Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

January 25, 2016

1 Preliminaries

```
module Prelims where

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

1.1 Conjunction

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

1.2 Functions

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

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

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

1.3 Equality

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

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

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

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

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

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

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

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

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

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

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

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

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

2 Datatypes

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

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

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

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

A *replacement* from U to V is simply a function $U \to V$.

```
Rep : FinSet → FinSet → Set
Rep U V = El U → El V
```

Given $f : A \to B$, define $f + 1 : A + 1 \to B + 1$ by

$$(f + 1)(\perp) = \perp$$
$$(f + 1)(\uparrow x) = \uparrow f(x)$$

```
lift : ∀ {U} {V} → Rep U V → Rep (Lift U) (Lift V)
lift _ ⊥ = ⊥
lift f (↑ x) = ↑ (f x)

liftwd : ∀ {U} {V} {f g : Rep U V} → f ∼ g → lift f ∼ lift g
liftwd f-is-g ⊥ = ref
liftwd f-is-g (↑ x) = wd ↑ (f-is-g x)
```

This makes $(-) + 1$ into a functor **FinSet** $\to$ **FinSet**; that is,

$$\mathrm{id}_V + 1 = \mathrm{id}_{V+1}$$
$$(g \circ f) + 1 = (g + 1) \circ (f + 1)$$

```
liftid : ∀ {V} → lift (id (El V)) ∼ id (El (Lift V))
liftid ⊥ = ref
liftid (↑ _) = ref

liftcomp : ∀ {U} {V} {W} {g : Rep V W} {f : Rep U V} → lift (g ∘ f) ∼ lift g ∘ lift f
liftcomp ⊥ = ref
liftcomp (↑ _) = ref

data List (A : Set) : Set where
  ⟨⟩ : List A
  _::_ : List A → A → List A
```

3 Grammars

```
module Grammar where
```

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

A *grammar* consists of:

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

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

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

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

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

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

```
data AbstractionKind (ExpressionKind : Set) : Set where
  out : ExpressionKind → AbstractionKind ExpressionKind
  Π   : ExpressionKind → AbstractionKind ExpressionKind → AbstractionKind ExpressionKin

data ConstructorKind {ExpressionKind : Set} (K : ExpressionKind) : Set where
  out : ConstructorKind K
  Π   : AbstractionKind ExpressionKind → ConstructorKind K → ConstructorKind K

record Grammar : Set₁ where
  field
    ExpressionKind : Set
    Constructor    : ∀ {K : ExpressionKind} → ConstructorKind K → Set
```

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

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

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

```
data Alphabet : Set where
  ∅ : Alphabet
  _,_ : Alphabet → ExpressionKind → Alphabet

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

mutual
  data Expression (V : Alphabet) (K : ExpressionKind) : Set where
    var : Var V K → Expression V K
    app : ∀ {C : ConstructorKind K} → Constructor C → Body V C → Expression V K

  data Body (V : Alphabet) {K : ExpressionKind} : ConstructorKind K → Set where
    out : Expression V K → Body V out
    app : ∀ {A} {C} → Abstraction V A → Body V C → Body V (Π A C)

  data Abstraction (V : Alphabet) : AbstractionKind ExpressionKind → Set where
    out : ∀ {K} → Expression V K → Abstraction V (out K)
    Λ   : ∀ {K} {A} → Abstraction (V , K) A → Abstraction V (Π K A)
```

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

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

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

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

  _⟨_⟩B : ∀ {U} {V} {K} {C : ConstructorKind K} → Body U C → Rep U V → Body V C
  out E ⟨ ρ ⟩B = out (E ⟨ ρ ⟩)
  (app A EE) ⟨ ρ ⟩B = app (A ⟨ ρ ⟩A) (EE ⟨ ρ ⟩B)

  _⟨_⟩A : ∀ {U} {V} {A} → Abstraction U A → Rep U V → Abstraction V A
  out E ⟨ ρ ⟩A = out (E ⟨ ρ ⟩)
  Λ A ⟨ ρ ⟩A = Λ (A ⟨ Rep↑ ρ ⟩A)
```

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

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

  Sub↑ : ∀ {U} {V} {K} → Sub U V → Sub (U , K) (V , K)
  Sub↑ _ _ ⊥ = var ⊥
  Sub↑ σ K (↑ x) = (σ K x) 〈 (λ _ → ↑) 〉

  mutual
    _⟦_⟧ : ∀ {U} {V} {K} → Expression U K → Sub U V → Expression V K
    (var x) ⟦ σ ⟧ = σ _ x
    (app c EE) ⟦ σ ⟧ = app c (EE ⟦ σ ⟧B)

    _⟦_⟧B : ∀ {U} {V} {K} {C : ConstructorKind K} → Body U C → Sub U V → Body V C
    (out E) ⟦ σ ⟧B = out (E ⟦ σ ⟧)
    (app A EE) ⟦ σ ⟧B = app (A ⟦ σ ⟧A) (EE ⟦ σ ⟧B)

    _⟦_⟧A : ∀ {U} {V} {A} → Abstraction U A → Sub U V → Abstraction V A
    (out E) ⟦ σ ⟧A = out (E ⟦ σ ⟧)
    (Λ A) ⟦ σ ⟧A = Λ (A ⟦ Sub↑ σ ⟧A)
```

```
module PL where
open import Prelims
```

4 Propositional Logic

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

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

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

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.

We write **Proof** (P) for the set of all proofs δ with $\mathrm{FV}(\delta) \subseteq V$.

```
infix 75 _⇒_
data Prp : Set where
  ⊥ : Prp
  _⇒_ : Prp → Prp → Prp

infix 80 _,_
```

```
data PContext : FinSet → Set where
  ⟨⟩ : PContext ∅
  _,_ : ∀ {P} → PContext P → Prp → PContext (Lift P)

propof : ∀ {P} → El P → PContext P → Prp
propof ⊥ (_ , φ) = φ
propof (↑ p) (Γ , _) = propof p Γ

data Proof : FinSet → Set where
  var : ∀ {P} → El P → Proof P
  app : ∀ {P} → Proof P → Proof P → Proof P
  Λ : ∀ {P} → Prp → Proof (Lift P) → Proof P
```

Let P, Q : **FinSet**. Given a term M : **Proof** (P) and a replacement $\rho : P \to Q$, we write $M\{\rho\}$: **Proof** (Q) for the result of replacing each variable x in M with $\rho(x)$.

```
infix 60 _<_>
_<_> : ∀ {P Q} → Proof P → Rep P Q → Proof Q
var p < ρ > = var (ρ p)
app δ ε < ρ > = app (δ < ρ >) (ε < ρ >)
Λ φ δ < ρ > = Λ φ (δ < lift ρ >)
```

With this as the action on arrows, **Proof** () becomes a functor **FinSet** → **Set**.

```
repwd : ∀ {P Q : FinSet} {ρ ρ' : El P → El Q} → ρ ∼ ρ' → ∀ δ → δ < ρ > ≡ δ < ρ' >
repwd ρ-is-ρ' (var p) = wd var (ρ-is-ρ' p)
repwd ρ-is-ρ' (app δ ε) = wd2 app (repwd ρ-is-ρ' δ) (repwd ρ-is-ρ' ε)
repwd ρ-is-ρ' (Λ φ δ) = wd (Λ φ) (repwd (liftwd ρ-is-ρ') δ)

repid : ∀ {Q : FinSet} δ → δ < id (El Q) > ≡ δ
repid (var _) = ref
repid (app δ ε) = wd2 app (repid δ) (repid ε)
repid {Q} (Λ φ δ) = wd (Λ φ) (let open Equational-Reasoning (Proof (Lift Q)) in
  ∵ δ < lift (id (El Q)) >
  ≡ δ < id (El (Lift Q)) >  [ repwd liftid δ ]
  ≡ δ                       [ repid δ ])

repcomp : ∀ {P Q R : FinSet} (ρ : El Q → El R) (σ : El P → El Q) M → M < ρ ∘ σ > ≡ M
repcomp ρ σ (var _) = ref
repcomp ρ σ (app δ ε) = wd2 app (repcomp ρ σ δ) (repcomp ρ σ ε)
repcomp {R = R} ρ σ (Λ φ δ) = wd (Λ φ) (let open Equational-Reasoning (Proof (Lift R)) in
  ∵ δ < lift (ρ ∘ σ) >
  ≡ δ < lift ρ ∘ lift σ >      [ repwd liftcomp δ ]
  ≡ (δ < lift σ >) < lift ρ > [ repcomp _ _ δ ])
```

A *substitution* σ from P to Q, $\sigma : P \Rightarrow Q$, is a function $\sigma : P \to \mathbf{Proof}\,(Q)$.

```
Sub : FinSet → FinSet → Set
Sub P Q = El P → Proof Q
```

The identity substitution $\mathrm{id}_Q : Q \Rightarrow Q$ is defined as follows.

```
idSub : ∀ Q → Sub Q Q
idSub _ = var
```

Given $\sigma : P \Rightarrow Q$ and $M : \mathbf{Proof}\,(P)$, we want to define $M[\sigma] : \mathbf{Proof}\,(Q)$, the result of applying the substitution σ to M. Only after this will we be able to define the composition of two substitutions. However, there is some work we need to do before we are able to do this.

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

```
infix 75 _•₁_
_•₁_ : ∀ {P} {Q} {R} → Rep Q R → Sub P Q → Sub P R
(ρ •₁ σ) u = σ u < ρ >
```

(On the other side, given $\rho : P \to Q$ and $\sigma : Q \Rightarrow R$, the composition is just function composition $\sigma \circ \rho : P \Rightarrow R$.)

Given a substitution $\sigma : P \Rightarrow Q$, define the substitution $\sigma{+}1 : P{+}1 \Rightarrow Q{+}1$ as follows.

```
liftSub : ∀ {P} {Q} → Sub P Q → Sub (Lift P) (Lift Q)
liftSub _ ⊥ = var ⊥
liftSub σ (↑ x) = σ x < ↑ >

liftSub-wd : ∀ {P Q} {σ σ' : Sub P Q} → σ ∼ σ' → liftSub σ ∼ liftSub σ'
liftSub-wd σ-is-σ' ⊥ = ref
liftSub-wd σ-is-σ' (↑ x) = wd (λ x → x < ↑ >) (σ-is-σ' x)
```

Lemma 1. *The operations $\bullet$ and $(-) + 1$ satisfiesd the following properties.*

1. $\mathrm{id}_Q + 1 = \mathrm{id}_{Q+1}$
2. *For $\rho : Q \to R$ and $\sigma : P \Rightarrow Q$, we have $(\rho \bullet \sigma) + 1 = (\rho + 1) \bullet (\sigma + 1)$.*
3. *For $\sigma : Q \Rightarrow R$ and $\rho : P \to Q$, we have $(\sigma \circ \rho) + 1 = (\sigma + 1) \circ (\rho + 1)$.*

```
liftSub-id : ∀ {Q : FinSet} → liftSub (idSub Q) ∼ idSub (Lift Q)
liftSub-id ⊥ = ref
liftSub-id (↑ x) = ref

liftSub-comp₁ : ∀ {P Q R : FinSet} (σ : Sub P Q) (ρ : Rep Q R) →
  liftSub (ρ •₁ σ) ∼ lift ρ •₁ liftSub σ
liftSub-comp₁ σ ρ ⊥ = ref
liftSub-comp₁ {R = R} σ ρ (↑ x) = let open Equational-Reasoning (Proof (Lift R)) in
```

```
  ∵ σ x < ρ > < ↑ >
  ≡ σ x < ↑ ∘ ρ >         [[ repcomp ↑ ρ (σ x) ]]
  ≡ σ x < ↑ > < lift ρ > [ repcomp (lift ρ) ↑ (σ x) ]

liftSub-comp2 : ∀ {P Q R : FinSet} (σ : Sub Q R) (ρ : Rep P Q) →
  liftSub (σ ∘ ρ) ∼ liftSub σ ∘ lift ρ
liftSub-comp2 σ ρ ⊥ = ref
liftSub-comp2 σ ρ (↑ x) = ref
```

Now define $M[\sigma]$ as follows.

```
infix 60 _⟦_⟧
_⟦_⟧ : ∀ {P Q : FinSet} → Proof P → Sub P Q → Proof Q
(var x)   ⟦ σ ⟧ = σ x
(app δ ε) ⟦ σ ⟧ = app (δ ⟦ σ ⟧) (ε ⟦ σ ⟧)
(Λ A δ)   ⟦ σ ⟧ = Λ A (δ ⟦ liftSub σ ⟧)

subwd : ∀ {P Q : FinSet} {σ σ’ : Sub P Q} → σ ∼ σ’ → ∀ δ → δ ⟦ σ ⟧ ≡ δ ⟦ σ’ ⟧
subwd σ-is-σ’ (var x) = σ-is-σ’ x
subwd σ-is-σ’ (app δ ε) = wd2 app (subwd σ-is-σ’ δ) (subwd σ-is-σ’ ε)
subwd σ-is-σ’ (Λ A δ) = wd (Λ A) (subwd (liftSub-wd σ-is-σ’) δ)
```

This interacts with our previous operations in a good way:

Lemma 2.

1. $M[\mathrm{id}_Q] \equiv M$
2. $M[\rho \bullet \sigma] \equiv d[\sigma]\{\rho\}$
3. $M[\sigma \circ \rho] \equiv d < \rho > [\sigma]$

```
subid : ∀ {Q : FinSet} (δ : Proof Q) → δ ⟦ idSub Q ⟧ ≡ δ
subid (var x) = ref
subid (app δ ε) = wd2 app (subid δ) (subid ε)
subid {Q} (Λ φ δ) = let open Equational-Reasoning (Proof Q) in
  ∵ Λ φ (δ ⟦ liftSub (idSub Q) ⟧)
  ≡ Λ φ (δ ⟦ idSub (Lift Q) ⟧)     [ wd (Λ φ) (subwd liftSub-id δ) ]
  ≡ Λ φ δ                          [ wd (Λ φ) (subid δ) ]

rep-sub : ∀ {P} {Q} {R} (σ : Sub P Q) (ρ : Rep Q R) (δ : Proof P) → δ ⟦ σ ⟧ < ρ > ≡ δ ⟦
rep-sub σ ρ (var x) = ref
rep-sub σ ρ (app δ ε) = wd2 app (rep-sub σ ρ δ) (rep-sub σ ρ ε)
rep-sub {R = R} σ ρ (Λ φ δ) = let open Equational-Reasoning (Proof R) in
  ∵ Λ φ ((δ ⟦ liftSub σ ⟧) < lift ρ >)
  ≡ Λ φ (δ ⟦ lift ρ •1 liftSub σ ⟧) [ wd (Λ φ) (rep-sub (liftSub σ) (lift ρ) δ) ]
  ≡ Λ φ (δ ⟦ liftSub (ρ •1 σ) ⟧)    [[ wd (Λ φ) (subwd (liftSub-comp1 σ ρ) δ) ]]
```

```
sub-rep : ∀ {P} {Q} {R} (σ : Sub Q R) (ρ : Rep P Q) δ → δ < ρ > ⟦ σ ⟧ ≡ δ ⟦ σ ∘ ρ ⟧
sub-rep σ ρ (var x) = ref
sub-rep σ ρ (app δ ε) = wd2 app (sub-rep σ ρ δ) (sub-rep σ ρ ε)
sub-rep {R = R} σ ρ (Λ φ δ) = let open Equational-Reasoning (Proof R) in
  ∵ Λ φ ((δ < lift ρ >) ⟦ liftSub σ ⟧)
  ≡ Λ φ (δ ⟦ liftSub σ ∘ lift ρ ⟧)      [ wd (Λ φ) (sub-rep (liftSub σ) (lift ρ) δ) ]
  ≡ Λ φ (δ ⟦ liftSub (σ ∘ ρ) ⟧)         [[ wd (Λ φ) (subwd (liftSub-comp₂ σ ρ) δ) ]]
```

We define the composition of two substitutions, as follows.

```
infix 75 _•_
_•_ : ∀ {P Q R : FinSet} → Sub Q R → Sub P Q → Sub P R
(σ • ρ) x = ρ x ⟦ σ ⟧
```

Lemma 3. *Let $\sigma : Q \Rightarrow R$ and $\rho : P \Rightarrow Q$.*

1. $(\sigma \bullet \rho) + 1 = (\sigma + 1) \bullet (\rho + 1)$
2. $M[\sigma \bullet \rho] \equiv d[\rho][\sigma]$

```
liftSub-comp : ∀ {P} {Q} {R} (σ : Sub Q R) (ρ : Sub P Q) →
  liftSub (σ • ρ) ∼ liftSub σ • liftSub ρ
liftSub-comp σ ρ ⊥ = ref
liftSub-comp σ ρ (↑ x) = trans (rep-sub σ ↑ (ρ x)) (sym (sub-rep (liftSub σ) ↑ (ρ x)))

subcomp : ∀ {P} {Q} {R} (σ : Sub Q R) (ρ : Sub P Q) δ → δ ⟦ σ • ρ ⟧ ≡ δ ⟦ ρ ⟧ ⟦ σ ⟧
subcomp σ ρ (var x) = ref
subcomp σ ρ (app δ ε) = wd2 app (subcomp σ ρ δ) (subcomp σ ρ ε)
subcomp σ ρ (Λ φ δ) = wd (Λ φ) (trans (subwd (liftSub-comp σ ρ) δ)  (subcomp (liftSub σ)
```

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

```
assoc : ∀ {P Q R S} {ρ : Sub R S} {σ : Sub Q R} {τ : Sub P Q} →
  ρ • (σ • τ) ∼ (ρ • σ) • τ
assoc {P} {Q} {R} {X} {ρ} {σ} {τ} x = sym (subcomp ρ σ (τ x))

subunitl : ∀ {P} {Q} {σ : Sub P Q} → idSub Q • σ ∼ σ
subunitl {P} {Q} {σ} x = subid (σ x)

subunitr : ∀ {P} {Q} {σ : Sub P Q} → σ • idSub P ∼ σ
subunitr _ = ref
```

Replacement is a special case of substitution, in the following sense:

Lemma 5. *For any replacement ρ,*

$$\delta\{\rho\} \equiv \delta[\rho]$$

```
rep-is-sub : ∀ {P} {Q} {ρ : El P → El Q} δ → δ < ρ > ≡ δ ⟦ var ∘ ρ ⟧
rep-is-sub (var x) = ref
rep-is-sub (app δ ε) = wd2 app (rep-is-sub δ) (rep-is-sub ε)
rep-is-sub {Q = Q} {ρ} (Λ φ δ) = let open Equational-Reasoning (Proof Q) in
  ∵ Λ φ (δ < lift ρ >)
  ≡ Λ φ (δ ⟦ var ∘ lift ρ ⟧)          [ wd (Λ φ) (rep-is-sub δ) ]
  ≡ Λ φ (δ ⟦ liftSub var ∘ lift ρ ⟧) [[ wd (Λ φ) (subwd (λ x → liftSub-id (lift ρ x)) δ
  ≡ Λ φ (δ ⟦ liftSub (var ∘ ρ) ⟧)    [[ wd (Λ φ) (subwd (liftSub-comp₂ var ρ) δ) ]]
```

Given $\delta : \mathbf{Proof}\,(P)$, let $[\bot := \delta] : P+1 \Rightarrow P$ be the substitution that maps $\bot$ to δ, and $\uparrow x$ to x for $x \in P$. We write $\delta[\epsilon]$ for $\delta[\bot := \epsilon]$.

```
botsub : ∀ {Q} → Proof Q → Sub (Lift Q) Q
botsub δ ⊥ = δ
botsub _ (↑ x) = var x

subbot : ∀ {P} → Proof (Lift P) → Proof P → Proof P
subbot δ ε = δ ⟦ botsub ε ⟧
```

Lemma 6. *Let $\delta : \mathbf{Proof}\,(P)$ and $\sigma : P \Rightarrow Q$. Then*

$$\sigma \bullet [\bot := \delta] \sim [\bot := \delta[\sigma]] \circ (\sigma + 1)$$

```
sub-botsub : ∀ {P} {Q} (σ : Sub P Q) (δ : Proof P) →
  σ • botsub δ ∼ botsub (δ ⟦ σ ⟧) • liftSub σ
sub-botsub σ δ ⊥ = ref
sub-botsub σ δ (↑ x) = let open Equational-Reasoning (Proof _) in
  ∵ σ x
  ≡ σ x ⟦ idSub _ ⟧                    [[ subid (σ x) ]]
  ≡ σ x < ↑ > ⟦ botsub (δ ⟦ σ ⟧) ⟧    [[ sub-rep (botsub (δ ⟦ σ ⟧)) ↑ (σ x) ]]
```

We write $\delta \twoheadrightarrow \epsilon$ iff δ β-reduces to ϵ in zero or more steps, $\delta \twoheadrightarrow^+ \epsilon$ iff δ β-reduces to ϵ in one or more steps, and $\delta \simeq \epsilon$ iff the terms δ and ϵ are β-convertible.

Given substitutions ρ and σ, we write $\rho \twoheadrightarrow \sigma$ iff $\rho(x) \twoheadrightarrow \sigma(x)$ for all x, and $\rho \simeq \sigma$ iff $\rho(x) \simeq \sigma(x)$ for all x.

```
data _→₁_ : ∀ {P} → Proof P → Proof P → Set where
  β : ∀ {P} {φ} {δ} {ε : Proof P} → app (Λ φ δ) ε →₁ subbot δ ε
  ξ : ∀ {P} {φ} {δ} {ε : Proof (Lift P)} → δ →₁ ε → Λ φ δ →₁ Λ φ ε
  appl : ∀ {P} {δ} {δ'} {ε : Proof P} → δ →₁ δ' → app δ ε →₁ app δ' ε
  appr : ∀ {P} {δ ε ε' : Proof P} → ε →₁ ε' → app δ ε →₁ app δ ε'

data _↠⁺_ {P} : Proof P → Proof P → Set where
  red : ∀ {δ} {ε} → δ →₁ ε → δ ↠⁺ ε
  ↠⁺trans : ∀ {γ} {δ} {ε} → γ ↠⁺ δ → δ ↠⁺ ε → γ ↠⁺ ε
```

```
data _↠_ {P} : Proof P → Proof P → Set where
  red : ∀ {δ} {ε} → δ →₁ ε → δ ↠ ε
  ref : ∀ {δ} → δ ↠ δ
  ↠trans : ∀ {γ} {δ} {ε} → γ ↠ δ → δ ↠ ε → γ ↠ ε

data _≃_ {P} : Proof P → Proof P → Set where
  red : ∀ {δ} {ε} → δ →₁ ε → δ ≃ ε
  ref : ∀ {δ} → δ ≃ δ
  ≃sym : ∀ {δ} {ε} → δ ≃ ε → ε ≃ δ
  ≃trans : ∀ {γ} {δ} {ε} → γ ≃ δ → δ ≃ ε → γ ≃ ε
```

Lemma 7. *1. If $\delta \twoheadrightarrow \epsilon$ then $\delta[\sigma] \twoheadrightarrow \epsilon[\sigma]$.*

2. If $\sigma \twoheadrightarrow \tau$ then $\delta[\sigma] \twoheadrightarrow \delta[\tau]$.

Proof. For part 2, we first prove that if $\sigma \twoheadrightarrow \tau$ then $\sigma + 1 \twoheadrightarrow \tau + 1$ using part 1. □

```
sub₁redl : ∀ {P} {Q} {ρ : Sub P Q} {δ ε : Proof P} → δ →₁ ε → δ ⟦ ρ ⟧ →₁ ε ⟦ ρ ⟧
sub₁redl {P} {Q} {ρ} (β .{P} {φ} {δ} {ε}) = subst (λ x → app (Λ φ (δ ⟦ liftSub ρ ⟧)) (ε
  (let open Equational-Reasoning (Proof Q) in
  ∵ (δ ⟦ liftSub ρ ⟧) ⟦ botsub (ε ⟦ ρ ⟧) ⟧
  ≡ δ ⟦ botsub (ε ⟦ ρ ⟧) • liftSub ρ ⟧ [[ subcomp (botsub (ε ⟦ ρ ⟧)) (liftSub ρ) δ ]]
  ≡ δ ⟦ ρ • botsub ε ⟧                  [[ subwd (sub-botsub ρ ε) δ ]]
  ≡ (δ ⟦ botsub ε ⟧) ⟦ ρ ⟧              [ subcomp ρ (botsub ε) δ ])
  β
sub₁redl (ξ δ→₁ε) = ξ (sub₁redl δ→₁ε)
sub₁redl (appl δ→₁ε) = appl (sub₁redl δ→₁ε)
sub₁redl (appr δ→₁ε) = appr (sub₁redl δ→₁ε)
```

The *strongly normalizable* terms are defined inductively as follows.

```
data SN {P} : Proof P → Set₁ where
  SNI : ∀ {φ} → (∀ ψ → φ →₁ ψ → SN ψ) → SN φ
```

Lemma 8. *1. If $de \in SN$ then $d \in SN$ and $e \in SN$.*

2. If $d[\bot := N] \in SN$ then $d \in SN$.

3. If $d \in SN$ and $d \twoheadrightarrow \epsilon$ then $e \in SN$.

4. If $d[x := e] \in SN$ and $e \in SN$ then $(\lambda x : f.d)e \in SN$.

```
SNappl : ∀ {Q} {δ ε : Proof Q} → SN (app δ ε) → SN δ
SNappl {Q} {δ} {ε} (SNI δε-is-SN) = SNI (λ δ' δ→₁δ' → SNappl (δε-is-SN (app δ' ε) (appl

SNappr : ∀ {Q} {δ ε : Proof Q} → SN (app δ ε) → SN ε
SNappr {Q} {δ} {ε} (SNI δε-is-SN) = SNI (λ ε' ε→₁ε' → SNappr (δε-is-SN (app δ ε') (appr
```

```
SNsub : ∀ {Q} {δ : Proof (Lift Q)} {ε} → SN (subbot δ ε) → SN δ
SNsub {Q} {δ} {ε} (SNI δε-is-SN) = SNI (λ δ' δ→₁δ' → SNsub (δε-is-SN (δ' ⟦ botsub ε ⟧)

preSNexp : ∀ {P} {δ : Proof (Lift P)} {ε} {φ} → SN (subbot δ ε) → SN ε → ∀ γ → (app (
preSNexp {P} {δ} {ε} SNδε SNε .(δ ⟦ botsub ε ⟧) β = SNδε
preSNexp {P} {δ} {ε} {φ} SNδε SNε (app .(Λ φ ε₁) .ε) (appl (ξ {.P} {.φ} {.δ} {ε₁} δ→₁ε₁)
  preSNexp SNδε SNε (app (Λ φ ε₁) ε) (appl (ξ δ→₁ε₁))
preSNexp {P} {δ} {ε} {φ} SNδε SNε .(app (Λ φ δ) ε') (appr {.P} {.(Λ φ δ)} {.ε} {ε'} ε→₁ε
  preSNexp SNδε SNε (app (Λ φ δ) ε') (appr ε→₁ε')

SNexp : ∀ {P} {δ : Proof (Lift P)} {ε} {φ} → SN (subbot δ ε) → SN ε → SN (app (Λ φ δ)
SNexp SNδε SNε = SNI (preSNexp SNδε SNε)
```

The rules of deduction of the system are as follows.

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

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

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

```
data _⊢_::_ : ∀ {P} → PContext P → Proof P → Prp → Set₁ where
  var : ∀ {P} {Γ : PContext P} {p} → Γ ⊢ var p :: propof 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} → (Γ , φ) ⊢ δ :: ψ → Γ ⊢ Λ φ δ :: φ ⇒ ψ
```

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

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

```
C : ∀ {P} → PContext P → Prp → Proof P → Set₁
C Γ ⊥ δ = (Γ ⊢ δ :: ⊥) ∧ SN δ
C Γ (φ ⇒ ψ) δ = (Γ ⊢ δ :: φ ⇒ ψ) ∧ (∀ ε → C Γ φ ε → C Γ ψ (app δ ε))
```

Lemma 9.

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

```
CsubSN : ∀ {P} {Γ : PContext P} {φ} {δ} → C Γ φ δ → SN δ
CsubSN {P} {Γ} {⊥} (_ , SNδ) = SNδ
CsubSN {P} {Γ} {φ ⇒ ψ} (x , x₁) = {!!}

module PHOPL where
open import Prelims
```

5 Predicative Higher-Order Propositional Logic

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

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

$$
\begin{array}{lrcl}
\text{Proof} & \delta & ::= & p \mid \delta\delta \mid \lambda p : \phi . \delta \\
\text{Term} & M, \phi & ::= & x \mid \bot \mid MM \mid \phi \to \phi \mid \lambda x : A.M \\
\text{Type} & A & ::= & \Omega \mid A \to A \\
\text{Term Context} & \Gamma & ::= & \langle\rangle \mid \Gamma, x : A \\
\text{Proof Context} & \Delta & ::= & \langle\rangle \mid \Delta, p : \phi \\
\text{Judgement} & \mathcal{J} & ::= & \Gamma \text{ valid} \mid \Gamma \vdash M : A \mid \Gamma, \Delta \text{ valid} \mid \Gamma, \Delta \vdash \delta : \phi
\end{array}
$$

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.

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

```
infix 80 _⇒_
data Type : Set where
  Ω : Type
  _⇒_ : Type → Type → Type

--Context V P is the set of all contexts whose domain consists of the term variables in V
infix 80 _,_
data TContext : FinSet → Set where
  〈〉 : TContext ∅
  _,_ : ∀ {V} → TContext V → Type → TContext (Lift V)

--Term V is the set of all terms M with FV(M) ⊆ V
data Term : FinSet → Set where
  var : ∀ {V} → El V → Term V
  ⊥ : ∀ {V} → Term V
  app : ∀ {V} → Term V → Term V → Term V
  Λ : ∀ {V} → Type → Term (Lift V) → Term V
  _⇒_ : ∀ {V} → Term V → Term V → Term V

data PContext (V : FinSet) : FinSet → Set where
  〈〉 : PContext V ∅
  _,_ : ∀ {P} → PContext V P → Term V → PContext V (Lift P)

--Proof V P is the set of all proofs with term variables among V and proof variables amon
data Proof (V : FinSet) : FinSet → Set₁ where
  var : ∀ {P} → El P → Proof V P
  app : ∀ {P} → Proof V P → Proof V P → Proof V P
  Λ : ∀ {P} → Term V → Proof V (Lift P) → Proof V P
```

Let U, V : **FinSet**. A *replacement* from U to V is just a function $U \to V$. Given a term M : **Term** (U) and a replacement $\rho : U \to V$, we write $M\{\rho\}$: **Term** (V) for the result of replacing each variable x in M with $\rho(x)$.

```
infix 60 _<_>
_<_> : ∀ {U V} → Term U → Rep U V → Term V
(var x) < ρ > = var (ρ x)
⊥ < ρ > = ⊥
(app M N) < ρ > = app (M < ρ >) (N < ρ >)
(Λ A M) < ρ > = Λ A (M < lift ρ >)
(φ ⇒ ψ) < ρ > = (φ < ρ >) ⇒ (ψ < ρ >)
```

With this as the action on arrows, **Term** () becomes a functor **FinSet** → **Set**.

```
repwd : ∀ {U V : FinSet} {ρ ρ' : El U → El V} → ρ ∼ ρ' → ∀ M → M < ρ > ≡ M < ρ' >
repwd ρ-is-ρ' (var x) = wd var (ρ-is-ρ' x)
repwd ρ-is-ρ' ⊥ = ref
repwd ρ-is-ρ' (app M N)= wd2 app (repwd ρ-is-ρ' M) (repwd ρ-is-ρ' N)
repwd ρ-is-ρ' (Λ A M) = wd (Λ A) (repwd (liftwd ρ-is-ρ') M)
repwd ρ-is-ρ' (φ ⇒ ψ) = wd2 _⇒_ (repwd ρ-is-ρ' φ) (repwd ρ-is-ρ' ψ)

repid : ∀ {V : FinSet} M → M < id (El V) > ≡ M
repid (var x) = ref
repid ⊥ = ref
repid (app M N) = wd2 app (repid M) (repid N)
repid (Λ A M) = wd (Λ A) (trans (repwd liftid M) (repid M))
repid (φ ⇒ ψ) = wd2 _⇒_ (repid φ) (repid ψ)

repcomp : ∀ {U V W : FinSet} (σ : El V → El W) (ρ : El U → El V) M → M < σ ∘ ρ > ≡ M
repcomp ρ σ (var x) = ref
repcomp ρ σ ⊥ = ref
repcomp ρ σ (app M N) = wd2 app (repcomp ρ σ M) (repcomp ρ σ N)
repcomp ρ σ (Λ A M) = wd (Λ A) (trans (repwd liftcomp M) (repcomp (lift ρ) (lift σ) M))
repcomp ρ σ (φ ⇒ ψ) = wd2 _⇒_ (repcomp ρ σ φ) (repcomp ρ σ ψ)
```

A *substitution* σ from U to V, $\sigma : U \Rightarrow V$, is a function $\sigma : U \to$ **Term** (V).

```
Sub : FinSet → FinSet → Set
Sub U V = El U → Term V
```

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

```
idSub : ∀ V → Sub V V
idSub _ = var
```

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

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

```
infix 75 _•₁_
_•₁_ : ∀ {U} {V} {W} → Rep V W → Sub U V → Sub U W
(ρ •₁ σ) u = σ u < ρ >
```

(On the other side, given $\rho : U \to V$ and $\sigma : V \Rightarrow W$, the composition is just function composition $\sigma \circ \rho : U \Rightarrow W$.)

Given a substitution $\sigma : U \Rightarrow V$, define the substitution $\sigma + 1 : U + 1 \Rightarrow V + 1$ as follows.

```
liftSub : ∀ {U} {V} → Sub U V → Sub (Lift U) (Lift V)
liftSub _ ⊥ = var ⊥
liftSub σ (↑ x) = σ x < ↑ >

liftSub-wd : ∀ {U V} {σ σ' : Sub U V} → σ ∼ σ' → liftSub σ ∼ liftSub σ'
liftSub-wd σ-is-σ' ⊥ = ref
liftSub-wd σ-is-σ' (↑ x) = wd (λ x → x < ↑ >) (σ-is-σ' x)
```

Lemma 10. *The operations* ﬄ$_1$ *and* $(-) + 1$ *satisfiesd the following properties.*

1. $\mathrm{id}_V + 1 = \mathrm{id}_{V+1}$
2. *For* $\rho : V \to W$ *and* $\sigma : U \Rightarrow V$, *we have* $(\rho \bullet \sigma) + 1 = (\rho + 1) \bullet (\sigma + 1)$.
3. *For* $\sigma : V \Rightarrow W$ *and* $\rho : U \to V$, *we have* $(\sigma \circ \rho) + 1 = (\sigma + 1) \circ (\rho + 1)$.

```
liftSub-id : ∀ {V : FinSet} → liftSub (idSub V) ∼ idSub (Lift V)
liftSub-id ⊥ = ref
liftSub-id (↑ x) = ref

liftSub-comp₁ : ∀ {U V W : FinSet} (σ : Sub U V) (ρ : Rep V W) →
  liftSub (ρ •₁ σ) ∼ lift ρ •₁ liftSub σ
liftSub-comp₁ σ ρ ⊥ = ref
liftSub-comp₁ {W = W} σ ρ (↑ x) = let open Equational-Reasoning (Term (Lift W)) in
   ∵ σ x < ρ > < ↑ >
   ≡ σ x < ↑ ∘ ρ >          [[ repcomp ↑ ρ (σ x) ]]
   ≡ σ x < ↑ > < lift ρ > [ repcomp (lift ρ) ↑ (σ x) ]
--because lift ρ (↑ x) = ↑ (ρ x)

liftSub-comp₂ : ∀ {U V W : FinSet} (σ : Sub V W) (ρ : Rep U V) →
  liftSub (σ ∘ ρ) ∼ liftSub σ ∘ lift ρ
liftSub-comp₂ σ ρ ⊥ = ref
liftSub-comp₂ σ ρ (↑ x) = ref
```

Now define $M[\sigma]$ as follows.

```
--Term is a monad with unit var and the following multiplication
infix 60 _⟦_⟧
_⟦_⟧ : ∀ {U V : FinSet} → Term U → Sub U V → Term V
(var x)   ⟦ σ ⟧ = σ x
⊥         ⟦ σ ⟧ = ⊥
(app M N) ⟦ σ ⟧ = app (M ⟦ σ ⟧) (N ⟦ σ ⟧)
(Λ A M)   ⟦ σ ⟧ = Λ A (M ⟦ liftSub σ ⟧)
(φ ⇒ ψ)   ⟦ σ ⟧ = (φ ⟦ σ ⟧) ⇒ (ψ ⟦ σ ⟧)

subwd : ∀ {U V : FinSet} {σ σ' : Sub U V} → σ ∼ σ' → ∀ M → M ⟦ σ ⟧ ≡ M ⟦ σ' ⟧
subwd σ-is-σ' (var x) = σ-is-σ' x
subwd σ-is-σ' ⊥ = ref
subwd σ-is-σ' (app M N) = wd2 app (subwd σ-is-σ' M) (subwd σ-is-σ' N)
subwd σ-is-σ' (Λ A M) = wd (Λ A) (subwd (liftSub-wd σ-is-σ') M)
subwd σ-is-σ' (φ ⇒ ψ) = wd2 _⇒_ (subwd σ-is-σ' φ) (subwd σ-is-σ' ψ)
```

This interacts with our previous operations in a good way:

Lemma 11. *1.* $M[\mathrm{id}_V] \equiv M$

2. $M[\rho \bullet \sigma] \equiv M[\sigma]\{\rho\}$

3. $M[\sigma \circ \rho] \equiv M < \rho > [\sigma]$

```
subid : ∀ {V : FinSet} (M : Term V) → M ⟦ idSub V ⟧ ≡ M
subid (var x) = ref
subid ⊥ = ref
subid (app M N) = wd2 app (subid M) (subid N)
subid {V} (Λ A M) = let open Equational-Reasoning (Term V) in
  ∵ Λ A (M ⟦ liftSub (idSub V) ⟧)
  ≡ Λ A (M ⟦ idSub (Lift V) ⟧)     [ wd (Λ A) (subwd liftSub-id M) ]
  ≡ Λ A M                          [ wd (Λ A) (subid M) ]
subid (φ ⇒ ψ) = wd2 _⇒_ (subid φ) (subid ψ)

rep-sub : ∀ {U} {V} {W} (σ : Sub U V) (ρ : Rep V W) (M : Term U) → M ⟦ σ ⟧ < ρ > ≡ M ⟦ ρ
rep-sub σ ρ (var x) = ref
rep-sub σ ρ ⊥ = ref
rep-sub σ ρ (app M N) = wd2 app (rep-sub σ ρ M) (rep-sub σ ρ N)
rep-sub {W = W} σ ρ (Λ A M) = let open Equational-Reasoning (Term W) in
  ∵ Λ A ((M ⟦ liftSub σ ⟧) < lift ρ >)
  ≡ Λ A (M ⟦ lift ρ •₁ liftSub σ ⟧) [ wd (Λ A) (rep-sub (liftSub σ) (lift ρ) M) ]
  ≡ Λ A (M ⟦ liftSub (ρ •₁ σ) ⟧)    [[ wd (Λ A) (subwd (liftSub-comp₁ σ ρ) M) ]]
rep-sub σ ρ (φ ⇒ ψ) = wd2 _⇒_ (rep-sub σ ρ φ) (rep-sub σ ρ ψ)

sub-rep : ∀ {U} {V} {W} (σ : Sub V W) (ρ : Rep U V) M → M < ρ > ⟦ σ ⟧ ≡ M ⟦ σ ∘ ρ ⟧
sub-rep σ ρ (var x) = ref
sub-rep σ ρ ⊥ = ref
```

```
sub-rep σ ρ (app M N) = wd2 app (sub-rep σ ρ M) (sub-rep σ ρ N)
sub-rep {W = W} σ ρ (Λ A M) = let open Equational-Reasoning (Term W) in
  ∵ Λ A ((M < lift ρ >) ⟦ liftSub σ ⟧)
  ≡ Λ A (M ⟦ liftSub σ ∘ lift ρ ⟧)       [ wd (Λ A) (sub-rep (liftSub σ) (lift ρ) M) ]
  ≡ Λ A (M ⟦ liftSub (σ ∘ ρ) ⟧)          [[ wd (Λ A) (subwd (liftSub-comp₂ σ ρ) M) ]]
sub-rep σ ρ (φ ⇒ ψ) = wd2 _⇒_ (sub-rep σ ρ φ) (sub-rep σ ρ ψ)
```

We define the composition of two substitutions, as follows.

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

Lemma 12. *Let $\sigma : V \Rightarrow W$ and $\rho : U \Rightarrow V$.*

1. $(\sigma \bullet \rho) + 1 = (\sigma + 1) \bullet (\rho + 1)$
2. $M[\sigma \bullet \rho] \equiv M[\rho][\sigma]$

```
liftSub-comp : ∀ {U} {V} {W} (σ : Sub V W) (ρ : Sub U V) →
  liftSub (σ • ρ) ∼ liftSub σ • liftSub ρ
liftSub-comp σ ρ ⊥ = ref
liftSub-comp σ ρ (↑ x) = trans (rep-sub σ ↑ (ρ x)) (sym (sub-rep (liftSub σ) ↑ (ρ x)))

subcomp : ∀ {U} {V} {W} (σ : Sub V W) (ρ : Sub U V) M → M ⟦ σ • ρ ⟧ ≡ M ⟦ ρ ⟧ ⟦ σ ⟧
subcomp σ ρ (var x) = ref
subcomp σ ρ ⊥ = ref
subcomp σ ρ (app M N) = wd2 app (subcomp σ ρ M) (subcomp σ ρ N)
subcomp σ ρ (Λ A M) = wd (Λ A) (trans (subwd (liftSub-comp σ ρ) M)  (subcomp (liftSub σ)
subcomp σ ρ (φ ⇒ ψ) = wd2 _⇒_ (subcomp σ ρ φ) (subcomp σ ρ ψ)
```

Lemma 13. *The finite sets and substitutions form a category under this composition.*

```
assoc : ∀ {U V W X} {ρ : Sub W X} {σ : Sub V W} {τ : Sub U V} →
  ρ • (σ • τ) ∼ (ρ • σ) • τ
assoc {U} {V} {W} {X} {ρ} {σ} {τ} x = sym (subcomp ρ σ (τ x))

subunitl : ∀ {U} {V} {σ : Sub U V} → idSub V • σ ∼ σ
subunitl {U} {V} {σ} x = subid (σ x)

subunitr : ∀ {U} {V} {σ : Sub U V} → σ • idSub U ∼ σ
subunitr _ = ref

-- The second monad law

rep-is-sub : ∀ {U} {V} {ρ : El U → El V} M → M < ρ > ≡ M ⟦ var ∘ ρ ⟧
rep-is-sub (var x) = ref
```

```
rep-is-sub ⊥ = ref
rep-is-sub (app M N) = wd2 app (rep-is-sub M) (rep-is-sub N)
rep-is-sub {V = V} {ρ} (Λ A M) = let open Equational-Reasoning (Term V) in
  ∵ Λ A (M < lift ρ >)
  ≡ Λ A (M ⟦ var ∘ lift ρ ⟧)          [ wd (Λ A) (rep-is-sub M) ]
  ≡ Λ A (M ⟦ liftSub var ∘ lift ρ ⟧) [[ wd (Λ A) (subwd (λ x → liftSub-id (lift ρ x)) M
  ≡ Λ A (M ⟦ liftSub (var ∘ ρ) ⟧)    [[ wd (Λ A) (subwd (liftSub-comp₂ var ρ) M) ]]
--wd (Λ A) (trans (rep-is-sub M) (subwd {!!} M))
rep-is-sub (φ ⇒ ψ) = wd2 _⇒_ (rep-is-sub φ) (rep-is-sub ψ)

typeof : ∀ {V} → El V → TContext V → Type
typeof ⊥ (_ , A) = A
typeof (↑ x) (Γ , _) = typeof x Γ

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

liftSub-var' : ∀ {U} {V} (ρ : El U → El V) → liftSub (var ∘ ρ) ∼ var ∘ lift ρ
liftSub-var' ρ ⊥ = ref
liftSub-var' ρ (↑ x) = ref

botsub : ∀ {V} → Term V → Sub (Lift V) V
botsub M ⊥ = M
botsub _ (↑ x) = var x

sub-botsub : ∀ {U} {V} (σ : Sub U V) (M : Term U) (x : El (Lift U)) →
  botsub M x ⟦ σ ⟧ ≡ liftSub σ x ⟦ botsub (M ⟦ σ ⟧) ⟧
sub-botsub σ M ⊥ = ref
sub-botsub σ M (↑ x) = let open Equational-Reasoning (Term _) in
  ∵ σ x
  ≡ σ x ⟦ idSub _ ⟧                     [[ subid (σ x) ]]
  ≡ σ x < ↑ > ⟦ botsub (M ⟦ σ ⟧) ⟧      [[ sub-rep (botsub (M ⟦ σ ⟧)) ↑ (σ x) ]]

rep-botsub : ∀ {U} {V} (ρ : El U → El V) (M : Term U) (x : El (Lift U)) →
  botsub M x < ρ > ≡ botsub (M < ρ >) (lift ρ x)
rep-botsub ρ M x = trans (rep-is-sub (botsub M x))
  (trans (sub-botsub (var ∘ ρ) M x) (trans (subwd (λ x₁ → wd (λ y → botsub y x₁) (sym (
  (wd (λ x → x ⟦ botsub (M < ρ >)⟧) (liftSub-var' ρ x))))
--TODO Inline this?

subbot : ∀ {V} → Term (Lift V) → Term V → Term V
subbot M N = M ⟦ botsub N ⟧
```

We write $M \simeq N$ iff the terms M and N are β-convertible, and similarly for proofs.

```
data _↠_ : ∀ {V} → Term V → Term V → Set where
  β : ∀ {V} A (M : Term (Lift V)) N → app (Λ A M) N ↠ subbot M N
  ref : ∀ {V} {M : Term V} → M ↠ M
  ↠trans : ∀ {V} {M N P : Term V} → M ↠ N → N ↠ P → M ↠ P
  app : ∀ {V} {M M' N N' : Term V} → M ↠ M' → N ↠ N' → app M N ↠ app M' N'
  Λ : ∀ {V} {M N : Term (Lift V)} {A} → M ↠ N → Λ A M ↠ Λ A N
  imp : ∀ {V} {φ φ' ψ ψ' : Term V} → φ ↠ φ' → ψ ↠ ψ' → φ ⇒ ψ ↠ φ' ⇒ ψ'

repred : ∀ {U} {V} {ρ : El U → El V} {M N : Term U} → M ↠ N → M < ρ > ↠ N < ρ >
repred {U} {V} {ρ} (β A M N) = subst (λ x → app (Λ A (M < lift ρ > )) (N < ρ >) ↠ x) (s
repred ref = ref
repred (↠trans M↠N N↠P) = ↠trans (repred M↠N) (repred N↠P)
repred (app M↠N M'↠N') = app (repred M↠N) (repred M'↠N')
repred (Λ M↠N) = Λ (repred M↠N)
repred (imp φ↠φ' ψ↠ψ') = imp (repred φ↠φ') (repred ψ↠ψ')

liftSub-red : ∀ {U} {V} {ρ σ : Sub U V} → (∀ x → ρ x ↠ σ x) → (∀ x → liftSub ρ x ↠
liftSub-red ρ↠σ ⊥ = ref
liftSub-red ρ↠σ (↑ x) = repred (ρ↠σ x)

subred : ∀ {U} {V} {ρ σ : Sub U V} (M : Term U) → (∀ x → ρ x ↠ σ x) → M ⟦ ρ ⟧ ↠ M ⟦ σ
subred (var x) ρ↠σ = ρ↠σ x
subred ⊥ ρ↠σ = ref
subred (app M N) ρ↠σ = app (subred M ρ↠σ) (subred N ρ↠σ)
subred (Λ A M) ρ↠σ = Λ (subred M (liftSub-red ρ↠σ))
subred (φ ⇒ ψ) ρ↠σ = imp (subred φ ρ↠σ) (subred ψ ρ↠σ)

subsub : ∀ {U} {V} {W} (σ : Sub V W) (ρ : Sub U V) M → M ⟦ ρ ⟧ ⟦ σ ⟧ ≡ M ⟦ σ • ρ ⟧
subsub σ ρ (var x) = ref
subsub σ ρ ⊥ = ref
subsub σ ρ (app M N) = wd2 app (subsub σ ρ M) (subsub σ ρ N)
subsub σ ρ (Λ A M) = wd (Λ A) (trans (subsub (liftSub σ) (liftSub ρ) M)
  (subwd (λ x → sym (liftSub-comp σ ρ x)) M))
subsub σ ρ (φ ⇒ ψ) = wd2 _⇒_ (subsub σ ρ φ) (subsub σ ρ ψ)

subredr : ∀ {U} {V} {σ : Sub U V} {M N : Term U} → M ↠ N → M ⟦ σ ⟧ ↠ N ⟦ σ ⟧
subredr {U} {V} {σ} (β A M N) = subst (λ x → app (Λ A (M ⟦ liftSub σ ⟧)) (N ⟦ σ ⟧) ↠ x)
  (sym (trans (subsub (botsub (N ⟦ σ ⟧)) (liftSub σ) M) (subwd (λ x → sym (sub-botsub σ
subredr ref = ref
subredr (↠trans M↠N N↠P) = ↠trans (subredr M↠N) (subredr N↠P)
subredr (app M↠M' N↠N') = app (subredr M↠M') (subredr N↠N')
subredr (Λ M↠N) = Λ (subredr M↠N)
subredr (imp φ↠φ' ψ↠ψ') = imp (subredr φ↠φ') (subredr ψ↠ψ')

data _≃_ : ∀ {V} → Term V → Term V → Set₁ where
  β : ∀ {V} {A} {M : Term (Lift V)} {N} → app (Λ A M) N ≃ subbot M N
```

```
ref : ∀ {V} {M : Term V} → M ≃ M
≃sym : ∀ {V} {M N : Term V} → M ≃ N → N ≃ M
≃trans : ∀ {V} {M N P : Term V} → M ≃ N → N ≃ P → M ≃ P
app : ∀ {V} {M M' N N' : Term V} → M ≃ M' → N ≃ N' → app M N ≃ app M' N'
Λ : ∀ {V} {M N : Term (Lift V)} {A} → M ≃ N → Λ A M ≃ Λ A N
imp : ∀ {V} {φ φ' ψ ψ' : Term V} → φ ≃ φ' → ψ ≃ ψ' → φ ⇒ ψ ≃ φ' ⇒ ψ'
```

The *strongly normalizable* terms are defined inductively as follows.

```
data SN {V} : Term V → Set₁ where
  SNI : ∀ {M} → (∀ N → M ↠ N → SN N) → SN M
```

Lemma 14. *1. If $MN \in SN$ then $M \in SN$ and $N \in SN$.*

2. If $M[x := N] \in SN$ then $M \in SN$.

3. If $M \in SN$ and $M \rhd N$ then $N \in SN$.

4. If $M[x := N]\vec{P} \in SN$ and $N \in SN$ then $(\lambda x M)N\vec{P} \in SN$.

```
SNappl : ∀ {V} {M N : Term V} → SN (app M N) → SN M
SNappl {V} {M} {N} (SNI MN-is-SN) = SNI (λ P M▷P → SNappl (MN-is-SN (app P N) (app M▷P

SNappr : ∀ {V} {M N : Term V} → SN (app M N) → SN N
SNappr {V} {M} {N} (SNI MN-is-SN) = SNI (λ P N▷P → SNappr (MN-is-SN (app M P) (app ref

SNsub : ∀ {V} {M : Term (Lift V)} {N} → SN (subbot M N) → SN M
SNsub {V} {M} {N} (SNI MN-is-SN) = SNI (λ P M▷P → SNsub (MN-is-SN (P ⟦ botsub N ⟧) (sub
```

The rules of deduction of the system are as follows.

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

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

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

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

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

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

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

  data _⊢_:_ : ∀ {V} → TContext V → Term V → Type → Set₁ where
    var : ∀ {V} {Γ : TContext V} {x} → Tvalid Γ → Γ ⊢ var x : typeof x Γ
    ⊥ : ∀ {V} {Γ : TContext V} → Tvalid Γ → Γ ⊢ ⊥ : Ω
    imp : ∀ {V} {Γ : TContext V} {φ} {ψ} → Γ ⊢ φ : Ω → Γ ⊢ ψ : Ω → Γ ⊢ φ ⇒ ψ : Ω
    app : ∀ {V} {Γ : TContext V} {M} {N} {A} {B} → Γ ⊢ M : A ⇒ B → Γ ⊢ N : A → Γ ⊢ ap
    Λ : ∀ {V} {Γ : TContext V} {A} {M} {B} → Γ , A ⊢ M : B → Γ ⊢ Λ A M : A ⇒ B

data Pvalid : ∀ {V} {P} → TContext V → PContext V P → Set₁ where
  ⟨⟩ : ∀ {V} {Γ : TContext V} → Tvalid Γ → Pvalid Γ ⟨⟩
  _,_ : ∀ {V} {P} {Γ : TContext V} {Δ : PContext V P} {φ : Term V} → Pvalid Γ Δ → Γ ⊢ 

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