Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

January 20, 2016

```
module main where
```

1 Preliminaries

```
module Prelims where
```

1.1 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)
```

1.2 Equality

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

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

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

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

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

```
trans ref ref = ref

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

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

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

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

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

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

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

2 Datatypes

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

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

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

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

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

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

```
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

open import Prelims

module PL where
open import Prelims
```

3 Propositional Logic

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

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

Proof	δ	::=	$p \mid \delta\delta \mid \lambda p : \phi.\delta$
Proposition	ϕ	::=	$\bot \mid \phi \to \phi$
Proof Context	Δ	::=	$\langle\rangle \mid \Delta, p : \phi$
Judgement	$\mathcal{J}$	::=	$\Delta \vdash \delta : \phi$

where p ranges over proof variables and x ranges over term variables. The variable p is bound within δ in the proof $\lambda p : \phi.\delta$, and the variable x is bound within M in the term $\lambda x : A.M$. We identify proofs and terms up to α-conversion.

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)

--Proof V P is the set of all proofs with term variables among V and proof variables amon
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**. A *replacement* from P to Q is just a function $P \to Q$. 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)) i
  ∵ 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) ]
--because lift ρ (↑ x) = ↑ (ρ 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. $M[\rho \bullet \sigma] \equiv \delta[\sigma]\{\rho\}$
3. $M[\sigma \circ \rho] \equiv \delta < \rho > [\sigma]$

```
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. $M[\sigma \bullet \rho] \equiv \delta[\rho][\sigma]$

```
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) ]]

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

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

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

sub-botsub : ∀ {P} {Q} (σ : Sub P Q) (δ : Proof P) (x : El (Lift P)) →
  botsub δ x ⟦ σ ⟧ ≡ liftSub σ x ⟦ botsub (δ ⟦ 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) ]]

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

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

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

```
data _↠_ : ∀ {Q} → Proof Q → Proof Q → Set where
  β : ∀ {Q} ϕ (δ : Proof (Lift Q)) ϵ → app (Λ ϕ δ) ϵ ↠ subbot δ ϵ
```

```
  ref : ∀ {Q} {δ : Proof Q} → δ ↠ δ
  ↠trans : ∀ {Q} {δ ε P : Proof Q} → δ ↠ ε → ε ↠ P → δ ↠ P
  app : ∀ {Q} {δ δ' ε ε' : Proof Q} → δ ↠ δ' → ε ↠ ε' → app δ ε ↠ app δ' ε'
  ξ : ∀ {Q} {δ ε : Proof (Lift Q)} {φ} → δ ↠ ε → Λ φ δ ↠ Λ φ ε

repred : ∀ {P} {Q} {ρ : El P → El Q} {δ ε : Proof P} → δ ↠ ε → δ < ρ > ↠ ε < ρ >
repred {P} {Q} {ρ} (β φ δ ε) = subst (λ x → app (Λ φ (δ < lift ρ > )) (ε < ρ >) ↠ x) (
repred ref = ref
repred (↠trans M↠ε N↠P) = ↠trans (repred M↠ε) (repred N↠P)
repred (app M↠ε M'↠N') = app (repred M↠ε) (repred M'↠N')
repred (ξ M↠ε) = ξ (repred M↠ε)

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

subred : ∀ {P} {Q} {ρ σ : Sub P Q} (δ : Proof P) → (∀ x → ρ x ↠ σ x) → δ ⟦ ρ ⟧ ↠ δ ⟦
subred (var x) ρ↠σ = ρ↠σ x
subred (app δ ε) ρ↠σ = app (subred δ ρ↠σ) (subred ε ρ↠σ)
subred (Λ φ δ) ρ↠σ = ξ (subred δ (liftSub-red ρ↠σ))

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

subredr : ∀ {P} {Q} {σ : Sub P Q} {δ ε : Proof P} → δ ↠ ε → δ ⟦ σ ⟧ ↠ ε ⟦ σ ⟧
subredr {P} {Q} {σ} (β φ δ ε) = subst (λ x → app (Λ φ (δ ⟦ liftSub σ ⟧)) (ε ⟦ σ ⟧) ↠ x)
  (sym (trans (subsub (botsub (ε ⟦ σ ⟧)) (liftSub σ) δ) (subwd (λ x → sym (sub-botsub σ
subredr ref = ref
subredr (↠trans M↠ε N↠P) = ↠trans (subredr M↠ε) (subredr N↠P)
subredr (app M↠M' N↠N') = app (subredr M↠M') (subredr N↠N')
subredr (ξ δ↠δ') = ξ (subredr δ↠δ')

data _≃_ : ∀ {Q} → Proof Q → Proof Q → Set₁ where
  β : ∀ {Q} {φ} {δ : Proof (Lift Q)} {ε} → app (Λ φ δ) ε ≃ subbot δ ε
  ref : ∀ {Q} {δ : Proof Q} → δ ≃ δ
  ≃sym : ∀ {Q} {δ ε : Proof Q} → δ ≃ ε → ε ≃ ε
  ≃trans : ∀ {Q} {δ ε P : Proof Q} → δ ≃ ε → ε ≃ P → δ ≃ P
  app : ∀ {Q} {δ M' ε N' : Proof Q} → δ ≃ M' → ε ≃ N' → app δ ε ≃ app M' N'
  Λ : ∀ {Q} {δ ε : Proof (Lift Q)} {φ} → δ ≃ ε → Λ φ δ ≃ Λ φ ε
```

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

```
data SN {Q} : Proof Q → Set₁ where
  SNI : ∀ {δ} → (∀ ε → δ ↠ ε → SN ε) → SN δ
```

Lemma 6. *1. If $\delta\epsilon \in SN$ then $\delta \in SN$ and $\epsilon \in SN$.*

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

3. If $\delta \in SN$ and $\delta \rhd N$ then $\epsilon \in SN$.

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

```
SNappl : ∀ {Q} {δ ϵ : Proof Q} → SN (app δ ϵ) → SN δ
SNappl {Q} {δ} {ϵ} (SNI δN-is-SN) = SNI (λ P δ▷P → SNappl (δN-is-SN (app P ϵ) (app δ▷P 

SNappr : ∀ {Q} {δ ϵ : Proof Q} → SN (app δ ϵ) → SN ϵ
SNappr {Q} {δ} {ϵ} (SNI δN-is-SN) = SNI (λ P N▷P → SNappr (δN-is-SN (app δ P) (app ref 

SNsub : ∀ {Q} {δ : Proof (Lift Q)} {ϵ} → SN (subbot δ ϵ) → SN δ
SNsub {Q} {δ} {ϵ} (SNI δN-is-SN) = SNI (λ P δ▷P → SNsub (δN-is-SN (P ⟦ botsub ϵ ⟧) (subr
```

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 ⊢ ap
  Λ : ∀ {P} {Γ : PContext P} {ϕ} {δ} {ψ} → Γ , ϕ ⊢ δ :: ψ → Γ ⊢ Λ ϕ δ :: ϕ ⇒ ψ

module PHOPL where
open import Prelims
```

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	δ	::=	$p \mid \delta\delta \mid \lambda p : \phi.\delta$
Term	M, ϕ	::=	$x \mid \bot \mid MM \mid \phi \to \phi \mid \lambda x : A.M$
Type	A	::=	$\Omega \mid A \to A$
Term Context	Γ	::=	$\langle\rangle \mid \Gamma, x : A$
Proof Context	Δ	::=	$\langle\rangle \mid \Delta, p : \phi$
Judgement	$\mathcal{J}$	::=	$\Gamma \text{ valid} \mid \Gamma \vdash M : A \mid \Gamma, \Delta \text{ valid} \mid \Gamma, \Delta \vdash \delta : \phi$

where p ranges over proof variables and x ranges over term variables. The variable p is bound within δ in the proof $\lambda p : \phi.\delta$, and the variable x is bound within M in the term $\lambda x : A.M$. We identify proofs and terms up to α-conversion.

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

```
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 $U, V : \mathbf{FinSet}$. A *replacement* from U to V is just a function $U \to V$. Given a term $M : \mathbf{Term}\,(U)$ and a replacement $\rho : U \to V$, we write $M\{\rho\} : \mathbf{Term}\,(V)$ 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 $\sigma : U \to \mathbf{Term}\,(V)$.

```
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 $M : \mathbf{Term}\,(U)$, we want to define $M[\sigma] : \mathbf{Term}\,(V)$, 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 7. *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 8. *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 ⟦
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 9. *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 10. *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) (
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 ⟦
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 11. *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 ⊢ va
  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 →
```