto-substitution : LiftFamily
proto-substitution = record {
  preOpFamily = pre-substitution ;
  lifting = SUB↑ ;
  isLiftFamily = record { liftOp-x₀ = refl ; liftOp-↑ = λ {_} {_} {_} {_} {σ} x → re
```

Composition is defined by $(\sigma \circ \rho)(x) \equiv \rho(x)[\sigma]$.

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

Most of the axioms of a family of operations are easy to verify.

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

Sub↑-comp₁ : ∀ {U} {V} {W} {K} {ρ : Rep V W} {σ : Sub U V} → Sub↑ K (ρ •₁ σ) ∼ Rep↑
Sub↑-comp₁ {K = K} x₀ = refl
Sub↑-comp₁ {U} {V} {W} {K} {ρ} {σ} {L} (↑ x) = let open ≡-Reasoning {A = Expression
  begin
    (σ L x) 〈 ρ 〉 〈 upRep 〉
  ≡〈〈 rep-comp {E = σ L x} 〉〉
    (σ L x) 〈 upRep •R ρ 〉
  ≡〈〉
    (σ L x) 〈 Rep↑ K ρ •R upRep 〉
  ≡〈 rep-comp {E = σ L x} 〉
    (σ L x) 〈 upRep 〉 〈 Rep↑ K ρ 〉
    □
```

```
sub-comp₁ : ∀ {U} {V} {W} {C} {K} {E : Subexpression U C K} {ρ : Rep V W} {σ : Sub U
  E [ ρ •₁ σ ] ≡ E [ σ ] 〈 ρ 〉
sub-comp₁ {E = E} = Composition.ap-circ {proto-replacement} {proto-substitution} {pro
              _•₁_ Sub↑-comp₁ refl E

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)

Sub↑-comp₂ : ∀ {U} {V} {W} {K} {σ : Sub V W} {ρ : Rep U V} → Sub↑ K (σ •₂ ρ) ∼ Sub↑
Sub↑-comp₂ {K = K} x₀ = refl
Sub↑-comp₂ (↑ x) = refl

sub-comp₂ : ∀ {U} {V} {W} {C} {K} {E : Subexpression U C K} {σ : Sub V W} {ρ : Rep U
sub-comp₂ {E = E} = Composition.ap-circ {proto-substitution} {proto-replacement} {pro
              _•₂_ Sub↑-comp₂ refl E

Sub↑-comp : ∀ {U} {V} {W} {ρ : Sub U V} {σ : Sub V W} {K} → Sub↑ K (σ • ρ) ∼ Sub↑ K
Sub↑-comp x₀ = refl
Sub↑-comp {W = W} {ρ = ρ} {σ = σ} {K = K} {L} (↑ x) =
  let open ≡-Reasoning {A = Expression (W , K) (varKind L)} in
  begin
    (ρ L x) [ σ ] 〈 upRep 〉
  ≡〈〈 sub-comp₁ {E = ρ L x} 〉〉
    ρ L x [ upRep •₁ σ ]
  ≡〈 sub-comp₂ {E = ρ L x} 〉
    (ρ L x) 〈 upRep 〉 [ Sub↑ K σ ]
    □

substitution : OpFamily
substitution = record {
  liftFamily = proto-substitution ;
  isOpFamily = record {
    comp = _•_ ;
    apV-comp = refl ;
    liftOp-comp = Sub↑-comp } }
```

Replacement is a special case of substitution:

**Lemma 3.** *Let $\rho$ be a replacement $U \to V$.*

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

```
open OpFamily substitution using (assoc) renaming (liftOp-idOp to Sub↑-idOp;ap-idOp t
```

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 4.** *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:= (E 〈 ρ 〉)) •₂ Rep↑ K ρ
comp₁-botsub x₀ = refl
comp₁-botsub (↑ _) = refl

comp-botsub : ∀ {U} {V} {K} {E : Expression U (varKind K)} {σ : Sub U V} →
  σ • (x₀:= E) ∼ (x₀:= (E ⟦ σ ⟧)) • Sub↑ K σ
comp-botsub x₀ = refl
comp-botsub {σ = σ} {L} (↑ x) = trans (sym sub-idOp) (sub-comp₂ {E = σ L x})
```

## 2.4 Congruences

A *congruence* is a relation $R$ on expressions such that:

1. if $MRN$, then $M$ and $N$ have the same kind;
2. if $M_i R N_i$ for all $i$, then $c[[\vec{x_1}]M_1, \ldots, [\vec{x_n}]M_n]Rc[[\vec{x_1}]N_1, \ldots, [\vec{x_n}]N_n]$.

```
Relation : Set₁
Relation = ∀ {V} {C} {K} → Subexpression V C K → Subexpression V C K → Set

record IsCongruence (R : Relation) : Set where
  field
    ICapp : ∀ {V} {K} {C} {c} {MM NN : Body V {K} C} → R MM NN → R (app c MM) (app
    ICout₂ : ∀ {V} {K} → R {V} { -Constructor K} {out₂} out₂ out₂
    ICappl : ∀ {V} {K} {A} {C} {M N : Abstraction V A} {PP : Body V {K} C} → R M N
    ICappr : ∀ {V} {K} {A} {C} {M : Abstraction V A} {NN PP : Body V {K} C} → R NN
```

## 2.5 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 $\{x_1, \ldots, x_n\}$.

We give ourselves the following operations. Given an alphabet $A$ and finite set $F$, let $\mathsf{extend}\ A\ K\ F$ be the alphabet $A \uplus F$, where each element of $F$ has kind $K$. Let $\mathsf{embedr}$ be the canonical injection $F \to \mathsf{extend}\ A\ K\ F$; thus, for all $x \in F$, we have $\mathsf{embedr}\ x$ is a variable of $\mathsf{extend}\ A\ K\ F$ of kind $K$.

```
  extend : Alphabet → VarKind → ℕ → Alphabet
  extend A K zero = A
  extend A K (suc F) = extend A K F , K

  embedr : ∀ {A} {K} {F} → Fin F → Var (extend A K F) K
  embedr zero = x₀
  embedr (suc x) = ↑ (embedr x)
```

Let $\mathsf{embedl}$ be the canonical injection $A \to \mathsf{extend}\ A\ K\ F$, which is a replacement.

```
  embedl : ∀ {A} {K} {F} → Rep A (extend A K F)
  embedl {F = zero} _ x = x
  embedl {F = suc F} K x = ↑ (embedl {F = F} K x)

  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) = A 〈 upRep 〉
  typeof (↑ x) (Γ , _) = typeof x Γ 〈 upRep 〉

  data Context' (A : Alphabet) (K : VarKind) : ℕ → Set where
    〈〉 : Context' A K zero
    _,_ : ∀ {F} → Context' A K F → Expression (extend A K F) (parent K) → Context' A

  typeof' : ∀ {A} {K} {F} → Fin F → Context' A K F → Expression (extend A K F) (pare
  typeof' zero (_ , A) = A 〈 upRep 〉
  typeof' (suc x) (Γ , _) = typeof' x Γ 〈 upRep 〉

record Grammar : Set₁ where
  field
    taxonomy : Taxonomy
    toGrammar : Taxonomy.ToGrammar taxonomy
  open 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 Prelims
open import Grammar
import Reduction
```

# 3 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 $\delta$ 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 : ∀ {K : ExpressionKind} → Kind (-Constructor K) → Set where
    app : PLCon (Π₂ (out (varKind -Proof)) (Π₂ (out (varKind -Proof)) (out₂ {K = varKind
    lam : PLCon (Π₂ (out (nonVarKind -Prp)) (Π₂ (Π -Proof (out (varKind -Proof))) (out₂ {
    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

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

⊥P : Prp
⊥P = app bot out₂

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

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 (app₂ δ (app₂ ε out₂))

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

data β : ∀ {V} {K} {C : Kind (-Constructor K)} → Constructor C → Subexpression V (-Cons
  βI : ∀ {V} {φ} {δ} {ε} → β {V} app (app₂ (ΛP φ δ) (app₂ ε out₂)) (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} (app₂ (var _) _) ()
β-creates-rep {c = app} (app₂ (app app _) _) ()
β-creates-rep {c = app} (app₂ (app lam (app₂ A (app₂ δ out₂))) (app₂ ε out₂)) {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 -Pro
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) (non
  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 ⊢ app
  Λ : ∀ {P} {Γ : PContext P} {φ} {δ} {ψ} → (_,_ {K = -Proof} Γ φ) ⊢ δ :: liftE ψ → Γ ⊢
```

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

```
toRep : ∀ {P} {Q} → (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
  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 = Ex
  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 = Express
  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 ::
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
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* $\sigma$ from a context $\Gamma$ to a context $\Delta$, $\sigma : \Gamma \to \Delta$, is a substitution $\sigma$ on the syntax such that, for every $x : \phi$ in $\Gamma$, 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 I
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)} {δ
  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) (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↑ -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 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 φ (app₂ ψ out₂))} (app (appl φ→φ')) = prop-triv-red {P}
prop-triv-red {P} {app imp (app₂ φ (app₂ ψ out₂))} (app (appr (appl ψ→ψ'))) = prop-triv-
prop-triv-red {P} {app imp (app₂ _ (app₂ _ out₂))} (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 $\Gamma$ and proposition $\phi$ 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 :: ⊥P 〈 (λ _ ()) 〉 ) × SN δ
C Γ (app imp (app₂ φ (app₂ ψ out₂))) 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 out₂} = proj₁
C-typed {Γ = Γ} {φ = app imp (app₂ φ (app₂ ψ out₂))} {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 out₂} (Γ⊢δ::x₀ , SNδ) ρ::Γ→Δ = (Weakening Γ⊢δ::x₀ ρ::Γ→Δ) , SNap β-creat
C-rep {P} {Q} {Γ} {Δ} {app imp (app₂ φ (app₂ ψ out₂))} {δ} {ρ} (Γ⊢δ::φ⇛ψ , Cδ) ρ::Γ→Δ = (
  (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 out₂} (Γ⊢δ::x₀ , SNδ) δ→ε = (SR Γ⊢δ::x₀ δ→ε) , (SNred SNδ (osr-red δ—
C-red {Γ = Γ} {φ = app imp (app₂ φ (app₂ ψ out₂))} {δ = δ} (Γ⊢δ::φ⇒ψ , Cδ) δ→δ' = (SR (s
  (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-Crea
```

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 5.** *If $\delta$ is neutral and $\delta \to_\beta \epsilon$ then $\epsilon$ is neutral.*

```
neutral-red : ∀ {P} {δ ε : Proof P} → Neutral δ → δ ⇒ ε → Neutral ε
neutral-red (varNeutral _) ()
neutral-red (appNeutral .(app lam (app₂ _ (app₂ _ out₂))) _ ()) (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 _ neutral
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 6.** *Let $\Gamma \vdash \delta : \phi$. If $\delta$ is neutral and, for all $\epsilon$ 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 ⇒ 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 (app₂ _ (app₂ _ out₂))) (app (appl δ→δ')) = hyp1 _ δ→δ'
NeutralC-lm _ _ hyp2 .(app app (app₂ _ (app₂ _ out₂))) (app (appr (appl ε→ε'))) = hyp2 _
NeutralC-lm _ _ _ .(app app (app₂ _ (app₂ _ _))) (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 out₂} Γ⊢δ::x₀ Neutralδ hyp = Γ⊢δ::x₀ , SNI δ (λ ε δ→ε → p
NeutralC {P} {Γ} {δ} {app imp (app₂ φ (app₂ ψ out₂))} 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 (
  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 7.**

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

```
CsubSN : ∀ {P} {Γ : PContext P} {φ} {δ} → C Γ φ δ → SN δ
CsubSN {P} {Γ} {app bot out₂} P₁ = proj₂ P₁
CsubSN {P} {Γ} {app imp (app₂ φ (app₂ ψ out₂))} {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 Prelims
open import Grammar
import Reduction
```

# 4 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 | $M, \phi$ | $::=$ | $x \mid \bot \mid MM \mid \lambda x : A.M \mid \phi \to \phi$ |
| Type | $A$ | $::=$ | $\Omega \mid A \to A$ |
| 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 $V$ : **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} → Kind (-Constructor K) → Set where
    -appProof : PHOPLcon (Π2 (out (varKind -Proof)) (Π2 (out (varKind -Proof)) (out2 {K =
    -lamProof : PHOPLcon (Π2 (out (varKind -Term)) (Π2 (Π -Proof (out (varKind -Proof)))
    -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 -Term (out (varKind -Term)))
    -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;-Omeg
  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 A (app2 B out2))) = app -func (app2 (liftType A) (app2 (liftT
```

```
  Ω : Type
  Ω = app -Omega out₂

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

  lowerType : ∀ {V} → Expression V (nonVarKind -Type) → Type
  lowerType (app -Omega out₂) = Ω
  lowerType (app -func (app₂ φ (app₂ 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 M (app₂ N out₂))

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

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

  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 (app₂ δ (app₂ ε out₂))

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

--  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 (app₂ (ΛTerm A M) (app₂ N out₂))
  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 (app₂ A (app₂ B out
  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
```