Type Theories with Computation Rules for the Univalence Axiom

Robin Adams

February 2, 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

open import Prelims hiding (Rep;_∼_;lift)
```

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 binary relation of *parenthood* on the set of expression kinds.

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 ExpressionKi

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
    parent         : ExpressionKind → ExpressionKind → 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
  x₀ : ∀ {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 : 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

_∼R_ : ∀ {U} {V} → Rep U V → Rep U V → Set
ρ ∼R ρ' = ∀ {K} x → ρ K x ≡ ρ' K x
```

The alphabets and replacements form a category.

```
idRep : ∀ V → Rep V V
idRep _ _ x = x

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)

--We choose not to prove the category axioms, as they hold up to judgemental equality.
```

Given a replacement $\rho : U \to V$, we can 'lift´ this to a replacement $(\rho, K) : (U, K) \to (V, K)$. Under this operation, the mapping $(-, K)$ becomes an endofunctor on the category of alphabets and replacements.

```
Rep↑ : ∀ {U} {V} {K} → Rep U V → Rep (U , K) (V , K)
Rep↑ _ _ x₀ = x₀
Rep↑ ρ K (↑ x) = ↑ (ρ K x)

Rep↑-wd : ∀ {U} {V} {K} {ρ ρ' : Rep U V} → ρ ∼R ρ' → Rep↑ {K = K} ρ ∼R Rep↑ ρ'
Rep↑-wd ρ-is-ρ' x₀ = ref
Rep↑-wd ρ-is-ρ' (↑ x) = wd ↑ (ρ-is-ρ' x)

Rep↑-id : ∀ {V} {K} → Rep↑ (idRep V) ∼R idRep (V , K)
Rep↑-id x₀ = ref
Rep↑-id (↑ _) = ref

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

Finally, we can define $E\langle\rho\rangle$, the result of replacing each variable x in E with $\rho(x)$. Under this operation, the mapping $\mathsf{Expression}\ -\ K$ becomes a functor from the category of alphabets and replacements to the category of sets.

```
mutual
  infix 60 _⟨_⟩
  _⟨_⟩ : ∀ {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)

  infix 60 _⟨_⟩B
  _⟨_⟩B : ∀ {U} {V} {K} {C : ConstructorKind K} → Body U C → Rep U V → Body V C
  out ⟨ ρ ⟩B = out
  (app A EE) ⟨ ρ ⟩B = app (A ⟨ ρ ⟩A) (EE ⟨ ρ ⟩B)

  infix 60 _⟨_⟩A
  _⟨_⟩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)

mutual
  rep-wd : ∀ {U} {V} {K} {E : Expression U K} {ρ : Rep U V} {ρ'} → ρ ∼R ρ' → E ⟨ ρ ⟩
  rep-wd {U} {V} {K} {var x} ρ-is-ρ' = wd var (ρ-is-ρ' x)
  rep-wd {U} {V} {K} {app c EE} ρ-is-ρ' = wd (app c) (rep-wdB ρ-is-ρ')

  rep-wdB : ∀ {U} {V} {K} {C : ConstructorKind K} {EE : Body U C} {ρ ρ' : Rep U V} → ρ
  rep-wdB {U} {V} {K} {out} {out} ρ-is-ρ' = ref
```

```
  rep-wdB {U} {V} {K} {Π A C} {app A' EE} ρ-is-ρ' = wd2 app (rep-wdA ρ-is-ρ') (rep-wdB

  rep-wdA : ∀ {U} {V} {A} {E : Abstraction U A} {ρ ρ' : Rep U V} → ρ ∼R ρ' → E 〈 ρ 〉A
  rep-wdA {U} {V} {out K} {out E} ρ-is-ρ' = wd out (rep-wd ρ-is-ρ')
  rep-wdA {U} {V} {Π K A} {Λ E} ρ-is-ρ' = wd Λ (rep-wdA (Rep↑-wd ρ-is-ρ'))

mutual
  rep-id : ∀ {V} {K} {E : Expression V K} → E 〈 idRep V 〉 ≡ E
  rep-id {E = var _} = ref
  rep-id {E = app c _} = wd (app c) rep-idB

  rep-idB : ∀ {V} {K} {C : ConstructorKind K} {EE : Body V C} → EE 〈 idRep V 〉B ≡ EE
  rep-idB {EE = out} = ref
  rep-idB {EE = app _ _} = wd2 app rep-idA rep-idB

  rep-idA : ∀ {V} {K} {A : Abstraction V K} → A 〈 idRep V 〉A ≡ A
  rep-idA {A = out _} = wd out rep-id
  rep-idA {A = Λ _} = wd Λ (trans (rep-wdA Rep↑-id) rep-idA)

mutual
  rep-comp : ∀ {U} {V} {W} {K} {ρ : Rep U V} {ρ' : Rep V W} {E : Expression U K} → E 
  rep-comp {E = var _} = ref
  rep-comp {E = app c _} = wd (app c) rep-compB

  rep-compB : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {ρ : Rep U V} {ρ' : Rep V W} {
  rep-compB {EE = out} = ref
  rep-compB {U} {V} {W} {K} {Π L C} {ρ} {ρ'} {app A EE} = wd2 app rep-compA rep-compB

  rep-compA : ∀ {U} {V} {W} {K} {ρ : Rep U V} {ρ' : Rep V W} {A : Abstraction U K} → 
  rep-compA {A = out _} = wd out rep-comp
  rep-compA {U} {V} {W} {Π K L} {ρ} {ρ'} {Λ A} = wd Λ (trans (rep-wdA Rep↑-comp) rep-c
```

This provides us with the canonical mapping from an expression over V to an expression over (V, K):

```
lift : ∀ {V} {K} {L} → Expression V L → Expression (V , K) L
lift E = E 〈 (λ _ → ↑) 〉
```

A *substitution* σ from alphabet U to alphabet V, $\sigma : U \Rightarrow V$, 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

_∼_ : ∀ {U} {V} → Sub U V → Sub U V → Set
σ ∼ τ = ∀ K x → σ K x ≡ τ K x
```

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

```
idSub : ∀ {V} → Sub V V
idSub _ x = var x
```

Given $\sigma : U \to V$ and an expression E over U, we want to define the expression $E[\sigma]$ over 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) K x = σ K x ⟨ ρ ⟩

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)
```

Given a substitution $\sigma : U \Rightarrow V$, define a substitution $(\sigma, K) : (U, K) \Rightarrow (V, K)$ as follows.

```
Sub↑ : ∀ {U} {V} {K} → Sub U V → Sub (U , K) (V , K)
Sub↑ _ _ x₀ = var x₀
Sub↑ σ K (↑ x) = lift (σ K x)

Sub↑-wd : ∀ {U} {V} {K} {σ σ' : Sub U V} → σ ∼ σ' → Sub↑ {K = K} σ ∼ Sub↑ σ'
Sub↑-wd {K = K} σ-is-σ' .K x₀ = ref
Sub↑-wd σ-is-σ' L (↑ x) = wd lift (σ-is-σ' L x)
```

Lemma 1. *The operations we have defined satisfy the following properties.*

1. $(\mathrm{id}_V, K) = \mathrm{id}_{(V,K)}$
2. $(\rho \bullet_1 \sigma, K) = (\rho, K) \bullet_1 (\sigma, K)$
3. $(\sigma \bullet_2 \rho, K) = (\sigma, K) \bullet_2 (\rho, K)$

```
Sub↑-id : ∀ {V} {K} → Sub↑ {V} idSub ∼ idSub
Sub↑-id {K = K} .K x₀ = ref
Sub↑-id _ (↑ _) = ref

Sub↑-comp₁ : ∀ {U} {V} {W} {K} {ρ : Rep V W} {σ : Sub U V} → Sub↑ (ρ •₁ σ) ∼ Rep↑ ρ •
Sub↑-comp₁ {K = K} .K x₀ = ref
Sub↑-comp₁ {U} {V} {W} {K} {ρ} {σ} L (↑ x) = let open Equational-Reasoning (Expression
  ∵ lift (σ L x ⟨ ρ ⟩)
  ≡ σ L x ⟨ (λ _ x → ↑ (ρ _ x)) ⟩ [[ rep-comp ]]
  ≡ (lift (σ L x)) ⟨ Rep↑ ρ ⟩     [ rep-comp ]
```

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

We can now define the result of applying a substitution σ to an expression E, which we denote $E[\sigma]$.

```
mutual
  infix 60 _⟦_⟧
  _⟦_⟧ : ∀ {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)

  infix 60 _⟦_⟧B
  _⟦_⟧B : ∀ {U} {V} {K} {C : ConstructorKind K} → Body U C → Sub U V → Body V C
  out ⟦ σ ⟧B = out
  (app A EE) ⟦ σ ⟧B = app (A ⟦ σ ⟧A) (EE ⟦ σ ⟧B)

  infix 60 _⟦_⟧A
  _⟦_⟧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)

mutual
  sub-wd : ∀ {U} {V} {K} {E : Expression U K} {σ σ' : Sub U V} → σ ∼ σ' → E ⟦ σ ⟧ ≡
  sub-wd {E = var x} σ-is-σ' = σ-is-σ' _ x
  sub-wd {U} {V} {K} {app c EE} σ-is-σ' = wd (app c) (sub-wdB σ-is-σ')

  sub-wdB : ∀ {U} {V} {K} {C : ConstructorKind K} {EE : Body U C} {σ σ' : Sub U V} → σ
  sub-wdB {EE = out} σ-is-σ' = ref
  sub-wdB {EE = app A EE} σ-is-σ' = wd2 app (sub-wdA σ-is-σ') (sub-wdB σ-is-σ')

  sub-wdA : ∀ {U} {V} {K} {A : Abstraction U K} {σ σ' : Sub U V} → σ ∼ σ' → A ⟦ σ ⟧A
  sub-wdA {A = out E} σ-is-σ' = wd out (sub-wd {E = E} σ-is-σ')
  sub-wdA {U} {V} {Π K L} {Λ A} σ-is-σ' = wd Λ (sub-wdA (Sub↑-wd σ-is-σ'))
```

Lemma 2.

1. $M[\mathrm{id}_V] \equiv M$
2. $M[\rho \bullet_1 \sigma] \equiv M[\sigma]\langle\rho\rangle$
3. $M[\sigma \bullet_2 \rho] \equiv M\langle\rho\rangle[\sigma]$

```
mutual
  subid : ∀ {V} {K} {E : Expression V K} → E ⟦ idSub ⟧ ≡ E
  subid {E = var _} = ref
  subid {V} {K} {app c _} = wd (app c) subidB
```

```
  subidB : ∀ {V} {K} {C : ConstructorKind K} {EE : Body V C} → EE ⟦ idSub ⟧B ≡ EE
  subidB {EE = out} = ref
  subidB {EE = app _ _} = wd2 app subidA subidB

  subidA : ∀ {V} {K} {A : Abstraction V K} → A ⟦ idSub ⟧A ≡ A
  subidA {A = out _} = wd out subid
  subidA {A = Λ _} = wd Λ (trans (sub-wdA Sub↑-id) subidA)

mutual
  sub-comp₁ : ∀ {U} {V} {W} {K} {E : Expression U K} {ρ : Rep V W} {σ : Sub U V} →
    E ⟦ ρ •₁ σ ⟧ ≡ E ⟦ σ ⟧ 〈 ρ 〉
  sub-comp₁ {E = var _} = ref
  sub-comp₁ {E = app c _} = wd (app c) sub-comp₁B

  sub-comp₁B : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {EE : Body U C} {ρ : Rep V W}
    EE ⟦ ρ •₁ σ ⟧B ≡ EE ⟦ σ ⟧B 〈 ρ 〉B
  sub-comp₁B {EE = out} = ref
  sub-comp₁B {U} {V} {W} {K} {(Π L C)} {app A EE} = wd2 app sub-comp₁A sub-comp₁B

  sub-comp₁A : ∀ {U} {V} {W} {K} {A : Abstraction U K} {ρ : Rep V W} {σ : Sub U V} →
    A ⟦ ρ •₁ σ ⟧A ≡ A ⟦ σ ⟧A 〈 ρ 〉A
  sub-comp₁A {A = out E} = wd out (sub-comp₁ {E = E})
  sub-comp₁A {U} {V} {W} {(Π K L)} {Λ A} = wd Λ (trans (sub-wdA Sub↑-comp₁) sub-comp₁A)

mutual
  sub-comp₂ : ∀ {U} {V} {W} {K} {E : Expression U K} {σ : Sub V W} {ρ : Rep U V} → E ⟦
  sub-comp₂ {E = var _} = ref
  sub-comp₂ {U} {V} {W} {K} {app c EE} = wd (app c) sub-comp₂B

  sub-comp₂B : ∀ {U} {V} {W} {K} {C : ConstructorKind K} {EE : Body U C}
    {σ : Sub V W} {ρ : Rep U V} → EE ⟦ σ •₂ ρ ⟧B ≡ EE 〈 ρ 〉B ⟦ σ ⟧B
  sub-comp₂B {EE = out} = ref
  sub-comp₂B {U} {V} {W} {K} {Π L C} {app A EE} = wd2 app sub-comp₂A sub-comp₂B

  sub-comp₂A : ∀ {U} {V} {W} {K} {A : Abstraction U K} {σ : Sub V W} {ρ : Rep U V} → A
  sub-comp₂A {A = out E} = wd out (sub-comp₂ {E = E})
  sub-comp₂A {U} {V} {W} {Π K L} {Λ A} = wd Λ (trans (sub-wdA Sub↑-comp₂) sub-comp₂A)
```

We define the composition of two substitutions, as follows.

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

Lemma 3. *Let $\sigma : V \Rightarrow W$ and $\rho : U \Rightarrow V$.*

1. $(\sigma \bullet \rho, K) \sim (\sigma, K) \bullet (\rho, K)$

2. $E[\sigma \bullet \rho] \equiv E[\rho][\sigma]$

```
Sub↑-comp : ∀ {U} {V} {W} {ρ : Sub U V} {σ : Sub V W} {K} →
  Sub↑ {K = K} (σ • ρ) ∼ Sub↑ σ • Sub↑ ρ
Sub↑-comp _ x₀ = ref
Sub↑-comp {W = W} {ρ = ρ} {σ = σ} {K = K} L (↑ x) =
  let open Equational-Reasoning (Expression (W , K) L) in
  ∵ lift ((ρ L x) ⟦ σ ⟧)
  ≡ (lift (ρ L x)) ⟦ Sub↑ σ ⟧ [ {!!} ]
```

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

```
data Context : Alphabet → Set where
  〈〉 : Context ∅
  _,_ : ∀ {V} {K} {L} {_ : parent K L} → Context V → Expression V L → Context (V , K

typekindof : ∀ {V} {K} → Var V K → Context V → ExpressionKind
typekindof x₀ (_,_ {L = L} _ _) = L
typekindof (↑ x) (Γ , _) = typekindof x Γ

typeof : ∀ {V} {K} (x : Var V K) (Γ : Context V) → Expression V (typekindof x Γ)
typeof x₀ (_ , A)     = A 〈 (λ _ → ↑) 〉
typeof (↑ x) (Γ , _) = typeof x Γ 〈 ( λ _ → ↑ ) 〉
```

```
module PL where

open import Grammar
```

4 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 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.