Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

January 17, 2016

```
module main where

postulate Level : Set
postulate zero : Level
postulate suc : Level → Level

{-# BUILTIN LEVEL Level #-}
{-# BUILTIN LEVELZERO zero #-}
{-# BUILTIN LEVELSUC suc #-}
```

1 Preliminaries

1.1 Functions

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

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

1.2 Equality

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

subst : ∀ {i} {A : Set i} (P : A → Set₁) {a} {b} → a ≡ b → P a → P b
subst P ref Pa = Pa

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

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

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

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

module Equational-Reasoning {i} (A : Set i) where
  ∵_ : ∀ (a : A) → a ≡ a
  ∵ _ = ref

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

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

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

2 Datatypes

```
data ∅ : Set where

data Lift (A : Set) : Set where
  ⊥ : Lift A
  ↑ : A → Lift A

infix 80 _⟩⟩_
_⟩⟩_ : ∀ {A B : Set} → (A → Lift B) → Lift A → Lift B
f ⟩⟩ ⊥ = ⊥
f ⟩⟩ ↑ x = f x

⟩⟩wd : ∀ {A B : Set} {f g : A → Lift B} → f ∼ g → ∀ x → f ⟩⟩ x ≡ g ⟩⟩ x
⟩⟩wd f-is-g ⊥ = ref
⟩⟩wd f-is-g (↑ x) = f-is-g x

return-do : ∀ {A : Set} (x : Lift A) → ↑ ⟩⟩ x ≡ x
return-do ⊥ = ref
return-do (↑ x) = ref

do-comp : ∀ {A B C : Set} {g : B → Lift C} {f : A → Lift B} (x : Lift A) → g ⟩⟩ (f ⟩⟩ x
```

```
do-comp ⊥ = ref
do-comp (↑ x) = ref

lift : ∀ {A B : Set} → (A → B) → Lift A → Lift B
lift f x = (↑ ∘ f) 〉〉 x

liftwd : ∀ {A B : Set} {f g : A → B} → f ∼ g → lift f ∼ lift g
liftwd f-is-g = 〉〉wd (λ y → wd ↑ (f-is-g y))

liftid : ∀ {A : Set} → lift (id A) ∼ id (Lift A)
liftid = return-do

liftcomp : ∀ {A B C : Set} {f : A → B} {g : B → C} → lift (g ∘ f) ∼ lift g ∘ lift f
liftcomp x = sym (do-comp x)
```

3 Predicative Higher-Order Propositional Logic

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

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

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

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.

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

--Term V is the set of all terms M with FV(M) ⊆ V
data Term : Set → Set₁ where
  var : ∀ {V} → 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

--Proof V P is the set of all proofs with term variables among V and proof variables amon
data Proof (V : Set) : Set → Set₁ where
  var : ∀ {P} → 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

--Context V P is the set of all contexts whose domain consists of the term variables in V
infix 80 _,_
infix 80 _,,_
data Context : Set → Set → Set₁ where
  ⟨⟩ : Context ∅ ∅
  _,_ : ∀ {V} {P} → Context V P → Type → Context (Lift V) P
  _,,_ : ∀ {V} {P} → Context V P → Term V → Context V (Lift P)

--The operation of replacing one variable with another in a term
rep : ∀ {U V : Set} → (U → V) → Term U → Term V
rep ρ (var x) = var (ρ x)
rep ρ ⊥ = ⊥
rep ρ (app M N) = app (rep ρ M) (rep ρ N)
rep ρ (Λ A M) = Λ A (rep (lift ρ) M)
rep ρ (φ ⇒ ψ) = rep ρ φ ⇒ rep ρ ψ

repwd : ∀ {U V : Set} {ρ ρ' : U → V} → ρ ∼ ρ' → rep ρ ∼ rep 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)

rep-comp : ∀ {U V W : Set} (σ : V → W) (ρ : U → V) → rep (σ ∘ ρ) ∼ rep σ ∘ rep ρ
rep-comp ρ σ (var x) = ref
rep-comp ρ σ ⊥ = ref
rep-comp ρ σ (app M N) = wd2 app (rep-comp ρ σ M) (rep-comp ρ σ N)
rep-comp ρ σ (Λ A M) = wd (Λ A) (trans (repwd liftcomp M) (rep-comp (lift ρ) (lift σ) M)
rep-comp ρ σ (φ ⇒ ψ) = wd2 _⇒_ (rep-comp ρ σ φ) (rep-comp ρ σ ψ)

liftTerm : ∀ {V : Set} → Term V → Term (Lift V)
liftTerm = rep ↑
--TODO Inline this?

liftSub : ∀ {U V : Set} → (U → Term V) → Lift U → Term (Lift V)
liftSub _ ⊥ = var ⊥
liftSub σ (↑ x) = liftTerm (σ x)

liftSub-wd : ∀ {U V : Set} {σ σ' : U → Term V} → σ ∼ σ' → liftSub σ ∼ liftSub σ'
liftSub-wd σ-is-σ' ⊥ = ref
liftSub-wd σ-is-σ' (↑ x) = wd (rep ↑) (σ-is-σ' x)

liftSub-var : ∀ {V : Set} (x : Lift V) → liftSub var x ≡ var x
```

```
liftSub-var ⊥ = ref
liftSub-var (↑ x) = ref

liftSub-rep : ∀ {U V W : Set} (σ : U → Term V) (ρ : V → W) (x : Lift U) → liftSub (λ :
liftSub-rep σ ρ ⊥ = ref
liftSub-rep σ ρ (↑ x) = trans (sym (rep-comp ↑ ρ (σ x))) (rep-comp (lift ρ) ↑ (σ x))

liftSub-lift : ∀ {U V W : Set} (σ : V → Term W) (ρ : U → V) (x : Lift U) →
  liftSub σ (lift ρ x) ≡ liftSub (λ x → σ (ρ x)) x
liftSub-lift σ ρ ⊥ = ref
liftSub-lift σ ρ (↑ x) = ref

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

--Term is a monad with unit var and the following multiplication
sub : ∀ {U V : Set} → (U → Term V) → Term U → Term V
sub σ (var x) = σ x
sub σ ⊥ = ⊥
sub σ (app M N) = app (sub σ M) (sub σ N)
sub σ (Λ A M) = Λ A (sub (liftSub σ) M)
sub σ (φ ⇒ ψ) = sub σ φ ⇒ sub σ ψ

subwd : ∀ {U V : Set} {σ σ' : U → Term V} → σ ∼ σ' → sub σ ∼ sub 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)

--The first monad law

subvar : ∀ {V : Set} (M : Term V) → sub var M ≡ M
subvar (var x) = ref
subvar ⊥ = ref
subvar (app M N) = wd2 app (subvar M) (subvar N)
subvar (Λ A M) = wd (Λ A) (trans (subwd liftSub-var M) (subvar M))
subvar (φ ⇒ ψ) = wd2 _⇒_ (subvar φ) (subvar ψ)

infix 75 _•_
_•_ : ∀ {U V W : Set} → (V → Term W) → (U → Term V) → U → Term W
(σ • ρ) x = sub σ (ρ x)

rep-sub : ∀ {U} {V} {W} (σ : U → Term V) (ρ : V → W) → rep ρ ∘ sub σ ∼ sub (rep ρ ∘ σ
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 σ ρ (Λ A M) = wd (Λ A) (trans (rep-sub (liftSub σ) (lift ρ) M) (subwd (λ x → sy
rep-sub σ ρ (φ ⇒ ψ) = wd2 _⇒_ (rep-sub σ ρ φ) (rep-sub σ ρ ψ)

sub-rep : ∀ {U} {V} {W} (σ : V → Term W) (ρ : U → V) →
  sub σ ∘ rep ρ ∼ sub (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 σ ρ (Λ A M) = wd (Λ A) (trans (sub-rep (liftSub σ) (lift ρ) M) (subwd (liftSub-l
sub-rep σ ρ (φ ⇒ ψ) = wd2 _⇒_ (sub-rep σ ρ φ) (sub-rep σ ρ ψ)

liftSub-comp : ∀ {U} {V} {W} (σ : V → Term W) (ρ : U → Term 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)))

-- The second monad law

subcomp : ∀ {U} {V} {W} (σ : V → Term W) (ρ : U → Term V) →
  sub (σ • ρ) ∼ sub σ ∘ sub ρ
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)

rep-is-sub : ∀ {U} {V} {ρ : U → V} → rep ρ ∼ sub (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 (Λ A M) = wd (Λ A) (trans (rep-is-sub M) (subwd var-lift M))
rep-is-sub (φ ⇒ ψ) = wd2 _⇒_ (rep-is-sub φ) (rep-is-sub ψ)

typeof : ∀ {V} {P} → V → Context V P → Type
typeof () ⟨⟩
typeof ⊥ (_ , A) = A
typeof (↑ x) (Γ , _) = typeof x Γ
typeof x (Γ ,, _) = typeof x Γ

propof : ∀ {V} {P} → P → Context V P → Term V
propof () ⟨⟩
propof p (Γ , _) = liftTerm (propof p Γ)
propof p (_ ,, φ) = φ
```

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

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

botsub-liftTerm : ∀ {V} (M N : Term V) → sub (botsub M) (liftTerm N) ≡ N
botsub-liftTerm M (var x) = ref
botsub-liftTerm M ⊥ = ref
botsub-liftTerm M (app N P) = wd2 app (botsub-liftTerm M N) (botsub-liftTerm M P)
botsub-liftTerm M (Λ A N) = wd (Λ A) (trans (sub-rep _ _ N) (trans (subwd (λ x → trans
botsub-liftTerm M (φ ⇒ ψ) = wd2 _⇒_ (botsub-liftTerm M φ) (botsub-liftTerm M ψ)

sub-botsub : ∀ {U} {V} (σ : U → Term V) (M : Term U) (x : Lift U) →
  sub σ (botsub M x) ≡ sub (botsub (sub σ M)) (liftSub σ x)
sub-botsub σ M ⊥ = ref
sub-botsub σ M (↑ x) = sym (botsub-liftTerm (sub σ M) (σ x))

rep-botsub : ∀ {U} {V} (ρ : U → V) (M : Term U) (x : Lift U) →
  rep ρ (botsub M x) ≡ botsub (rep ρ M) (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 (
--TODO Inline this?

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

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

```
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
  ↠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 : U → V} {M N : Term U} → M ↠ N → rep ρ M ↠ rep ρ N
repred {U} {V} {ρ} (β A M N) = subst (λ x → app (Λ A (rep (lift ρ) M)) (rep ρ N) ↠ 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 : U → Term 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 : U → Term V} (M : Term U) → (∀ x → ρ x ↠ σ x) → sub 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 : V → Term W) (ρ : U → Term V) (M : Term U) →
  sub σ (sub ρ M) ≡ sub (λ x → sub σ (ρ x)) M
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 : U → Term V} {M N : Term U} → M ↠ N → sub σ M ↠ sub σ N
subredr {U} {V} {σ} (β A M N) = subst (λ x → app (Λ A (sub (liftSub σ) M)) (sub σ N) →
  (sym (trans (subsub (botsub (sub σ N)) (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 1. *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 (sub (botsub N) P) (s
```

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 valid : ∀ {V} {P} → Context V P → Set₁ where
    ⟨⟩ : valid ⟨⟩
    ctxV : ∀ {V} {P} {Γ : Context V P} {A} → valid Γ → valid (Γ , A)
    ctxP : ∀ {V} {P} {Γ : Context V P} {φ} → Γ ⊢ φ : Ω → valid (Γ ,, φ)

  data _⊢_:_ : ∀ {V} {P} → Context V P → Term V → Type → Set₁ where
    var : ∀ {V} {P} {Γ : Context V P} {x} → valid Γ → Γ ⊢ var x : typeof x Γ
    ⊥ : ∀ {V} {P} {Γ : Context V P} → valid Γ → Γ ⊢ ⊥ : Ω
    imp : ∀ {V} {P} {Γ : Context V P} {φ} {ψ} → Γ ⊢ φ : Ω → Γ ⊢ ψ : Ω → Γ ⊢ φ ⇒ ψ
    app : ∀ {V} {P} {Γ : Context V P} {M} {N} {A} {B} → Γ ⊢ M : A ⇒ B → Γ ⊢ N : A → Γ
```

```
  Λ : ∀ {V} {P} {Γ : Context V P} {A} {M} {B} → Γ , A ⊢ M : B → Γ ⊢ Λ A M : A ⇒ B

data _⊢_::_ : ∀ {V} {P} → Context V P → Proof V P → Term V → Set₁ where
  var : ∀ {V} {P} {Γ : Context V P} {p} → valid Γ → Γ ⊢ var p :: propof p Γ
  app : ∀ {V} {P} {Γ : Context V P} {δ} {ε} {φ} {ψ} → Γ ⊢ δ :: φ ⇒ ψ → Γ ⊢ ε :: φ → Γ
  Λ : ∀ {V} {P} {Γ : Context V P} {φ} {δ} {ψ} → Γ ,, φ ⊢ δ :: ψ → Γ ⊢ Λ φ δ :: φ ⇒ ψ
  conv : ∀ {V} {P} {Γ : Context V P} {δ} {φ} {ψ} → Γ ⊢ δ :: φ → Γ ⊢ ψ : Ω → φ ≃ ψ →
```