Nominal techniques as an Agda library

Murdoch J. Gabbay, Orestis Melkonian

14 June 2023

Nominal/Abs.agda

```
open import Prelude.Init; open SetAsType
open import Prelude.DecEq

module Nominal.Abs (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Abs.Base Atom public
open import Nominal.Abs.Lift Atom public
open import Nominal.Abs.Support Atom public
open import Nominal.Abs.Product Atom public
```

Nominal/Abs/Base.agda

```agda
{-# OPTIONS --v equivariance:100 #-}
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.Setoid
open import Prelude.Bifunctor
open import Prelude.InferenceRules

module Nominal.Abs.Base (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.New  Atom
open import Nominal.Swap Atom

-- TODO: maybe this is broken, user has access to `atom`
record Abs (A : Type ℓ) : Type ℓ where
  constructor abs
```

```
    field atom : Atom
          term : A
  open Abs public

  module _ {A : Type ℓ} ⦃ _ : Swap A ⦄ where

    conc : Abs A → Atom → A
    conc (abs 𝕒 x) 𝕓 = swap 𝕓 𝕒 x

    instance
      Swap-Abs : Swap (Abs A)
      Swap-Abs .swap 𝕒 𝕓 (abs 𝕔 x) = abs (swap 𝕒 𝕓 𝕔) (swap 𝕒 𝕓 x)
      -- this is the conjugation action for nominal abstractions
      -- (terminology from G-sets, sets with a group action)

    private
```

```
  variable
    𝕒 𝕓 𝕔 : Atom
    x : A

  _ : swap 𝕒 𝕓 (abs 𝕔 x)
    ≡ abs (swap 𝕒 𝕓 𝕔) (swap 𝕒 𝕓 x)
  _ = refl

  _ : conc (abs 𝕒 x) 𝕓 ≡ swap 𝕓 𝕒 x
  _ = refl

module _ ⦃ is : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄ ⦃ _ : SwapLaws A ⦄ whe
  -- swap-conc : ∀ (f : Abs A) →
  --   ⦅ 𝕒 ↔ 𝕓 ⦆ (conc f 𝕔) ≈ conc (⦅ 𝕒 ↔ 𝕓 ⦆ f) (⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔
  swap-conc : Equivariant conc
  swap-conc _ _ = swap-swap
```

```
-- ** α-equivalence
_≈α_ : Rel (Abs A) (is .relℓ)
f ≈α g = И (λ 𝕩 → conc f 𝕩 ≈ conc g 𝕩)

instance
  Setoid-Abs : ISetoid (Abs A)
  Setoid-Abs = λ where
    .relℓ → is .relℓ
    ._≈_ → _≈α_

private variable f g h : Abs A

≈α-refl : f ≈α f
≈α-refl = [] , (λ _ _ → ≈-refl)

≈α-sym : f ≈α g → g ≈α f
```

```
≈α-sym = map₂′ (≈-sym ∘₂_)

≈α-trans : f ≈α g → g ≈α h → f ≈α h
≈α-trans (xs , f≈g) (ys , g≈h) = (xs ++ ys) , λ y y∉ →
  ≈-trans (f≈g y (y∉ ∘ L.Mem.∈-++⁺ˡ)) (g≈h y (y∉ ∘ L.Mem.∈-++⁺ʳ xs

instance
  SetoidLaws-Abs : SetoidLaws (Abs A)
  SetoidLaws-Abs .isEquivalence = record
    { refl = ≈α-refl ; sym = ≈α-sym ; trans = ≈α-trans }

cong-abs : ∀ {t t′ : A} → t ≈ t′ → abs 𝕒 t ≈ abs 𝕒 t′
cong-abs t≈ = [] , λ _ _ → cong-swap t≈

cong-conc : ∀ {t̂ t̂′ : Abs A} →
  ∀ (eq : t̂ ≈ t̂′) →
```

```
    • 𝕒 ∉ eq .proj₁
      ────────────────
      conc t̂ 𝕒
    ≈ conc t̂′ 𝕒
  cong-conc (_ , eq) = eq _

  cong-conc∘abs : ∀ {t t′ : A} →
    ∀ (eq : t ≈ t′) →
      ────────────────
      conc (abs 𝕓 t) 𝕒
    ≈ conc (abs 𝕓 t′) 𝕒
  cong-conc∘abs eq = cong-conc (cong-abs eq) λ ()

  open ≈-Reasoning

  instance
```

```agda
SwapLaws-Abs : SwapLaws (Abs A)
SwapLaws-Abs .cong-swap {f@(abs 𝕩 t)}{g@(abs 𝕪 t′)}{a}{b} (xs
  = a ∷ b ∷ xs , λ x x∉ →
    begin
      conc (⦅ a ↔ b ⦆ f) x
    ≡⟨⟩
      conc (abs (⦅ a ↔ b ⦆ 𝕩) (⦅ a ↔ b ⦆ t)) x
    ≡⟨⟩
      ⦅ x ↔ ⦅ a ↔ b ⦆ 𝕩 ⦆ ⦅ a ↔ b ⦆ t
    ≡˘⟨ cong (λ ◆ → ⦅ ◆ ↔ ⦅ a ↔ b ⦆ 𝕩 ⦆ ⦅ a ↔ b ⦆ t)
          $ swap-noop a b x (λ where 𝟘 → x∉ 𝟘; 𝟙 → x∉ 𝟙) ⟩
      ⦅ ⦅ a ↔ b ⦆ x ↔ ⦅ a ↔ b ⦆ 𝕩 ⦆ ⦅ a ↔ b ⦆ t
    ≈˘⟨ swap-conc _ _ ⟩
      ⦅ a ↔ b ⦆ conc f x
    ≈⟨ cong-swap $ f≈g x (x∉ ∘′ there ∘′ there) ⟩
```

```
      ⦅ a ↔ b ⦆ conc g x
    ≈⟨ swap-conc _ _ ⟩
      ⦅ ⦅ a ↔ b ⦆ x ↔ ⦅ a ↔ b ⦆ y ⦆ ⦅ a ↔ b ⦆ t′
    ≡⟨ cong (λ ◆ → ⦅ ◆ ↔ ⦅ a ↔ b ⦆ y ⦆ ⦅ a ↔ b ⦆ t′)
          $ swap-noop a b x (λ where 𝟘 → x∉ 𝟘; 𝟙 → x∉ 𝟙) ⟩
      ⦅ x ↔ ⦅ a ↔ b ⦆ y ⦆ ⦅ a ↔ b ⦆ t′
    ≡⟨⟩
      conc (abs (⦅ a ↔ b ⦆ y) (⦅ a ↔ b ⦆ t′)) x
    ≡⟨⟩
      conc (⦅ a ↔ b ⦆ g) x
    ∎
SwapLaws-Abs .swap-id {a}{abs x t} =
  begin
    ⦅ a ↔ a ⦆ abs x t
  ≡⟨⟩
```

```
    abs (⟪ a ↔ a ⟫ x) (⟪ a ↔ a ⟫ t)
  ≡⟨ cong (λ ◆ → abs ◆ ((⟪ a ↔ a ⟫ t)) swap-id ⟩
    abs x (⟪ a ↔ a ⟫ t)
  ≈⟨ cong-abs swap-id ⟩
    abs x t
  ∎
SwapLaws-Abs .swap-rev {a}{b}{f@(abs 𝕩 t)} =
  a ∷ b ∷ [] , λ x x∉ →
  begin
    conc (⟪ a ↔ b ⟫ f) x
  ≡⟨⟩
    conc (abs (⟪ a ↔ b ⟫ 𝕩) (⟪ a ↔ b ⟫ t)) x
  ≡⟨ cong (λ ◆ → conc (abs ◆ (⟪ a ↔ b ⟫ t)) x) swap-rev ⟩
    conc (abs (⟪ b ↔ a ⟫ 𝕩) (⟪ a ↔ b ⟫ t)) x
  ≈⟨ cong-abs swap-rev .proj₂ x (λ ()) ⟩
```

```
      conc (abs (⦅ b ↔ a ⦆ 𝕩) (⦅ b ↔ a ⦆ t)) x
    ≡⟨⟩
      conc (⦅ b ↔ a ⦆ f) x
    ∎
  SwapLaws-Abs .swap-sym {a}{b}{f@(abs 𝕩 t)} =
    a ∷ b ∷ [] , λ x x∉ →
    begin
      conc (⦅ a ↔ b ⦆ ⦅ b ↔ a ⦆ f) x
    ≡⟨⟩
      conc (abs (⦅ a ↔ b ⦆ ⦅ b ↔ a ⦆ 𝕩) (⦅ a ↔ b ⦆ ⦅ b ↔ a ⦆ t)) x
    ≡⟨ cong (λ ◆ → conc (abs ◆ (⦅ a ↔ b ⦆ ⦅ b ↔ a ⦆ t)) x) swap-sym ⟩
      conc (abs 𝕩 (⦅ a ↔ b ⦆ ⦅ b ↔ a ⦆ t)) x
    ≈⟨ cong-abs swap-sym .proj₂ x (λ ()) ⟩
      conc (abs 𝕩 t) x
    ≡⟨⟩
```

```
    conc f x
  ∎
SwapLaws-Abs .swap-swap {a}{b}{c}{d}{f@(abs 𝕩 t)} =
  a ∷ b ∷ c ∷ d ∷ [] , λ x x∉ →
  begin
    conc (⦅ a ↔ b ⦆ ⦅ c ↔ d ⦆ f) x
  ≡⟨⟩
    conc (abs (⦅ a ↔ b ⦆ ⦅ c ↔ d ⦆ 𝕩) (⦅ a ↔ b ⦆ ⦅ c ↔ d ⦆ t)) x
  ≡⟨ cong (λ ◆ → conc (abs ◆ (⦅ a ↔ b ⦆ ⦅ c ↔ d ⦆ t)) x) swap-swap ⟩
    conc (abs (⦅ ⦅ a ↔ b ⦆ c ↔ ⦅ a ↔ b ⦆ d ⦆ ⦅ a ↔ b ⦆ 𝕩)
              (⦅ a ↔ b ⦆ ⦅ c ↔ d ⦆ t)) x
  ≈⟨ cong-abs swap-swap .proj₂ x (λ ()) ⟩
    conc (⦅ ⦅ a ↔ b ⦆ c ↔ ⦅ a ↔ b ⦆ d ⦆ ⦅ a ↔ b ⦆ f) x
  ∎
```

```
--   concₓ : Abs A → A
--   concₓ = flip conc 𝕏




--   mor : Abs A —𝔾→ A
--   mor = record { f = concₓ ; equivariant = {!swap-swap!} }
```

Nominal/Abs/Lift.agda

```agda
open import Prelude.Init; open SetAsType
open import Prelude.DecEq
open import Prelude.Setoid

module Nominal.Abs.Lift (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap Atom
open import Nominal.Fun  Atom
open import Nominal.Abs.Base Atom

module _ {A : Type ℓ} {B : Type ℓ′} ⦃ _ : Swap A ⦄ ⦃ _ : Swap B ⦄ where

  theorem→ : Abs (A → B) → (Abs A → Abs B)
  theorem→ (abs 𝕒 f) (abs 𝕓 a) = abs 𝕒 $ swap 𝕓 𝕒 (f a)

  postulate theorem← : (Abs A → Abs B) → Abs (A → B)
  -- theorem← F = abs {!!} (λ a → {!!})
```

```
private variable A : Type ℓ

record Lift (P : Type ℓ → Type ℓ′) : Type (lsuc ℓ ⊔ₗ ℓ′) where
  field lift : P A → P (Abs A)
open Lift ⦃...⦄ public

instance
  -- Lift-Fun : ∀ {B : Type ℓ′} → Lift (λ A → A → B)
  -- Lift-Fun .lift f (abs a x) = {!!}

  Lift-Rel : Lift (λ (A : Type ℓ) → Rel A ℓ′)
  Lift-Rel .lift _~_ = λ where
    (abs _ x) (abs _ y) → x ~ y

-- lift : Rel A ℓ → Rel (Abs A) ℓ
-- (lift _~_) = λ where
```

```
--    -- (abs _ x) (abs _ y) → x ~ y
--    -- (abs 𝕒 x) (abs 𝕓 y) → x ~ swap 𝕓 𝕒 y
--    x y → let 𝕔 = freshAtom in conc x 𝕔 ~ conc y 𝕔
--     where postulate freshAtom : Atom







-- instance
--   Setoid-Abs : ⦃ ISetoid A ⦄ → ISetoid (Abs A)
--   Setoid-Abs = λ where
--     .relℓ → _
--     ._≈_  → lift _≈_
```

Nominal/Abs/Product.agda

```
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.InfEnumerable
open import Prelude.Setoid
-- open import Prelude.Bifunctor
-- open import Prelude.InferenceRules

module Nominal.Abs.Product (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enu

open import Nominal.New     Atom
open import Nominal.Swap    Atom
open import Nominal.Support Atom
open import Nominal.Abs.Base Atom
open import Nominal.Product Atom
open import Nominal.Abs.Support Atom
```

```
module _
  {A : Type ℓ} {B : Type ℓ′}
  ⦃ _ : ISetoid A ⦄ ⦃ _ : ISetoid B ⦄
  ⦃ _ : SetoidLaws A ⦄ ⦃ _ : SetoidLaws B ⦄
  ⦃ _ : Swap A ⦄ ⦃ _ : Swap B ⦄
  ⦃ _ : SwapLaws A ⦄ ⦃ _ : SwapLaws B ⦄ where

  open ≈-Reasoning

  Abs-× : Abs (A × B) → Abs A × Abs B
  Abs-× (abs x (a , b)) = abs x a , abs x b

  module _ ⦃ _ : FinitelySupported A ⦄ ⦃ _ : FinitelySupported B ⦄ whe
    Abs-×˘ : Abs A × Abs B → Abs (A × B)
    Abs-×˘ (t̂ , t̂′) =
```

```
let z = minFresh (supp t̂ ++ supp t̂′) .proj₁
  in abs z (conc t̂ z , conc t̂′ z)
```

Nominal/Abs/Support.agda

```
{-# OPTIONS --allow-unsolved-metas #-}
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.Setoid
open import Prelude.Bifunctor
open import Prelude.InferenceRules
open import Prelude.InfEnumerable

module Nominal.Abs.Support (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enu

open import Nominal.New Atom
open import Nominal.Swap Atom
open import Nominal.Support Atom
open import Nominal.Abs.Base Atom
```

```
module _ {A : Type ℓ}
  ⦃ _ : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄
  ⦃ _ : Swap A ⦄ ⦃ _ : SwapLaws A ⦄ where

  open ≈-Reasoning

  module _ ⦃ _ : ∃FinitelySupported A ⦄ where
    -- abstractions over finitely supported types are themselve
    instance
      ∃FinSupp-Abs : ∃FinitelySupported (Abs A)
      ∃FinSupp-Abs .∀∃fin (abs x t) =
        let xs , p = ∀∃fin t
        in x ∷ xs , λ y z y∉ z∉ →
        begin
          ⦅ z ↔ y ⦆ (abs x t)
        ≡⟨ ⟩
```

```
    -- ⟪ 𝕒 ↔ 𝕓 ⟫ (f 𝕔) ≈ (⟪ 𝕒 ↔ 𝕓 ⟫ f) (⟪ 𝕒 ↔ 𝕓 ⟫ 𝕔)
    abs (⟪ z ↔ y ⟫ x) (⟪ z ↔ y ⟫ t)
  ≡⟨ cong (λ ◆ → abs ◆ (⟪ z ↔ y ⟫ t))
      $ swap-noop z y x (λ where 𝟘 → z∉ 𝟘; 𝟙 → y∉ 𝟘) ⟩
    abs x (⟪ z ↔ y ⟫ t)
  ≈⟨ cong-abs $ p y z (y∉ ∘ there) (z∉ ∘ there) ⟩
    abs x t
  ∎ where open ≈-Reasoning

-- Two ways to fix functoriality:
 -- 1. require that (f : A → A) is equivariant
--   2. ...or that it at least has finite support
mapAbs : Op₁ A → Op₁ (Abs A)
  -- ≈ (A → A) → (Abs A → Abs A)
-- TODO: In order to resolve termination issues (via well-f
```

```
  -- we need a more restrainted version of mapAbs with type:
  -- mapAbs : (x' : Abs A) → (f : (a : A) → a ≺ f → A) → Abs
  -- NB: a generalisation would be to say that the size behav
  --     `mapAbs f` corresponds to that of `f`
  mapAbs f x' =
    let a = ∃fresh-var x' -- TODO: ++ supp?? f
    in abs a (f $ conc x' a)

  freshen : Op₁ (Abs A)
  freshen f@(abs a t) =
    let xs , _ = ∀∃fin f
        b , b∉ = minFresh xs
    in abs b (conc f b)

module _ ⦃ _ : FinitelySupported A ⦄ where
```

```
-- abstractions over finitely supported types are themselve
instance
  FinSupp-Abs : FinitelySupported (Abs A)
  FinSupp-Abs .∀fin t̂@(abs x t)
    with xs , p , ¬p ← ∀fin t
    = xs' , eq , ¬eq
    where
      xs' = filter (¬? ∘ (_≟ x)) xs

      eq : ∀ y z → y ∉ xs' → z ∉ xs' → swap z y t̂ ≈ t̂
      eq y z y∉' z∉'
        with y ≟ x | z ≟ x
      ... | yes refl | yes refl
        = swap-id
      ... | yes refl | no z≢
```

```
rewrite ≟-refl y
  | dec-no (y ≟ z) (≢-sym z≢) .proj₂
=
begin
  abs z (⦅ z ↔ x ⦆ t)
≈⟨ x ∷ xs , (λ w w∉ →
  begin
    conc (abs z (⦅ z ↔ x ⦆ t)) w
  ≡⟨⟩
    ⦅ w ↔ z ⦆ ⦅ z ↔ x ⦆ t
  ≈⟨ swap-swap ⟩
    ⦅ ⦅ w ↔ z ⦆ z ↔ ⦅ w ↔ z ⦆ x ⦆ ⦅ w ↔ z ⦆ t
  ≡⟨ cong (λ ◆ → ⦅ ◆ ↔ ⦅ w ↔ z ⦆ x ⦆ ⦅ w ↔ z ⦆ t)
         $ swapʳ w z ⟩
    ⦅ w ↔ ⦅ w ↔ z ⦆ x ⦆ ⦅ w ↔ z ⦆ t
```

```
  ≡⟨ cong (λ ◆ → ⦅ w ↔ ◆ ⦆ ⦅ w ↔ z ⦆ t)
       $ swap-noop w z x (λ where 𝟘 → w∉ 𝟘; 𝟙 → z≢ refl) ⟩
   ⦅ w ↔ x ⦆ ⦅ w ↔ z ⦆ t
  ≈⟨ cong-swap $ p z w z∉ (w∉ ∘ there) ⟩
   ⦅ w ↔ x ⦆ t
  ≡⟨⟩
   conc (abs x t) w
  ∎) ⟩
  abs x t
 ∎
 where
  z∉ : z ∉ xs
  z∉ z∈ = z∉′ $ ∈-filter⁺ (¬? ∘ (_≟ x)) z∈ z≢
... | no y≢ | yes refl
 rewrite ≟-refl z
```

```
-- abs y (⦅ x ↔ y ⦆ t)
=
begin
  abs y (⦅ x ↔ y ⦆ t)
≈⟨ x ∷ xs , (λ w w∉ →
  begin
    conc (abs y (⦅ x ↔ y ⦆ t)) w
  ≡⟨⟩
    ⦅ w ↔ y ⦆ ⦅ x ↔ y ⦆ t
  ≈⟨ swap-swap ⟩
    ⦅ ⦅ w ↔ y ⦆ x ↔ ⦅ w ↔ y ⦆ y ⦆ ⦅ w ↔ y ⦆ t
  ≡⟨ cong (λ ◆ → ⦅ ⦅ w ↔ y ⦆ x ↔ ◆ ⦆ ⦅ w ↔ y ⦆ t)
        $ swapʳ w y ⟩
    ⦅ ⦅ w ↔ y ⦆ x ↔ w ⦆ ⦅ w ↔ y ⦆ t
  ≡⟨ cong (λ ◆ → ⦅ ◆ ↔ w ⦆ ⦅ w ↔ y ⦆ t)
```

```
          $ swap-noop w y x (λ where 𝟘 → w∉ 𝟘; 𝟙 → y≢ refl) ⟩
        ⦅ x ↔ w ⦆ ⦅ w ↔ y ⦆ t
      ≈⟨ swap-rev ⟩
        ⦅ w ↔ x ⦆ ⦅ w ↔ y ⦆ t
      ≈⟨ cong-swap $ p y w y∉ (w∉ ∘ there) ⟩
        ⦅ w ↔ x ⦆ t
      ≡⟨⟩
        conc (abs x t) w
      ∎) ⟩
      abs x t
    ∎
    where
      y∉ : y ∉ xs
      y∉ y∈ = y∉′ $ ∈-filter⁺ (¬? ∘ (_≟ x)) y∈ y≢
... | no y≢ | no z≢
```

```
    rewrite swap-noop z y x (λ where 𝟘 → z≢ refl; 𝟙 → y≢ refl)
    = cong-abs $ p y z y∉ z∉
    where
      y∉ : y ∉ xs
      y∉ y∈ = y∉′ $ ∈-filter⁺ (¬? ∘ (_≟ x)) y∈ y≢

      z∉ : z ∉ xs
      z∉ z∈ = z∉′ $ ∈-filter⁺ (¬? ∘ (_≟ x)) z∈ z≢

  ¬eq : ∀ y z → y ∈ xs′ → z ∉ xs′ → swap z y t̂ ≉ t̂
  ¬eq y z y∈′ z∉′
    with y∈ , y≢ ← ∈-filter⁻ (¬? ∘ (_≟ x)) {xs = xs} y∈′
{-
begin
  swap z y t̂
≡⟨ ⟩
```

```
  abs (swap z y x) (swap z y t)
≢⟨ ? ⟩
  abs x t
≡⟨⟩
  t̂

-}
           with z ≟ x
          ... | yes refl
           -- abs y (⦅ x ↔ y ⦆ t) ≢ abs x t
{-
begin
  swap x y t̂
≡⟨⟩
  abs (swap x y x) (swap x y t)
```

```
≡⟨ swapˡ ⟩
  abs y (swap x y t)
≉⟨ ? ⟩
  abs x t
≡⟨⟩
  t̂
█
-}
          = {!¬p y z y? !}
        ... | no z≢
          rewrite dec-no (z ≟ x) z≢ .proj₂
          -- abs x (⦅ z ↔ y ⦆ t) ≉ abs x t
{-
¬p y z y∈ (∉-∷⁺ z≢ z∉) : swap z y t ≉ t
```

```
begin
  swap z y t̂
≡⟨⟩
  abs (swap z y x) (swap z y t)
≡⟨ swap-noop z y z z≢ y≢ ⟩
  abs x (swap z y t)
≢⟨ ? ⟩
  abs x t
≡⟨⟩
  t̂

-}
               = {!¬p y z y? (?-?⁺ z? z? ?)!}
```

Nominal.agda

```agda
open import Prelude.Init; open SetAsType
open import Prelude.DecEq

module Nominal (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.New Atom public
open import Nominal.Swap Atom public
open import Nominal.Perm Atom public
open import Nominal.Support Atom public

open import Nominal.Fun Atom public
open import Nominal.Abs Atom public

-- open import Nominal.Product Atom public
-- [BUG]
-- Don't export this together with Abs!
```

```
-- Otherwise instance resolution fails for no reason
-- as demonstrated by the example imported below.
open import Nominal.ImportIssue
```

Nominal/Fun.agda

```
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.General
open import Prelude.DecEq
open import Prelude.Decidable
open import Prelude.Setoid
open import Prelude.InferenceRules
open import Prelude.InfEnumerable

module Nominal.Fun (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap Atom
open import Nominal.Support Atom

module _ {A : Type ℓ} {B : Type ℓ′} ⦃ _ : Swap A ⦄ ⦃ _ : Swap B ⦄ where
```

```
open ≈-Reasoning

instance
  Swap-Fun : Swap (A → B)
  Swap-Fun .swap a b f = swap a b ∘ f ∘ swap a b

  Setoid-Fun : ⦃ ISetoid B ⦄ → ISetoid (A → B)
  Setoid-Fun = λ where
    .relℓ → ℓ ⊔ₗ relℓ {A = B}
    ._≈_  f g → ∀ x → f x ≈ g x
    -- ._≈_  f g → ∀ x y → x ≈ y → f x ≈ g y

  SetoidLaws-Fun :
    ⦃ _ : ISetoid B ⦄ → ⦃ SetoidLaws B ⦄
    → SetoidLaws (A → B)
  SetoidLaws-Fun .isEquivalence = record
```

```
  { refl  = λ {f} x → ≈-refl
  ; sym   = λ f~g x → ≈-sym (f~g x)
  ; trans = λ f~g g~h x → ≈-trans (f~g x) (g~h x)
  }

SwapLaws-Fun :
  ⦃ _ : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄ ⦃ _ : CongSetoid A ⦄ ⦃ _ : Sw
  ⦃ _ : ISetoid B ⦄ ⦃ _ : SetoidLaws B ⦄ ⦃ _ : SwapLaws B ⦄
  → SwapLaws (A → B)
SwapLaws-Fun .cong-swap {f}{g}{a}{b} f≗g x =
-- ∀ {f g : A → B} → x ≈ y → ⦅ 𝕒 ↔ 𝕓 ⦆ f ≈ ⦅ 𝕒 ↔ 𝕓 ⦆ g
  cong-swap (f≗g _)
SwapLaws-Fun .swap-id {a}{f} x =
-- ∀ {f : A → B} → ⦅ 𝕒 ↔ 𝕒 ⦆ f ≈ f
  begin
```

```
    ⟪ a ↔ a ⟫ (f (⟪ a ↔ a ⟫ x))
  ≈⟨ swap-id ⟩
    f (⟪ a ↔ a ⟫ x)
  ≈⟨ ≈-cong f swap-id ⟩
    f x
  ∎
SwapLaws-Fun .swap-rev {a}{b}{f} x =
-- ∀ {f : A → B} → ⟪ 𝕒 ↔ 𝕓 ⟫ f ≈ ⟪ 𝕓 ↔ 𝕒 ⟫ f
  begin
    (⟪ a ↔ b ⟫ f) x
  ≡⟨⟩
    ⟪ a ↔ b ⟫ (f $ ⟪ a ↔ b ⟫ x)
  ≈⟨ cong-swap $ ≈-cong f swap-rev ⟩
    ⟪ a ↔ b ⟫ (f $ ⟪ b ↔ a ⟫ x)
  ≈⟨ swap-rev ⟩
```

```
    ⦇ b ↔ a ⦈ (f $ ⦇ b ↔ a ⦈ x)
  ≡⟨⟩
    (⦇ b ↔ a ⦈ f) x
  ∎
SwapLaws-Fun .swap-sym {a}{b}{f} x =
-- ∀ {f : A → B} → ⦇ 𝕒 ↔ 𝕓 ⦈ ⦇ 𝕓 ↔ 𝕒 ⦈ f ≈ f
  begin
    (⦇ a ↔ b ⦈ ⦇ b ↔ a ⦈ f) x
  ≡⟨⟩
    ⦇ a ↔ b ⦈ ⦇ b ↔ a ⦈ (f $ ⦇ b ↔ a ⦈ ⦇ a ↔ b ⦈ x)
  ≈⟨ cong-swap $ cong-swap $ ≈-cong f swap-sym ⟩
    ⦇ a ↔ b ⦈ ⦇ b ↔ a ⦈ (f x)
  ≈⟨ swap-sym ⟩
    f x
  ∎
```

```
SwapLaws-Fun .swap-swap {𝕒 = a}{b}{c}{d}{f} x =
-- ∀ {f : A → B} → ⟪ 𝕒 ↔ 𝕓 ⟫ ⟪ 𝕔 ↔ 𝕕 ⟫ f
--                   ≈ ⟪ ⟪ 𝕒 ↔ 𝕓 ⟫ 𝕔 ↔ ⟪ 𝕒 ↔ 𝕓 ⟫ 𝕕 ⟫ ⟪ 𝕒 ↔ 𝕓 ⟫
  begin
    (⟪ a ↔ b ⟫ ⟪ c ↔ d ⟫ f) x
  ≡⟨⟩
    ⟪ a ↔ b ⟫ ⟪ c ↔ d ⟫ (f $ ⟪ c ↔ d ⟫ ⟪ a ↔ b ⟫ x)
  ≈⟨ swap-swap ⟩
    ⟪ ⟪ a ↔ b ⟫ c ↔ ⟪ a ↔ b ⟫ d ⟫ ⟪ a ↔ b ⟫
      (f $ ⟪ c ↔ d ⟫ ⟪ a ↔ b ⟫ x)
  --                     ↑ NB: note the change of ordering on swa
  ≈⟨ cong-swap $ cong-swap $ ≈-cong f
   $ begin
      ⟪ c ↔ d ⟫ ⟪ a ↔ b ⟫ x
    ≡˘⟨ cong (λ ◆ → ⟪ c ↔ ◆ ⟫ ⟪ a ↔ b ⟫ x) swap-sym′ ⟩
```

```
      ⟪ c ↔ ⟪ a ↔ b ⟫ ⟪ a ↔ b ⟫ d ⟫ ⟪ a ↔ b ⟫ x
    ≡˘⟨ cong (λ ◆ → ⟪ ◆ ↔ ⟪ a ↔ b ⟫ ⟪ a ↔ b ⟫ d ⟫ ⟪ a ↔ b ⟫ x) swap-sym’
      ⟪ ⟪ a ↔ b ⟫ ⟪ a ↔ b ⟫ c ↔ ⟪ a ↔ b ⟫ ⟪ a ↔ b ⟫ d ⟫ ⟪ a ↔ b ⟫ x
    ≈˘⟨ swap-swap ⟩
      ⟪ a ↔ b ⟫ ⟪ ⟪ a ↔ b ⟫ c ↔ ⟪ a ↔ b ⟫ d ⟫ x
    ∎
   ⟩
    ⟪ ⟪ a ↔ b ⟫ c ↔ ⟪ a ↔ b ⟫ d ⟫ ⟪ a ↔ b ⟫
      (f $ ⟪ a ↔ b ⟫ ⟪ ⟪ a ↔ b ⟫ c ↔ ⟪ a ↔ b ⟫ d ⟫ x)
  ≡⟨⟩
    (⟪ ⟪ a ↔ b ⟫ c ↔ ⟪ a ↔ b ⟫ d ⟫ ⟪ a ↔ b ⟫ f) x
  ∎

-- NB: swapping takes the conjugation action on functions
module _
```

```
{{ _ : ISetoid A }} {{ _ : SetoidLaws A }} {{ _ : SwapLaws A }} {{ _ : CongSe
{{ _ : ISetoid B }} {{ _ : SetoidLaws B }} {{ _ : SwapLaws B }}
where
conj : ∀ {𝕒 𝕓 : Atom} (f : A → B) (x : A) →
  (swap 𝕒 𝕓 f) x ≈ swap 𝕒 𝕓 (f $ swap 𝕒 𝕓 x)
conj {𝕒} {𝕓} f x =
  begin
      (swap 𝕒 𝕓 f) x
  ≡⟨⟩
      (swap 𝕒 𝕓 ∘ f ∘ swap 𝕒 𝕓) x
  ≡⟨⟩
      swap 𝕒 𝕓 (f $ swap 𝕒 𝕓 x)
  ≈˘⟨ cong-swap $ ≈-cong f swap-sym′ ⟩
      swap 𝕒 𝕓 (f $ swap 𝕒 𝕓 $ swap 𝕒 𝕓 $ swap 𝕒 𝕓 x)
  ≡⟨⟩
```

```
      (swap 𝕒 𝕓 ∘ f ∘ swap 𝕒 𝕓) (swap 𝕒 𝕓 $ swap 𝕒 𝕓 x)
    ≡⟨⟩
      (swap 𝕒 𝕓 f) (swap 𝕒 𝕓 $ swap 𝕒 𝕓 x)
    ≈˘⟨ distr-f 𝕒 𝕓 ⟩
      swap 𝕒 𝕓 (f $ swap 𝕒 𝕓 x)
    ∎ where distr-f = swap↔ f

private
  postulate
    𝕒 𝕓 : Atom
    𝕒≠𝕓 : 𝕒 ≠ 𝕓

  unquoteDecl Swap-Maybe = DERIVE Swap [ quote Maybe , Swap-Maybe ]

  justAtom : Atom → Maybe Atom
  justAtom n =
```

```
  if n == 𝕒 then
    just 𝕒
  else
    nothing

justAtom′ : Atom → Maybe Atom
justAtom′ = ⟅ 𝕒 ↔ 𝕓 ⟆ justAtom

_ : justAtom 𝕒 ≡ just 𝕒
_ rewrite ≟-refl 𝕒 = refl

_ : justAtom 𝕓 ≡ nothing
_ rewrite dec-no (𝕓 ≟ 𝕒) (≢-sym 𝕒≢𝕓) .proj₂ = refl

_ : justAtom′ 𝕒 ≡ nothing
_ rewrite dec-no (𝕒 ≟ 𝕓) 𝕒≢𝕓 .proj₂
```

```
             | ≟-refl 𝕒
             | dec-no (𝕓 ≟ 𝕒) (≠-sym 𝕒≠𝕓) .proj₂
             = refl

  _ : justAtom′ 𝕓 ≡ just 𝕓
  _ rewrite ≟-refl 𝕓
             | dec-no (𝕓 ≟ 𝕒) (≠-sym 𝕒≠𝕓) .proj₂
             | ≟-refl 𝕒
             | ≟-refl 𝕒
             = refl

module _
  ⦃ _ : Enumerable∞ Atom ⦄
  {A : Type ℓ} ⦃ _ : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄
  ⦃ _ : Swap A ⦄ ⦃ _ : SwapLaws A ⦄
  where
```

```
-- ∗ in the case of _→_, Equivariant′ is equivalent to Equi
equivariant-equiv : ∀ {f : A → A} →
  Equivariant f
  ═════════════
  Equivariant′ f
equivariant-equiv {f = f} = ↝ , ↜
  where
    open ≈-Reasoning

    ↝ : Equivariant f
        ─────────────
        Equivariant′ f
    ↝ equiv-f = fin-f , refl
      where
        fin-f : FinSupp f
```

```
fin-f = [] , (λ x y _ _ a →
  begin
    ⦅ y ↔ x ⦆ (f $ ⦅ y ↔ x ⦆ a)
  ≈˘⟨ cong-swap $ equiv-f _ _ ⟩
    ⦅ y ↔ x ⦆ ⦅ y ↔ x ⦆ f a
  ≈⟨ swap-sym′ ⟩
    f a
  ∎) , λ _ _ ()

↜ : Equivariant′ f
    ────────────────
    Equivariant f
↜ (fin-f , refl) a b {x} =
  begin
    ⦅ a ↔ b ⦆ f x
```

```
        ≈˘⟨ cong-swap $ fin-f .proj₂ .proj₁ _ _ (λ ()) (λ ()) _ ⟩
          ⟪ a ↔ b ⟫ ⟪ a ↔ b ⟫ f (⟪ a ↔ b ⟫ x)
        ≈⟨ swap-sym′ ⟩
          f (⟪ a ↔ b ⟫ x)
        ∎

private
  f′ : A → A
  f′ = id

  suppF′ = Atoms ∋ []

  g′ : A → A
  g′ x = x

  f≗g : f′ ≗ g′
```

```
f≗g _ = refl

f≈g : f′ ≈ g′
f≈g _ = ≈-refl

∃fin-f : ∃FinSupp f′
∃fin-f = suppF′ , λ _ _ _ _ _ → swap-sym′

fin-f : FinSupp f′
fin-f = suppF′ , (λ _ _ _ _ _ → swap-sym′) , (λ _ _ ())

equiv-f : Equivariant f′
equiv-f _ _ = ≈-refl

equiv-f′ : Equivariant′ f′
equiv-f′ = fin-f , refl
```

```
instance
  Setoid-Bool : ISetoid Bool
  Setoid-Bool = λ where
      .relℓ → 0ℓ
      ._≈_ → _≡_

  SetoidLaws-Bool : SetoidLaws Bool
  SetoidLaws-Bool .isEquivalence = PropEq.isEquivalence

postulate x y : Atom

f : Atom → Bool
f z = (z == x) ∨ (z == y)
suppF = List Atom ∋ x ∷ y ∷ []
-- fresh f = False
```

```
finF : ∃FinSupp f
finF = -, go
  where
    ∀x∉suppF : ∀ {z} → z ∉ suppF → f z ≡ false
    ∀x∉suppF {z} z∉ with z ≟ x
    ... | yes refl = ⊥-elim $ z∉ $ here refl
    ... | no _ with z ≟ y
    ... | yes refl = ⊥-elim $ z∉ $ there $′ here refl
    ... | no _ = refl

    go : ∀ 𝕒 𝕓 → 𝕒 ∉ suppF → 𝕓 ∉ suppF → f ∘ swap 𝕓 𝕒 ≗ f
    go 𝕒 𝕓 𝕒∉ 𝕓∉ z with z ≟ 𝕓
    ... | yes refl rewrite ∀x∉suppF 𝕒∉ | ∀x∉suppF 𝕓∉ = refl
    ... | no _ with z ≟ 𝕒
    ... | yes refl rewrite ∀x∉suppF 𝕒∉ | ∀x∉suppF 𝕓∉ = refl
```

```
      ... | no _ = refl

  _ = finF .proj₁ ≡ suppF
    ∋ refl

  g : Atom → Bool
  g z = (z ≠ x) ∧ (z ≠ y)
  suppG = List Atom ∋ x ∷ y ∷ []
  -- fresh g = True
  -- NB: g is infinite, but has finite support!

  finG : ∃FinSupp g
  finG = -, go
    where
      ∀x∉suppG : ∀ {z} → z ∉ suppG → g z ≡ true
      ∀x∉suppG {z} z∉ with z ≟ x
```

```
    ... | yes refl = ⊥-elim $ z∉ $ here refl
    ... | no _ with z ≟ y
    ... | yes refl = ⊥-elim $ z∉ $ there $′ here refl
    ... | no _ = refl

    go : ∀ 𝕒 𝕓 → 𝕒 ∉ suppG → 𝕓 ∉ suppG → g ∘ swap 𝕓 𝕒 ≗ g
    go 𝕒 𝕓 𝕒∉ 𝕓∉ z with z ≟ 𝕓
    ... | yes refl rewrite ∀x∉suppG 𝕒∉ | ∀x∉suppG 𝕓∉ = refl
    ... | no _ with z ≟ 𝕒
    ... | yes refl rewrite ∀x∉suppG 𝕒∉ | ∀x∉suppG 𝕓∉ = refl
    ... | no _ = refl

-- TODO: example where _≗_ is not the proper notion of equa
-- module _ ⦃ _ : Toℕ Atom ⦄ where
--   h : Atom → Bool
--   h z = even? (toℕ z)
```

```
  --   -- ∄ supp h ⇔ ∄ fresh h
```

```
-- Find the non-finSupp swappable example.
-- • ZFA ⇝ ZFA+choice
-- • the set of all total orderings on atoms
-- (empty support on the outside, infinite support inside ea
-- • FOL: ultra-filter construction
```

Nominal/ImportIssue.agda

```
{-# OPTIONS --allow-unsolved-metas #-}
open import Prelude.Init; open SetAsType
open import Prelude.DecEq
open import Prelude.Setoid

module Nominal.ImportIssue (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap Atom
-- importing both abstractions and products confuses instance
open import Nominal.Abs  Atom
open import Nominal.Product Atom

module _ {A : Type} ⦃ _ : ISetoid A ⦄ ⦃ _ : Swap A ⦄ where

  private variable 𝕒 𝕓 : Atom

  _ : ∀ {x y : Atom × Atom} → x ≈ y → ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ ⦅ 𝕒 ↔ 𝕓 ⦆ y
```

```
_ = cong-swap
```

```
_ : ∀ {x y : Abs Atom} → x ≈ y → ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ ⦅ 𝕒 ↔ 𝕓 ⦆ y
_ = cong-swap
-- _ = cong-swap {A = Abs _} -- this works
```

Nominal/New.agda

```
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq

module Nominal.New (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

-- ** The И "new" quantifier.
И : Pred (Pred Atom ℓ) ℓ
-- И "for all except finitely many"
И φ = ∃ λ (xs : List Atom) → (∀ y → y ∉ xs → φ y)
--                                    ∀ y ∉ xs → φ y
-- ƃ for "generous", i.e. "for infinitely many"
-- ƃ φ = ∀ (xs : List Atom) → (∃ y → (y ∈ xs) → φ y)
--                                    ∃ y ∉ xs → φ y
-- NB: И is *self-dual*
-- И̅ = ƃ
```

```
Иᘔ : Pred (Atom → Atom → Type ℓ) ℓ
Иᘔ φ = ∃ λ (xs : List Atom) →
  (∀ y z → y ∉ xs → z ∈ xs → φ y z)

-- И∗ : Pred (Pred (List Atom) ℓ) ℓ
-- И∗ φ = ∃ λ (xs : List Atom) → (∀ ys → All (_∉ xs) ys → φ ys

И^_ : (n : ℕ) → Pred (Pred (Vec Atom n) ℓ) ℓ
(И^ n) φ = ∃ λ (xs : List Atom) → (∀ ys → V.All.All (_∉ xs) ys → φ ys)

И² : Pred (Atom → Atom → Type ℓ) ℓ
-- И² φ = (И^ 2) λ where (x ∷ y ∷ []) → φ x y
И² φ = ∃ λ (xs : List Atom) → (∀ y z → y ∉ xs → z ∉ xs → φ y z)

И³ : Pred (Atom → Atom → Atom → Type ℓ) ℓ
-- И³ φ = (И^ 3) λ where (x ∷ y ∷ z ∷ []) → φ x y z
```

```
И³ φ = ∃ λ (xs : List Atom) → (∀ y z w → y ∉ xs → z ∉ xs → w ∉ xs → φ y z w)
```

```
-- ** the co-finite construction leads to issues with universe
-- open import Cofinite.agda
-- И : Pred (Pred Atom ℓ) (lsuc ℓ)
-- И P = powᶜᵒᶠ P
```

Nominal/Perm.agda

```
open import Prelude.Init; open SetAsType
open import Prelude.DecEq
open import Prelude.Semigroup
open import Prelude.Monoid
open import Prelude.Group
open import Prelude.Setoid

module Nominal.Perm (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap Atom

-- ** permutations
Perm = Atom × Atom
Perms = List Perm
-- SwapList implements Perms
-- ???      implements Perms
```

```
module _ {ℓ} {A : Type ℓ} ⦃ _ : Swap A ⦄ where

  permute : Perm → A → A
  permute = uncurry swap

  permute∗ : Perms → A → A
  permute∗ = chain ∘ map permute
    where chain = foldr _∘′_ id

  instance
    Setoid-Perms : ISetoid Perms
    Setoid-Perms = λ where
      .relℓ → ℓ
      ._≈_ → _≗_ on permute∗

    Semigroup-Perms : Semigroup Perms
```

```
  Semigroup-Perms ._◇_ = _++_

  -- SemigroupLaws-Perms : SemigroupLaws≡ Perms
  -- SemigroupLaws-Perms = record { ◇-comm = {!◇-comm!} ; ◇-a

  Monoid-Perms : Monoid Perms
  Monoid-Perms .ε = []

  MonoidLaws-Perms : MonoidLaws≡ Perms
  MonoidLaws-Perms = MonoidLaws-List

  Group-Perms : Group Perms
  Group-Perms ._⁻¹ = L.reverse ∘ map Product.swap

{-
   GroupLaws-Perms : GroupLaws Perms _≈_
   GroupLaws-Perms = record {inverse = invˡ , invʳ; ⁻¹-cong =
```

```
where
  open Alg _≈_
  -- open Group Group-Perms

  invˡ : LeftInverse [] _⁻¹ _++_
  invˡ [] = λ _ → refl
  invˡ (p ∷ ps)
    = {!!}

  -- rewrite invˡ ps x = {!!}

  invʳ : RightInverse [] _⁻¹ _++_
  invʳ = {!!}

  inv-cong : Congruent₁ _⁻¹
```

```
        inv-cong = {!!}
-}

 module _ ⦃ setoidA : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄ ⦃ _ : SwapLaws A

  open Actionˡ

  swaps-++ : ∀ (ps ps′ : Perms) {x : A} →
    swaps (ps ++ ps′) x ≈ swaps ps (swaps ps′ x)
  swaps-++ [] _ = ≈-refl
  swaps-++ (_ ∷ ps) _ = cong-swap $ swaps-++ ps _

  Perms-Action : Actionˡ Perms A
  Perms-Action = λ where
    ._·_ → swaps
    .identity → ≈-refl
```

```
  .compatibility {ps}{ps′} → ≈-sym $ swaps-++ ps ps′

instance
  Perms-GSet : GSet Perms A
  Perms-GSet .action = Perms-Action

Perms-GSet′ : GSet′ Perms
Perms-GSet′ = λ where
  .ℓₓ → ℓ
  .X → A
  .setoidX → setoidA
  .action′ → Perms-Action

open GSet-Morphisms Perms public renaming (equivariant to gset-
-- equivariant maps betweens G-sets X and Y are denoted X —
```

Nominal/Product.agda

```
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.Lists.Membership
open import Prelude.Lists.Dec
open import Prelude.General
open import Prelude.DecEq
open import Prelude.Decidable
open import Prelude.Setoid
open import Prelude.InfEnumerable

module Nominal.Product (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap Atom
open import Nominal.Support Atom

module _
```

```
{A : Type ℓ} {B : Type ℓ′}
⦃ _ : ISetoid A ⦄ ⦃ _ : ISetoid B ⦄
⦃ _ : SetoidLaws A ⦄ ⦃ _ : SetoidLaws B ⦄
⦃ _ : Swap A ⦄ ⦃ _ : Swap B ⦄
⦃ _ : SwapLaws A ⦄ ⦃ _ : SwapLaws B ⦄ where

open ≈-Reasoning

instance
  Setoid-× : ISetoid (A × B)
  Setoid-× = λ where
    .relℓ → _
    ._≈_ (a , b) (a′ , b′) → (a ≈ a′) × (b ≈ b′)

  SetoidLaws-× : SetoidLaws (A × B)
  SetoidLaws-× .isEquivalence = record
```

```
    { refl  = ≈-refl , ≈-refl
    ; sym   = λ (i , j) → ≈-sym i , ≈-sym j
    ; trans = λ (i , j) (k , l) → ≈-trans i k , ≈-trans j l
    }

  SwapLaws-× : SwapLaws (A × B)
  SwapLaws-× = record
    { cong-swap = λ (x , y) → cong-swap x , cong-swap y
    ; swap-id = swap-id , swap-id
    ; swap-rev = swap-rev , swap-rev
    ; swap-sym = swap-sym , swap-sym
    ; swap-swap = swap-swap , swap-swap
    }

module _ ⦃ _ : Enumerable∞ Atom ⦄ where
  instance
```

```
∃FinSupp-× : ⦃ ∃FinitelySupported A ⦄
           → ⦃ ∃FinitelySupported B ⦄
           → ∃FinitelySupported (A × B)
∃FinSupp-× .∀∃fin (a , b) =
  let xs , p = ∀∃fin a
      ys , q = ∀∃fin b
  in xs ++ ys , λ y z y∉ z∉ →
      p y z (y∉ ∘ ∈-++⁺ˡ) (z∉ ∘ ∈-++⁺ˡ)
    , q y z (y∉ ∘ ∈-++⁺ʳ _) (z∉ ∘ ∈-++⁺ʳ _)

FinSupp-× : ⦃ FinitelySupported A ⦄
          → ⦃ FinitelySupported B ⦄
          → FinitelySupported (A × B)
FinSupp-× .∀fin (a , b) =
  let xs , p , ¬p = ∀fin a
```

```
          ys , q , ¬q = ∀fin b
      in nub (xs ++ ys)
        , (λ y z y∉ z∉ →
            ( p y z (y∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ˡ) (z∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ˡ)
            , q y z (y∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ʳ xs) (z∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ʳ xs)
            )
          )
        , λ y z y∈′ z∉ (p , q) →
          let z∉ˡ , z∉ʳ = ∉-++⁻ $ ∉-nub⁻ z∉
          in case ∈-++⁻ xs $ ∈-nub⁻ y∈′ of λ where
            (inj₁ y∈) → ¬p y z y∈ z∉ˡ p
            (inj₂ y∈) → ¬q y z y∈ z∉ʳ q

  private
    postulate
```

```
  𝕒 𝕓 : Atom
  𝕒≢𝕓 : 𝕒 ≢ 𝕓

unquoteDecl Swap-Maybe = DERIVE Swap [ quote Maybe , Swap-Maybe ]

justAtom : Atom × Maybe Atom
justAtom = 𝕒 , just 𝕓

justAtom′ : Atom × Maybe Atom
justAtom′ = ⦅ 𝕒 ↔ 𝕓 ⦆ justAtom

_ = justAtom .proj₁ ≡ 𝕒
  ∋ refl

_ = justAtom .proj₂ ≡ just 𝕓
  ∋ refl
```

```
_ : justAtom′ .proj₁ ≡ 𝕓
_ rewrite dec-no (𝕒 ≟ 𝕓) 𝕒≢𝕓 .proj₂
      | ≟-refl 𝕒
      = refl
```

```
_ : justAtom′ .proj₂ ≡ just 𝕒
_ rewrite dec-no (𝕓 ≟ 𝕒) (≢-sym 𝕒≢𝕓) .proj₂
      | ≟-refl 𝕓
      = refl
```

Nominal/Support.agda

```
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.Setoid
open import Prelude.InfEnumerable
open import Prelude.InferenceRules

module Nominal.Support (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enumera

open import Nominal.New Atom
open import Nominal.Swap Atom

freshAtom : Atoms → Atom
freshAtom = proj₁ ∘ minFresh

freshAtom∉ : ∀ {xs : Atoms} → freshAtom xs ∉ xs
```

```
freshAtom∉ {xs} = minFresh xs .proj₂

private variable A : Type ℓ; B : Type ℓ′

module _ ⦃ _ : Swap A ⦄ ⦃ _ : ISetoid A ⦄ where

  ∃FinSupp FinSupp ∃Equivariant′ Equivariant′ : Pred A _

  -- NB: this is an over-approximation!
  -- e.g. ∃supp (ƛ x ⇒ x) = {x}
  ∃FinSupp x = И² λ 𝕒 𝕓 → swap 𝕓 𝕒 x ≈ x

  -- ** a proper notion of support
  -- e.g. in λ-calculus this would correspond to the free vari
  FinSupp a = ∃ λ (xs : Atoms) →
    (∀ x y → x ∉ xs → y ∉ xs → swap y x a ≈ a)
    ×
```

```
    (∀ x y → x ∈ xs → y ∉ xs → swap y x a ≉ a)

  -- alternative definition of equivariance based on (finite) s
  --  * equivariant(x) := supp(x) = ∅
  ∃Equivariant′ x = ∃ λ (fin-x : ∃FinSupp x) → fin-x .proj₁ ≡ []
  Equivariant′ x = ∃ λ (fin-x : FinSupp x) → fin-x .proj₁ ≡ []

-- counter-example: a function with infinite support
-- e.g. λ x → (x == 𝕒) ∨ (x == 𝕓)

record ∃FinitelySupported (A : Type ℓ)
  ⦃ _ : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄
  ⦃ _ : Swap A ⦄ ⦃ _ : SwapLaws A ⦄ : Typeω
  where

  field ∀∃fin : Unary.Universal ∃FinSupp
```

```
∃supp : A → Atoms
∃supp = proj₁ ∘ ∀∃fin

_•∃supp = ∃supp

-- TODO: extract minimal support
--   i.e. filter out elements of `supp` that already satisfy
-- module _ ⦃ _ : IDecSetoid A ⦄ where
--   minSupp : A → Atoms
--   minSupp a =
--     let xs , P = ∀fin a
--     in  filter ? xs
--     ?
-- NB: doesn't hold in general ⇒ leads to a solution to the 
-- TODO: find a characterization of this decidable sub-space
```

```
∃fresh∉ : (a : A) → ∃ (_∉ ∃supp a)
∃fresh∉ = minFresh ∘ ∃supp
-- NB: optimized fresh that generates the *least* element

∃fresh-var : A → Atom
∃fresh-var = proj₁ ∘ ∃fresh∉

swap-∃fresh : ∀ {𝕒 𝕓} (x : A) →
  • 𝕒 ∉ ∃supp x
  • 𝕓 ∉ ∃supp x
  ────────────────
  ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ x
swap-∃fresh x = flip (∀∃fin x .proj₂ _ _)

{-
```

```
supp-swap : ∀ {𝕒 𝕓} (t : A) → supp (swap 𝕒 𝕓 t) ⊆ 𝕒 ∷ 𝕓 ∷ t
-- ≡ swap 𝕒 𝕓 (supp t) -- [swap 𝕒 𝕓 x₁, swap 𝕒 𝕓 x₂, ...]
supp-swap {x}{a}{b} x∉ = ?

swap-∉ : ∀ {x 𝕒 𝕓} (t : A) → x ∉ supp t → swap 𝕒 𝕓 x ∉ supp
-- TODO: add hypothesis `x ∉ [a, b]`
swap-∉ {x}{a}{b} x∉
  with x ≟ a
... | yes refl
  -- b ∉ supp (swap a b t)
  = ?
... | no x≢a
  with x ≟ b
... | yes refl
  -- a ∉ supp (swap a b t)
```

```
      = ?
    ... | no x≢b
      -- x ∉ supp (swap a b t)
      = ?
-}
open ∃FinitelySupported ⦃...⦄ public

instance
  ∃FinSupp-Atom : ∃FinitelySupported Atom
  ∃FinSupp-Atom .∀∃fin 𝕒 = [ 𝕒 ] , λ _ _ y∉ z∉ →
    swap-noop _ _ _ λ where 𝟘 → z∉ 𝟘; 𝟙 → y∉ 𝟘

-- TODO: generalize this to more complex types than Atom (c.f.
∃supp-swap-atom : ∀ {𝕒 𝕓} (t : Atom) → ∃supp (swap 𝕒 𝕓 t) ⊆ 𝕒 ∷ 𝕓 ∷ t ∷ [
-- supp (swap 𝕒 𝕓 t) ≡ swap 𝕒 𝕓 (supp t)
∃supp-swap-atom {a}{b} t
```

```
  with t ≟ a
... | yes refl = λ where 𝟘 → 𝟙
... | no _
  with t ≟ b
... | yes refl = λ where 𝟘 → 𝟘
... | no _ = λ where 𝟘 → 𝟚

record FinitelySupported (A : Type ℓ)
  ⦃ _ : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄
  ⦃ _ : Swap A ⦄ ⦃ _ : SwapLaws A ⦄ : Typeω
  where

  field ∀fin : Unary.Universal FinSupp

  supp : A → Atoms
  supp = proj₁ ∘ ∀fin
```

```
_•supp = supp

infix 4 _♯_
_♯_ : Atom → A → Type _
𝕒 ♯ x = 𝕒 ∉ supp x

fresh∉-min : (a : A) → ∃ (_∉ supp a)
fresh∉-min = fresh ∘ supp
-- NB: optimized fresh that generates the *least* element

fresh-var-min : A → Atom
fresh-var-min = proj₁ ∘ fresh∉-min

swap-fresh-min : ∀ {𝕒 𝕓} (x : A) →
  • 𝕒 ∉ supp x
  • 𝕓 ∉ supp x
```

```
    ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ x
swap-fresh-min x = flip (∀fin x .proj₂ .proj₁ _ _)

∃fresh : ∀ (x : A) → ∃ λ 𝕒 → ∃ λ 𝕓 →
    (𝕒 ♯ x)
  × (𝕓 ♯ x)
  × (swap 𝕓 𝕒 x ≈ x)
∃fresh x =
  let xs , swap≈ , swap≉ = ∀fin x
      -- (a ∷ b ∷ [] , a∉ ∷ b∉ ∷ []) = (fresh^ 2) xs
      a , a∉ = minFresh xs
      b , b∉ = minFresh xs
  in a , b , a∉ , b∉ , swap≈ a b a∉ b∉

-- TODO: meta-programming tactic `fresh-in-context` (big sis
```

```
  -- NB: these tactics correspond to two fundamental axioms/no
  -- (c.f. EZFA)

open FinitelySupported ⦃...⦄ public

instance
  FinSupp-Atom : FinitelySupported Atom
  FinSupp-Atom .∀fin 𝕒 = [ 𝕒 ] , eq , ¬eq
    where
      eq : ∀ x y → x ∉ [ 𝕒 ] → y ∉ [ 𝕒 ] → swap y x 𝕒 ≈ 𝕒
      eq _ _ x∉ y∉ = swap-noop _ _ _ λ where 𝟘 → y∉ 𝟘; 𝟙 → x∉ 𝟘

      ¬eq : ∀ x y → x ∈ [ 𝕒 ] → y ∉ [ 𝕒 ] → swap y x 𝕒 ≉ 𝕒
      ¬eq _ y 𝟘 y∉
        rewrite ≟-refl 𝕒 | ≟-refl y
        with 𝕒 ≟ y
```

```
    ... | yes refl = ⊥-elim $ y∉ 𝟘
    ... | no y≢ = ≢-sym y≢



-- TODO: generalize this to more complex types than Atom (c.f.
supp-swap-atom : ∀ {𝕒 𝕓} (t : Atom) → supp (swap 𝕒 𝕓 t) ⊆ 𝕒 ∷ 𝕓 ∷ t ∷ []
supp-swap-atom {a}{b} t
  with t ≟ a
... | yes refl = λ where 𝟘 → 𝟙
... | no _
  with t ≟ b
... | yes refl = λ where 𝟘 → 𝟘
... | no _ = λ where 𝟘 → 𝟚
```

Nominal/Swap.agda

```agda
open import Prelude.Init; open SetAsType
open import Prelude.DecEq

module Nominal.Swap (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap.Base        Atom public
open import Nominal.Swap.Derive      Atom public
open import Nominal.Swap.Equivariance Atom public
```

Nominal/Swap/Base.agda

```
{- MOTTO: permutations distribute over everything -}
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.General
open import Prelude.DecEq
open import Prelude.Decidable
open import Prelude.Setoid
open import Prelude.InferenceRules

module Nominal.Swap.Base (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

Atoms = List Atom

-- TODO: use sized types to enforce size-preserving swap
record Swap (A : Type ℓ) : Type ℓ where
  field swap : Atom → Atom → A → A
```

```
  -- TODO: ++ swap forms a group action by the group of atom pe
  -- i.e. • id x = x
  --      • p (p′ x) = (p ∘ p′) x

  infixr 10 ⦅_↔_⦆_
  ⦅_↔_⦆_ = swap
  -- NB: equivariant functions commute with this group action

  swaps : List (Atom × Atom) → A → A
  swaps []              = id
  swaps ((x , y) ∷ as) = swap x y ∘ swaps as

open Swap ⦃...⦄ public

instance
  Swap-Atom : Swap Atom
```

```
  Swap-Atom.swap x y z =
    if      z == x then y
    else if z == y then x
    else z

-- TODO: permutations as bijections on `Atom` (infinite variar

-- TODO: to connect everything with the group theory behind
-- π∘π′ = (π′^π)∘π, where _^_ is the group conjugation action
--      = (π∘π′∘π⁻¹)∘π
--      = (π•π′)∘π

record CongSetoid (A : Set ℓ) ⦃ _ : ISetoid A ⦄ ⦃ _ : SetoidLaws A ⦄ : S
  field ≈-cong : ∀ {B : Set ℓ′} ⦃ _ : ISetoid B ⦄ ⦃ _ : SetoidLaws B ⦄ →
                 ∀ (f : A → B) → Congruent _≈_ _≈_ f
open CongSetoid ⦃...⦄ public
```

```
instance
  Setoid-Atom : ISetoid Atom
  Setoid-Atom = λ where
    .relℓ → 0ℓ
    ._≈_ → _≡_

  SetoidLaws-Atom : SetoidLaws Atom
  SetoidLaws-Atom .isEquivalence = PropEq.isEquivalence

  CongSetoid-Atom : CongSetoid Atom
  CongSetoid-Atom .≈-cong _ refl = ≈-refl

swapˡ : ∀ 𝕒 𝕓 → ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕒 ≡ 𝕓
swapˡ 𝕒 𝕓 rewrite ≟-refl 𝕒 = refl

swapʳ : ∀ 𝕒 𝕓 → ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕓 ≡ 𝕒
```

```
swapʳ 𝕒 𝕓 with 𝕓 ≟ 𝕒
... | yes refl = refl
... | no 𝕓≢
  rewrite T⇒true $ fromWitnessFalse {Q = 𝕓 ≟ 𝕒} 𝕓≢
    | ≟-refl 𝕓
    = refl

swap-noop : ∀ 𝕒 𝕓 x → x ∉ 𝕒 ∷ 𝕓 ∷ [] → ⦅ 𝕒 ↔ 𝕓 ⦆ x ≡ x
swap-noop 𝕒 𝕓 x x∉ with x ≟ 𝕒
... | yes refl = ⊥-elim $ x∉ $ here refl
... | no _ with x ≟ 𝕓
... | yes refl = ⊥-elim $ x∉ $ there $′ here refl
... | no _ = refl

pattern 𝟘 = here refl
pattern 𝟙 = there 𝟘
```

```
pattern 𝟚 = there 𝟙
pattern 𝟛 = there 𝟚

module _ (A : Type ℓ) ⦃ _ : Swap A ⦄ ⦃ _ : LawfulSetoid A ⦄ where

  private variable
    x y : A
    𝕒 𝕓 𝕔 𝕕 : Atom

  record SwapLaws : Type (ℓ ⊔ₗ relℓ) where
    field
      cong-swap : x ≈ y → ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ ⦅ 𝕒 ↔ 𝕓 ⦆ y
      swap-id   : ⦅ 𝕒 ↔ 𝕒 ⦆ x ≈ x
      swap-rev  : ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ ⦅ 𝕓 ↔ 𝕒 ⦆ x
      swap-sym  : ⦅ 𝕒 ↔ 𝕓 ⦆ ⦅ 𝕓 ↔ 𝕒 ⦆ x ≈ x
      swap-swap : ⦅ 𝕒 ↔ 𝕓 ⦆ ⦅ 𝕔 ↔ 𝕕 ⦆ x
```

```
  ≈ ⟪ ⟪ 𝕒 ↔ 𝕓 ⟫ 𝕔 ↔ ⟪ 𝕒 ↔ 𝕓 ⟫ 𝕕 ⟫ ⟪ 𝕒 ↔ 𝕓 ⟫ x

-- ** derived properties
swap-comm :
  Disjoint (𝕒 ∷ 𝕓 ∷ []) (𝕔 ∷ 𝕕 ∷ [])
  ────────────────────────────────────────────
  ⟪ 𝕒 ↔ 𝕓 ⟫ ⟪ 𝕔 ↔ 𝕕 ⟫ x ≈ ⟪ 𝕔 ↔ 𝕕 ⟫ ⟪ 𝕒 ↔ 𝕓 ⟫ x
swap-comm {𝕒 = a}{b}{c}{d}{x} ab♯cd
  with eq ← swap-swap {𝕒 = a}{b}{c}{d}{x}
  rewrite swap-noop a b c $ ab♯cd ∘ (_, 𝟘)
        | swap-noop a b d $ ab♯cd ∘ (_, 𝟙)
        = eq

swap-sym′ : ⟪ 𝕒 ↔ 𝕓 ⟫ ⟪ 𝕒 ↔ 𝕓 ⟫ x ≈ x
swap-sym′ = ≈-trans (cong-swap swap-rev) swap-sym
```

```
    swap-id≈ : x ≈ y → ⟪ 𝕒 ↔ 𝕒 ⟫ x ≈ y
    swap-id≈ x≈y = ≈-trans (cong-swap x≈y) swap-id

    swap-rev≈ : x ≈ y → ⟪ 𝕒 ↔ 𝕓 ⟫ x ≈ ⟪ 𝕓 ↔ 𝕒 ⟫ y
    swap-rev≈ x≈y = ≈-trans swap-rev (cong-swap x≈y)

    swap-sym≈ : x ≈ y → ⟪ 𝕒 ↔ 𝕓 ⟫ ⟪ 𝕓 ↔ 𝕒 ⟫ x ≈ y
    swap-sym≈ x≈y = ≈-trans swap-sym x≈y

    swap-swap≈ : x ≈ y → ⟪ 𝕒 ↔ 𝕓 ⟫ ⟪ 𝕔 ↔ 𝕕 ⟫ x
                       ≈ ⟪ ⟪ 𝕒 ↔ 𝕓 ⟫ 𝕔 ↔ ⟪ 𝕒 ↔ 𝕓 ⟫ 𝕕 ⟫ ⟪ 𝕒 ↔ 𝕓 ⟫ y
    swap-swap≈ x≈y = ≈-trans swap-swap (cong-swap $ cong-swap x≈y)


  open SwapLaws ⦃...⦄ public

  private variable A : Type ℓ
```

```
instance
  SwapLaws-Atom : SwapLaws Atom
  SwapLaws-Atom .cong-swap = λ where refl → refl
  SwapLaws-Atom .swap-id {a}{x}
    with x ≟ a
  ... | yes refl = refl
  ... | no _     = refl
  SwapLaws-Atom .swap-rev {a}{b}{c} with c ≟ a | c ≟ b
  ... | yes refl | yes refl = refl
  ... | yes refl | no _  = refl
  ... | no _     | yes refl = refl
  ... | no _     | no _ = refl
  SwapLaws-Atom .swap-sym {a}{b}{x}
    with x ≟ b
```

```
... | yes refl rewrite ≟-refl a = refl
... | no x≢b
  with x ≟ a
... | yes refl
  rewrite ≟-refl a
        | dec-no (b ≟ x) (≢-sym x≢b) .proj₂
        | ≟-refl b
        = refl
... | no x≢a
  rewrite dec-no (x ≟ a) x≢a .proj₂
        | dec-no (x ≟ b) x≢b .proj₂
        = refl
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
{- ⦅ 𝕒 ↔ 𝕓 ⦆ ⦅ 𝕔 ↔ 𝕕 ⦆ x
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ x -}
```

```
  with a ≟ b | c ≟ d
... | yes refl | _
{- ⦅ 𝕒 ↔ a ⦆ ⦅ 𝕔 ↔ 𝕕 ⦆ x
  ≈ ⦅ ⦅ 𝕒 ↔ a ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ a ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ a ⦆ x -}
  rewrite swap-id {𝕒 = a} {x = ⦅ c ↔ d ⦆ x}
    | swap-id {𝕒 = a} {x = c}
    | swap-id {𝕒 = a} {x = d}
    | swap-id {𝕒 = a} {x = x}
    = refl
... | _ | yes refl
{- ⦅ 𝕒 ↔ 𝕓 ⦆ ⦅ 𝕔 ↔ c ⦆ x
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ c ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ x -}
  rewrite swap-id {𝕒 = c} {x = x}
    | swap-id {𝕒 = ⦅ a ↔ b ⦆ c} {x = ⦅ a ↔ b ⦆ x}
    = refl
```

```
... | no a≢b | no c≢d
{- ⦅ 𝕒 ↔ 𝕓 ⦆ ⦅ 𝕔 ↔ 𝕕 ⦆ x
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ c ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ x -}
  with x ≟ c
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
  | no a≢b | no c≢d | yes refl
{- ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ c ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ c -}
  rewrite swapˡ (⦅ a ↔ b ⦆ c) (⦅ a ↔ b ⦆ d) = refl
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
  | no a≢b | no c≢d | no x≢c
  with x ≟ d
{- ⦅ 𝕒 ↔ 𝕓 ⦆ ⦅ ✓𝕔 ↔ 𝕕 ⦆ x
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ x -}
... | yes refl
```

```
{- ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ d -}
  rewrite swapʳ (⦅ a ↔ b ⦆ c) (⦅ a ↔ b ⦆ d) = refl
... | no x≢d
{- ⦅ 𝕒 ↔ 𝕓 ⦆ x
  ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ 𝕒 ↔ 𝕓 ⦆ x -}
  with x ≟ a
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
  | no a≢b | no c≢d | no a≢c | no a≢d | yes refl {-x≡a-}
{- 𝕓 ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ 𝕓 -}
  rewrite dec-no (c ≟ a) (≢-sym a≢c) .proj₂
{- 𝕓 ≈ ⦅ ⦅ ✓𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ 𝕓 -}
  rewrite dec-no (d ≟ a) (≢-sym a≢d) .proj₂
{- 𝕓 ≈ ⦅ ⦅ ✓𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ ✓𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ 𝕓 -}
  with c ≟ b | d ≟ b
```

```
... | yes refl {-c≡b-} | yes refl {-d≡b-} {- 𝕓 ≈ ⦅ 𝕒 ↔ 𝕒 ⦆ 𝕓 -}
  rewrite swap-id {𝕒 = a} {x = b} = refl
... | yes refl {-c≡b-} | no d≢b {- 𝕓 ≈ ⦅ 𝕒 ↔ 𝕕 ⦆ 𝕓 -}
  rewrite swap-noop a d b (λ where 𝟘 → a≢b refl; 𝟙 → d≢b refl) = refl
... | no c≢b | yes refl {-d≡b-} {- 𝕓 ≈ ⦅ 𝕔 ↔ 𝕒 ⦆ 𝕓 -}
  rewrite swap-noop c a b (λ where 𝟘 → c≢b refl; 𝟙 → a≢b refl) = refl
... | no c≢b | no d≢b {- 𝕓 ≈ ⦅ 𝕔 ↔ 𝕕 ⦆ 𝕓 -}
  rewrite swap-noop c d b (λ where 𝟘 → c≢b refl; 𝟙 → d≢b refl) = refl
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
  | no a≢b | no c≢d | no x≢c | no x≢d | no x≢a
{- ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ ⦅ √𝕒 ↔ 𝕓 ⦆ x -
  with x ≟ b
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
  | no a≢b | no c≢d | no b≢c | no b≢d | no b≢a | yes refl {-x≡b-}
{- 𝕒 ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ 𝕒 -}
```

```
    with c ≟ a | d ≟ a
  ... | yes refl {-c≡a-} | yes refl {-d≡a-} = ⊥-elim $ c≢d refl
  ... | yes refl {-c≡a-} | no d≢a {- 𝕒 ≈ ⟪ 𝕓 ↔ ⟪ ✓𝕒 ↔ 𝕓 ⟫ 𝕕 ⟫ 𝕒 -}
    rewrite dec-no (d ≟ b) (≢-sym b≢d) .proj₂
          | swap-noop b d a (λ where 𝟘 → a≢b refl; 𝟙 → d≢a refl)
          = refl
  ... | no c≢a | yes refl {-d≡a-} {- 𝕒 ≈ ⟪ ⟪ ✓𝕒 ↔ 𝕓 ⟫ 𝕔 ↔ 𝕓 ⟫ 𝕒 -}
    rewrite dec-no (c ≟ b) (≢-sym b≢c) .proj₂
          | swap-noop c b a (λ where 𝟘 → c≢a refl; 𝟙 → a≢b refl)
          = refl
  ... | no c≢a | no d≢a {- 𝕒 ≈ ⟪ ⟪ ✓𝕒 ↔ 𝕓 ⟫ 𝕔 ↔ ⟪ ✓𝕒 ↔ 𝕓 ⟫ 𝕕 ⟫ 𝕒 -
    rewrite dec-no (c ≟ b) (≢-sym b≢c) .proj₂
          | dec-no (d ≟ b) (≢-sym b≢d) .proj₂
          | swap-noop c d a (λ where 𝟘 → c≢a refl; 𝟙 → d≢a refl)
          = refl
```

```
SwapLaws-Atom .swap-swap {𝕒 = a}{b}{c}{d}{x}
  | no a≢b | no c≢d | no x≢c | no x≢d | no x≢a | no x≢b
{- ⦅ 𝕒 ↔ 𝕓 ⦆ x ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ x -}
  rewrite swap-noop a b x (λ where 𝟘 → x≢a refl; 𝟙 → x≢b refl)
{- x ≈ ⦅ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕔 ↔ ⦅ 𝕒 ↔ 𝕓 ⦆ 𝕕 ⦆ x -}
  with c ≟ a | c ≟ b | d ≟ a | d ≟ b
... | yes refl | _ | yes refl | _ = ⊥-elim $ c≢d refl
... | yes refl | _ | no d≢a | yes refl
  {- x ≈ ⦅ 𝕓 ↔ 𝕒 ⦆ x -}
  rewrite swap-noop b a x (λ where 𝟘 → x≢b refl; 𝟙 → x≢a refl) = refl
... | yes refl | _ | no d≢a | no d≢b
  {- x ≈ ⦅ 𝕓 ↔ 𝕕 ⦆ x -}
  rewrite swap-noop b d x (λ where 𝟘 → x≢b refl; 𝟙 → x≢d refl) = refl
... | _ | yes refl | _ | yes refl = ⊥-elim $ c≢d refl
... | no c≢a | yes refl | yes refl | _
```

```
    {- x ≈ ⦅ 𝕒 ↔ 𝕓 ⦆ x -}
    rewrite swap-noop a b x (λ where 𝟘 → x≢a refl; 𝟙 → x≢b refl) = refl
  ... | no c≢a | yes refl | no d≢a | no d≢b
    {- x ≈ ⦅ 𝕒 ↔ d ⦆ x -}
    rewrite swap-noop a d x (λ where 𝟘 → x≢a refl; 𝟙 → x≢d refl) = refl
  ... | no c≢a | no c≢b | yes refl | _
    {- x ≈ ⦅ 𝕔 ↔ 𝕓 ⦆ x -}
    rewrite swap-noop c b x (λ where 𝟘 → x≢c refl; 𝟙 → x≢b refl) = refl
  ... | no c≢a | no c≢b | no d≢a | yes refl
    {- x ≈ ⦅ 𝕔 ↔ 𝕒 ⦆ x -}
    rewrite swap-noop c a x (λ where 𝟘 → x≢c refl; 𝟙 → x≢a refl) = refl
  ... | no c≢a | no c≢b | no d≢a | no d≢b
    {- x ≈ ⦅ 𝕔 ↔ 𝕕 ⦆ x -}
    rewrite swap-noop c d x (λ where 𝟘 → x≢c refl; 𝟙 → x≢d refl) = refl
```

```
-- ** Nameless instances.
swapId : Atom → Atom → A → A
swapId _ _ = id

mkNameless : (A : Type) → Swap A
mkNameless A = λ where .swap → swapId

instance
  ⊤∅ = mkNameless ⊤
  𝔹∅ = mkNameless Bool
  ℕ∅ = mkNameless ℕ
  ℤ∅ = mkNameless ℤ
  Char∅  = mkNameless Char
  String∅ = mkNameless String

swap-≢ : ∀ {z w x y} → z ≢ w → swap x y z ≢ swap x y w
```

```
swap-≢ {z}{w}{x}{y} z≢w
  with z ≟ x
swap-≢ {z}{w}{x}{y} z≢w | yes refl
  rewrite dec-no (w ≟ z) (≢-sym z≢w) .proj₂
  with w ≟ y
... | yes refl = ≢-sym z≢w
... | no w≢y = ≢-sym w≢y
swap-≢ {z}{w}{x}{y} z≢w | no z≢x
  with z ≟ y
... | yes refl
  = QED
  where
  QED : x ≢ swap x z w
  QED with w ≟ x
  ... | yes refl = ≢-sym z≢x
```

```
    ... | no w≢x
      rewrite dec-no (w ≟ z) (≢-sym z≢w) .proj₂
      = ≢-sym w≢x
  ... | no z≢y
   with w ≟ x
  ... | yes refl = z≢y
  ... | no _
   with w ≟ y
  ... | yes refl = z≢x
  ... | no _ = z≢w
```

Nominal/Swap/Derive.agda

```
{-
Here we derive the canonical swapping for every inductive data
i.e. the swapping the equivariantly distributes amongst constr
a.k.a. constructors are equivariant.
-}

{-# OPTIONS -v nominal:100 #-}
open import Prelude.Init
open import Prelude.DecEq
open import Prelude.Monad
open import Prelude.Semigroup
open import Prelude.Show
open import Prelude.ToN
open import Prelude.Lists
open import Prelude.Functor
```

```
open import Prelude.Bifunctor
open import Prelude.Applicative

open Meta
open import Prelude.Generics
open import Prelude.Tactics.Extra using (getPatTele)
open Debug ("nominal" , 100)

module Nominal.Swap.Derive (Atom : Set) ⦃ _ : DecEq Atom ⦄ where

open import Nominal.Swap.Base Atom

{-# TERMINATING #-}
derive↔ : Definition → TC Term
derive↔ d with d
... | record-type rn fs = do
```

```
    print $ "RECORD {name = " ◇ show rn ◇ "; fs = " ◇ show fs ◇ "}"
    return `λ⟦ "a" ∣ "b" ∣ "r" ⇒ pat-lam (map (mkClause ∘ unArg) fs) [] 
    where
      ♯a = ♯ 2; ♯b = ♯ 1; ♯r = ♯ 0

      mkClause : Name → Clause
      mkClause fn = clause [] [ vArg $ proj fn ] (quote swap •⟦ ♯a ∣ ♯b ∣ 
... | data-type ps cs = do
    print $ "DATATYPE {pars = " ◇ show ps ◇ "; cs = " ◇ show cs ◇ "}"
    cs′ ← mapM mkClause cs
    return `λ⟦ "a" ∣ "b" ⇒ pat-lam cs′ [] ⟧
    where
      mkClause : Name → TC Clause
      mkClause cn = do
        tel ← getPatTele cn
```

```
print $ "Making pattern clause for constructor: " ◇ show cn ◇
let N = length tel; ♯a = ♯ (N + 1); ♯b = ♯ N
print $ "  ♯a: " ◇ show ♯a
print $ "  ♯b: " ◇ show ♯b
let tel′ = map (map₂ $ fmap $ const unknown) tel
print $ "  tel′: " ◇ show tel′
let
  tys : Args Type
  tys = map proj₂ tel′

  itys = enumerate tys

  cArgsᵖ : Args Pattern
  cArgsᵖ = map (λ (i , at) → fmap (const $ var $ toℕ i) at) (L.rev

  cArgs : Args Term
```

```
      cArgs = map (λ (i , at) → fmap (const $ quote swap • ⟦ ♯a ∣ ♯b ∣ ♯

     return $ clause tel′ [ vArg $ con cn cArgsᵖ ] (con cn cArgs)
... | function cs = do
  print $ "FUNCTION {cs = " ◇ show cs ◇ "}"
  clause tel ps t ∷ [] ← return cs
    where _ → error "[not supported] multi-clause function"
  print $ "  tel: " ◇ show tel
  ctx ← getContext
  print $ "  ctx: " ◇ show ctx
  inContext (L.reverse tel) $ do
    t′@(def n as) ← normalise t
      where _ → error "[not supported] rhs does not refer to anoth
    print $ "  t′: " ◇ show t′
    d ← getDefinition n
```

```
    print $ "  d: " ◇ show d
    derive↔ d
  ... | data-cons _ = error "[not supported] data constructors"
  ... | axiom     = error "[not supported] axioms or mutual definit
  ... | prim-fun = error "[not supported] primitive functions"

  private variable A : Set ℓ

  addHypotheses : Type → Type
  addHypotheses = λ where
   (Π[ s : a ] ty) →
    let ty′ = addHypotheses ty
        ty″ = iΠ[ "_" : quote Swap ∙⟦ ♯ 0 ⟧ ] mapVars suc ty′
    in
    Π[ s : a ]
     (case unArg a of λ where
```

```
        (agda-sort (lit _)) → ty″
        (agda-sort (set _)) → ty″
        _ → ty′)
  ty → ty

externalizeSwap : ℕ → Type → Type
externalizeSwap n = λ where
  (def (quote Swap) as) →
   def (quote Swap) (vArg (♯ (suc n)) ∷ iArg (♯ n) ∷ as)
  (Π[ s : arg i a ] ty) →
   Π[ s : arg i (externalizeSwap n a) ] externalizeSwap (suc n) ty
  t → t

addHypotheses′ : (Type → Type) → Type → Type
addHypotheses′ Swap• = λ where
  (Π[ s : a ] ty) →
```

```
    let ty′ = addHypotheses′ Swap• ty
        ty″ = iΠ[ "_" : Swap• (♯ 0) ] mapVars suc ty′
    in
    Π[ s : a ]
      (case unArg a of λ where
        (agda-sort (lit _)) → ty″
        (agda-sort (set _)) → ty″
        _ → ty′)
  ty → ty

instance
  Derivable-Swap : DERIVABLE Swap 1
  Derivable-Swap .derive args = do
    print "*****************************************************
    (record-type ‘swap _) ← getDefinition (quote Swap)
```

```
  where _ → _IMPOSSIBLE_
((n , f) :: []) ← return args
  where _ → error "not supported: mutual types"
print $ "Deriving " ◇ parens (show f ◇ " : Swap " ◇ show n)
T ← getType n
ctx ← getContext
-- ctx@(_ :: _ :: []) ← getContext
--   where _ → error "Context ≠ {Atom : Set} ⦃ _ : DecEq At
print $ "  Context: " ◇ show ctx
print $ "  Type: " ◇ show T
d ← getDefinition n
print $ "  Definition: " ◇ show d
print $ "  argTys: " ◇ show (argTys T)

print $ "  Parameters: " ◇ show (parameters d)
```

```
let tel = tyTele T
print $ "  Indices: " ◇ show (unzip tel .proj₂)
let n′ = apply⋯ tel n
print $ "  n′: " ◇ show n′
let T′ = addHypotheses
       $ ∀indices⋯ tel
       $ quote Swap • ⟦ n′ ⟧
print $ "  T′: " ◇ show T′
-- let mn = length $ flip L.boolTakeWhile ctx λ where
--   (¬? ∘ (_≟ iArg (def (quote DecEq) {!!})) ∘ unArg ∘ pro
suc (suc mn) ← pure $ length ctx
  where _ → error "module parameters should always start with
let T″ = externalizeSwap mn T′
print $ "  T″: " ◇ show T″
T ← (declareDef (iArg f) T′ >> return T′)
```

```
      ◁ > (declareDef (iArg f) T″ >> return T″)

    let mctx = argTys T
        mtel = map ("_" ,_) mctx
        pc = map (λ where (i , a) → fmap (const (` (length mctx ∸ suc (t
    print $ "  mctx: " ◇ show mctx
    print $ "  mtel: " ◇ show mtel
    print $ "  pc: " ◇ show pc

    t ← derive↔ d
    print $ "t: " ◇ show t
    defineFun f [ clause mtel pc (`swap ◆⟦ t ⟧) ]

-- ** deriving examples
unquoteDecl ⊎↔ = DERIVE Swap [ quote _⊎_ , ⊎↔ ]
-- unquoteDecl Σ↔ = DERIVE Swap [ quote Σ , Σ↔ ]
```

```
unquoteDecl ×↔ = DERIVE Swap [ quote _×_ , ×↔ ]
{-# TERMINATING #-}
unquoteDecl List↔ = DERIVE Swap [ quote List , List↔ ]

private
  -- ** record types
  record R⁰ : Set where
  instance
    R∅ : Swap R⁰
    R∅ = mkNameless R⁰

  record R¹ : Set where
    field x : ℕ
  unquoteDecl r¹ = DERIVE Swap [ quote R¹ , r¹ ]

  record R¹′ : Set where
```

```
  field
    x : ℕ
    y : ℕ
    r : R¹
unquoteDecl r¹′ = DERIVE Swap [ quote R¹′ , r¹′ ]

record R² : Set where
  field
    x : ℕ × ℤ
    y : ⊤ ⊎ Bool
unquoteDecl r² = DERIVE Swap [ quote R² , r² ]

record P (A : Set) : Set where
  field x : A
unquoteDecl p = DERIVE Swap [ quote P , p ]
```

```
-- ** inductive datatypes

data X⁰ : Set where
instance
  X⁰∅ : Swap X⁰
  X⁰∅ = mkNameless X⁰

data X¹ : Set where
  x : X¹
unquoteDecl x¹ = DERIVE Swap [ quote X¹ , x¹ ]

data X¹' : Set where
  x : ℕ → X¹'
unquoteDecl x¹' = DERIVE Swap [ quote X¹' , x¹' ]

data X² : Set where
```

```
  x y : X²
unquoteDecl x² = DERIVE Swap [ quote X² , x² ]

data X²′ : Set where
  x y : ℕ → Bool → X²′
unquoteDecl x²′ = DERIVE Swap [ quote X²′ , x²′ ]

data PAIR (A B : Set) : Set where
  ⦅_,_⦆ : A → B → PAIR A B
unquoteDecl PAIR↔ = DERIVE Swap [ quote PAIR , PAIR↔ ]

data HPAIR {a b : Level} (A : Set a) (B : Set b) : Set (a ⊔ₗ b) where
  ⦅_,_⦆ : A → B → HPAIR A B
unquoteDecl HPAIR↔ = DERIVE Swap [ quote HPAIR , HPAIR↔ ]

infixr 4 _≑_
```

```
data LIST (A : Set) : Set where
  ∅ : LIST A
  _∷_ : A → LIST A → LIST A
-- instance
--   List↔ : ⦃ Swap A ⦄ → Swap (LIST A)
--   List↔ {A} ⦃ A↔ ⦄ .swap = λ 𝕒 → λ 𝕓 → λ where
--     ∅ → ∅
--     (a ∷ as) → swap 𝕒 𝕓 a ∷ swap 𝕒 𝕓 as
{-# TERMINATING #-}
unquoteDecl LIST↔ = DERIVE Swap [ quote LIST , LIST↔ ]

postulate 𝕒 𝕓 : Atom

_ : swap 𝕒 𝕓 (1 ∷ 2 ∷ 3 ∷ ∅) ≡ (1 ∷ 2 ∷ 3 ∷ ∅)
_ = refl
```

```
data XX : Set where
  c₂ : List X² → XX
  c₁ : X¹ → X² → XX
unquoteDecl xx = DERIVE Swap [ quote XX , xx ]
```

Nominal/Swap/Equivariance.agda

```
{-# OPTIONS --v equivariance:100 #-}
open import Prelude.Init
open L.Mem
open import Prelude.DecEq
open import Prelude.Functor
open import Prelude.Monad
open import Prelude.Semigroup
open import Prelude.Show
open import Prelude.Setoid
open import Prelude.Lists
open import Prelude.ToN
open import Prelude.Tactics.PostulateIt

open import Prelude.Generics
open Meta
```

```
open Debug ("equivariance" , 100)

module Nominal.Swap.Equivariance (Atom : Set) ⦃ _ : DecEq Atom ⦄ whe

open import Nominal.Swap.Base Atom

-- ** generically postulate the axiom scheme expressing distri
{- ∀ (𝕒 𝕓 : Atom).
     •[n = 0]
       ∀ (x : A).
         swap 𝕒 𝕓 x ≈ swap 𝕒 𝕓 x
     •[n = 1]
       ∀ (f : A → B) (x : A).
         swap 𝕒 𝕓 (f x) ≈ (swap 𝕒 𝕓 f) (swap 𝕒 𝕓 x)
     •[n = 2]
       ∀ (f : A → B → C) (x : A) (y : B).
```

```
           swap 𝕒 𝕓 (f x y) → (swap 𝕒 𝕓 f) (swap 𝕒 𝕓 x) (swap 𝕒
       ⋮
-}
deriveSwapDistributiveType : Bool → Term → TC Type
deriveSwapDistributiveType equiv? t = do
  ty ← inferType t
  print $ show t ◇ " : " ◇ show ty
  printCurrentContext
  ctx ← getContext
  let
    as₀ = argTys ty
    as  = map (fmap $ mapVars (_+ 2)) as₀
    n = length as

    mkSwaps : Args Term → Term
```

```
mkSwaps as = def (quote swap) $ map vArg ({-a-} # (suc n) :: {-b-} #

mkSwap : Op₁ Term
mkSwap t = mkSwaps [ vArg t ]

mkHead : Args Term → Term
mkHead as = case t of λ where
  (def f as₀) → def f (as₀ ++ as)
  (con c as₀) → con c (as₀ ++ as)
  (var i as₀) → var (i + 2 + n) (as₀ ++ as)
  _           → unknown

mkSwapHead : Args Term → Term
mkSwapHead as =
  let
    a = case t of λ where
```

```
      (def f as₀) → def f as₀
      (con c as₀) → con c as₀
      (var i as₀) → var (i + 2 + n) as₀
      _ → unknown
  in mkSwaps (vArg a ∷ as)

mkTerm : Op₁ Term → Args Term
mkTerm mk = flip map (enumerate as) λ where
  (i , arg v _) → arg v $ mk (♯ (n ∸ suc (toℕ i)))

lhs = mkSwap $ mkHead (mkTerm id)
rhs = (if equiv? then mkHead else mkSwapHead) (mkTerm mkSwap)

equivTy = vΠ[ "𝕒" ∶ ♯ (length ctx ∸ 1) ]
          vΠ[ "𝕓" ∶ ♯ (length ctx) ]
          ∀args as (quote _≈_ ∙⟦ lhs ∣ rhs ⟧)
```

```
    print $ "Equivariant " ◇ show t ◇ "  :=  " ◇ show equivTy
    print "----------------------------------------------"
    return equivTy
    where
      ∀args : Args Type → (Type → Type)
      ∀args [] = id
      ∀args (a ∷ as) t = hΠ[ "_" : unArg a ] ∀args as t

  deriveSwap↔ = deriveSwapDistributiveType false

  macro
    Swap↔ : Term → Hole → TC ⊤
    Swap↔ t hole = deriveSwap↔ t >>= unify hole

    swap↔ : Term → Hole → TC ⊤
    swap↔ t hole = do
```

```
    n ← genPostulate =<< deriveSwap↔ t
    unify hole (n •)

postulateSwap↔ : Name → Term → TC ⊤
postulateSwap↔ f t = declarePostulate (vArg f) =<< deriveSwap↔ t

-- ** derive the statement of equivariance for given term of a
-- be it a definition, constructor, or local variable
{- ∀ (𝕒 𝕓 : Atom).
     •[n = 0]
       ∀ (x : A).
         swap 𝕒 𝕓 x ≈ x
     •[n = 1]
       ∀ (f : A → B) (x : A).
         swap 𝕒 𝕓 (f x) ≈ f (swap 𝕒 𝕓 x)
     •[n = 2]
```

```
      ∀ (f : A → B → C) (x : A) (y : B).
        swap 𝕒 𝕓 (f x y) → f (swap 𝕒 𝕓 x) (swap 𝕒 𝕓 y)
    ⋮
-}
macro
  Equivariant : Term → Hole → TC ⊤
  Equivariant t hole = deriveSwapDistributiveType true t >>= unify

private
  data X : Set where
    mkX : ℕ → ℕ → X

  variable
    𝕒 𝕓 𝕔 𝕕 : Atom
  postulate
    n m : ℕ
```

```
  f : ℕ → ℕ
  g : ℕ → ℕ → ℕ

  instance
    _ : ISetoid ℕ
    _ : ISetoid X
    _ : Swap X
    _ : Swap (ℕ → ℕ)
    _ : Swap (ℕ → ℕ → ℕ)
    _ : Swap (ℕ → ℕ → X)

module _ 𝕒 𝕓 where postulate
  distr-f : ∀ {n} →
    swap 𝕒 𝕓 (f n) ≈ (swap 𝕒 𝕓 f) (swap 𝕒 𝕓 n)
  equiv-f : ∀ {n} →
    swap 𝕒 𝕓 (f n) ≈ f (swap 𝕒 𝕓 n)
```

```
    distr-g : ∀ {n m} →
      swap 𝕒 𝕓 (g n m) ≈ (swap 𝕒 𝕓 g) (swap 𝕒 𝕓 n) (swap 𝕒 𝕓 m)
    equiv-g : ∀ {n m} →
      swap 𝕒 𝕓 (g n m) ≈ g (swap 𝕒 𝕓 n) (swap 𝕒 𝕓 m)
    distr-mkX : ∀ {n m} →
      swap 𝕒 𝕓 (mkX n m) ≈ (swap 𝕒 𝕓 mkX) (swap 𝕒 𝕓 n) (swap 𝕒 𝕓 m)
    equiv-mkX : ∀ {n m} →
      swap 𝕒 𝕓 (mkX n m) ≈ mkX (swap 𝕒 𝕓 n) (swap 𝕒 𝕓 m)
  module _ {f : ℕ → ℕ} 𝕒 𝕓 where postulate
    distr-∀f : ∀ {n} → swap 𝕒 𝕓 (f n) ≈ (swap 𝕒 𝕓 f) (swap 𝕒 𝕓 n)
    equiv-∀f : ∀ {n} → swap 𝕒 𝕓 (f n) ≈ f (swap 𝕒 𝕓 n)

  _ = Swap↔ f ∋ distr-f
  _ = Swap↔ f ∋ swap↔ f
  _ = Equivariant f ∋ equiv-f
```

```
_ = Swap↔ g ∋ distr-g
_ = Swap↔ g ∋ swap↔ g
_ = Equivariant g ∋ equiv-g
_ = Swap↔ mkX ∋ distr-mkX
_ = Swap↔ mkX ∋ swap↔ mkX
_ = Equivariant mkX ∋ equiv-mkX
module _ {f : ℕ → ℕ} where
  _ = Swap↔ f ∋ swap↔ f
  _ = Swap↔ f ∋ distr-∀f
  _ = Equivariant f ∋ equiv-∀f

unquoteDecl distr-f′ = postulateSwap↔ distr-f′ (quoteTerm f)
unquoteDecl distr-g′ = postulateSwap↔ distr-g′ (quoteTerm g)
unquoteDecl distr-mkX′ = postulateSwap↔ distr-mkX′ (quoteTerm mk
module _ {f : ℕ → ℕ} where
```

```
unquoteDecl distr-∀f’ = postulateSwap↔ distr-∀f’ (quoteTerm f)
```

Nominal/Swap/Example.agda

```
{-# OPTIONS -v nominal:100 #-}
module Nominal.Swap.Example where

open import Prelude.Init; open SetAsType
open import Prelude.DecEq

-- ** instantiate atoms to be the natural numbers
data Atom : Type where
  `_ : ℕ → Atom
unquoteDecl DecEq-Atom = DERIVE DecEq [ quote Atom , DecEq-Atom ]
open import Nominal.Swap Atom
𝕒 = ` 0; 𝕓 = ` 1

data λTerm : Type where
  _-APP-_ : λTerm → λTerm → λTerm
  VAR : Atom → λTerm
```

```
-- {-# TERMINATING #-}
-- unquoteDecl λTerm↔ = DERIVE Swap [ quote λTerm , λTerm↔ ]
instance
  λTerm↔ : Swap λTerm
  λTerm↔ .swap = λ 𝕒 𝕓 → λ where
    (l -APP- r) → swap 𝕒 𝕓 l -APP- swap 𝕒 𝕓 r
    (VAR x) → VAR (swap 𝕒 𝕓 x)

-- ** example swapping in a λ-term
_ = swap 𝕒 𝕓 (VAR 𝕒 -APP- VAR 𝕓) ≡ VAR 𝕓 -APP- VAR 𝕒
  ∋ refl

-- ** derive and check ad-hoc example datatypes
record TESTR : Type where
  field atom : Atom
open TESTR
```

```
-- [TODO] derive outside module
-- unquoteDecl TESTR↔ = DERIVE Swap [ quote TESTR , TESTR↔ ]
instance
  TESTR↔ : Swap TESTR
  TESTR↔ .swap 𝕒 𝕓 (record {atom = x}) = record {atom = swap 𝕒 𝕓 x}

_ = swap 𝕒 𝕓 (record {atom = 𝕒}) ≡ record {atom = 𝕓} ∋ refl

data TEST : Type where
  ATOM : Atom → TEST
-- unquoteDecl TEST↔ = DERIVE Swap [ quote TEST , TEST↔ ]
instance
  TEST↔ : Swap TEST
  TEST↔ .swap 𝕒 𝕓 (ATOM x) = ATOM (swap 𝕒 𝕓 x)
```

```
_ = swap 𝕒 𝕓 (ATOM 𝕒) ≡ ATOM 𝕓
  ∋ refl
```

ULC.agda

```
open import Prelude.Init; open SetAsType
open import Prelude.DecEq

module ULC (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import ULC.Base         Atom public
open import ULC.Measure      Atom public
open import ULC.Alpha        Atom public
open import ULC.Substitution Atom public
open import ULC.Reduction    Atom public
```

ULC/Alpha.agda

```
open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.General
open import Prelude.Closures
open import Prelude.InferenceRules
open import Prelude.Decidable
open import Prelude.Setoid
open import Prelude.Bifunctor
open import Prelude.Measurable
open import Prelude.Ord
open import Prelude.InfEnumerable
open import Prelude.Lists.Dec

-- ** α-equivalence.
```

```
module ULC.Alpha (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enumerable∞ A

open import ULC.Base Atom ⦃ it ⦄
open import ULC.Measure Atom ⦃ it ⦄
open import Nominal  Atom

private variable A : Type ℓ; f g h : Abs A

-- TODO: factor out abstractions, deal with them generically
data _≡α_ : Term → Term → Type₀ where

  ν≈ :
    x ≈ y
    ───────────
    ` x ≡α ` y

  ξ≡ :
```

```
    • L ≡α L′
    • M ≡α M′
      ────────────────────────
      (L · M) ≡α (L′ · M′)

  ζ≡_ : ∀ {f g : Abs Term} →
    -- f ≗α g
    И (λ 𝕩 → conc f 𝕩 ≡α conc g 𝕩)
    ─────────────────────────────────
    (ƛ f) ≡α (ƛ g)

_≢α_ = ¬_ ∘₂ _≡α_

pattern ν≡ = ν≈ refl

instance
```

```
  Setoid-Term : ISetoid Term
  Setoid-Term = λ where
    .relℓ → 0ℓ
    ._≈_   → _≡α_

_≗α_ : Rel₀ (Abs Term)
f ≗α g = И (λ 𝕩 → conc f 𝕩 ≡α conc g 𝕩)

data _≡α⊎_ : Rel₀ (Term ⊎ Abs Term) where
  ≡_ :
    t ≡α t′
    ────────────────────
    inj₁ t ≡α⊎ inj₁ t′

  ≗_ : ∀ {f g} →
    f ≗α g
```

```
   ─────────────────
   inj₂ f ≡α⊎ inj₂ g

≡˘_ : inj₁ t ≡α⊎ inj₁ t′ → t ≡α t′
≡˘ (≡ p) = p

≗˘_ : ∀ {f g} → inj₂ f ≡α⊎ inj₂ g → f ≗α g
≗˘ (≗ p) = p

≡α⊎-refl : ∀ x → x ≡α⊎ x
≡α⊎-refl = <-rec _ go
  where
    go : ∀ x → (∀ y → y < x → y ≡α⊎ y) → x ≡α⊎ x
    go x rec with x
    ... | inj₁ (` _) = ≡ ν≡
    ... | inj₁ (l · m) = ≡ ξ≡ (≡˘ rec _ (l ·<ˡ m)) (≡˘ rec _ (l ·<ʳ m))
```

```
    ... | inj₁ (ƛ f) = ≡ζ≡ ≐˘ go (inj₂ f) rec
    ... | inj₂ f     = ≐ ([] , (λ y _ → ≡˘ rec _ (conc< f y)))

≡α-refl : ∀ t → t ≡α t
≡α-refl t = ≡˘ ≡α⊎-refl (inj₁ t)

≐α-refl : ∀ f → f ≐α f
≐α-refl f = ≐˘ ≡α⊎-refl (inj₂ f)

≡α⊎-sym : ∀ x y → x ≡α⊎ y → y ≡α⊎ x
≡α⊎-sym = <-rec _ go
  where
    go : ∀ x → (∀ y → y < x → (∀ z → y ≡α⊎ z → z ≡α⊎ y))
              → (∀ y → x ≡α⊎ y → y ≡α⊎ x)
    go x rec _ eq with x | eq
    ... | inj₁ (` _) | ≡ν≡
```

```
      = ≡ν≡
  ... | inj₁ (l · m) | ≡ξ≡ p q
      = ≡ξ≡ (≡˘ rec _ (l ·<ˡ m) _ (≡ p)) (≡˘ rec _ (l ·<ʳ m) _ (≡ q))
  ... | inj₁ (ƛ f) | ≡ζ≡ p
      = ≡ζ≡ ≗˘ go (inj₂ f) rec _ (≗ p)
  ... | inj₂ f | ≗ (xs , p)
      = ≗ (xs , λ y y∉ → ≡˘ rec _ (conc< f y) _ (≡ p y y∉))

≗α-sym : f ≗α g → g ≗α f
≗α-sym = ≗˘_ ∘ ≡α⊎-sym _ _ ∘ ≗_

≡α-sym : t ≡α t′ → t′ ≡α t
≡α-sym = ≡˘_ ∘ ≡α⊎-sym _ _ ∘ ≡_

mutual
  ≗α-trans : f ≗α g → g ≗α h → f ≗α h
```

```
≗α-trans (xs , f≈g) (ys , g≈h) =
  (xs ++ ys) , λ y y∉ → ≡α-trans (f≈g y (y∉ ∘ L.Mem.∈-++⁺ˡ)) (g≈h y (y

≡α-trans : t ≡α t′ → t′ ≡α t″ → t ≡α t″
≡α-trans ν≡ q = q
≡α-trans (ξ≡ p₁ p₂) (ξ≡ q₁ q₂) = ξ≡ (≡α-trans p₁ q₁) (≡α-trans p₂ q₂
≡α-trans (ζ≡ f≈g) (ζ≡ g≈h) = ζ≡ ≗α-trans f≈g g≈h

instance
  SetoidLaws-Term : SetoidLaws Term
  SetoidLaws-Term .isEquivalence = record {refl = ≡α-refl _; sym = ≡

  {-# TERMINATING #-}
  SwapLaws-Term : SwapLaws Term
  SwapLaws-Term .cong-swap = λ where
    ν≡       → ν≡
```

```
    (ξ≡ p q) → ξ≡ (cong-swap p) (cong-swap q)
    (ζ≡ f≈g) → ζ≡ (cong-swap f≈g)
SwapLaws-Term .swap-id {a}{t} with t
... | ` x   = ν≈ swap-id
... | l · r = ξ≡ swap-id swap-id
... | ƛ f   = ζ≡ swap-id
SwapLaws-Term .swap-rev {a}{b}{t} with t
... | ` x   = ν≈ swap-rev
... | l · r = ξ≡ swap-rev swap-rev
... | ƛ f   = ζ≡ swap-rev
SwapLaws-Term .swap-sym {a}{b}{t} with t
... | ` x   = ν≈ swap-sym
... | l · r = ξ≡ swap-sym swap-sym
... | ƛ f   = ζ≡ swap-sym
SwapLaws-Term .swap-swap {a}{b}{c}{d}{t} with t
```

```
  ... | ` x   = ν≈ swap-swap
  ... | l · r = ξ≡ swap-swap swap-swap
  ... | ƛ f   = ζ≡ swap-swap

open ≈-Reasoning

cong-ƛ : t ≡α t′ → (ƛ x ⇒ t) ≡α (ƛ x ⇒ t′)
cong-ƛ t≡ = ζ≡ ([] , λ _ _ → cong-swap t≡)

instance
  ∃FinSupp-Term : ∃FinitelySupported Term
  ∃FinSupp-Term .∀∃fin = λ where
    (` x) → [ x ] , λ a b a∉ b∉ →
      ≈-reflexive $ cong `_ $
        swap-noop b a x λ where 𝟘 → b∉ 𝟘; 𝟙 → a∉ 𝟘
    (l · m) →
```

```
      let supˡ , pˡ = ∀∃fin l
          supᵐ , pᵐ = ∀∃fin m
      in (supˡ ++ supᵐ) , λ a b a∉ b∉ →
      ξ≡ (pˡ a b (a∉ ∘ ∈-++⁺ˡ) (b∉ ∘ ∈-++⁺ˡ))
         (pᵐ a b (a∉ ∘ ∈-++⁺ʳ _) (b∉ ∘ ∈-++⁺ʳ _))
  (ƛ x ⇒ t) → fin-ƛ t (∀∃fin t) x
   where
      fin-ƛ : ∀ (t : Term) → ∃FinSupp t → (∀ x → ∃FinSupp (ƛ x ⇒ t))
      fin-ƛ t (sup , p) x = x ∷ sup , λ a b a∉ b∉ →
        begin
          ⦅ b ↔ a ⦆ (ƛ x ⇒ t)
        ≡⟨⟩
          (ƛ ⦅ b ↔ a ⦆ x ⇒ ⦅ b ↔ a ⦆ t)
        ≡⟨ cong (λ ◆ → ƛ ◆ ⇒ ⦅ b ↔ a ⦆ t)
             $ swap-noop b a x (λ where 𝟘 → b∉ 𝟘; 𝟙 → a∉ 𝟘) ⟩
```

```
    (ƛ x ⇒ ⦅ b ↔ a ⦆ t)
  ≈⟨ cong-ƛ $ p a b (a∉ ∘ there) (b∉ ∘ there) ⟩
    (ƛ x ⇒ t)
  ∎

FinSupp-Term : FinitelySupported Term
FinSupp-Term .∀fin (` x) = xs , eq , ¬eq
  where
    xs = [ x ]

    eq : ∀ a b → a ∉ xs → b ∉ xs → swap b a (` x) ≈ ` x
    eq a b a∉ b∉ =
      ≈-reflexive $ cong `_ $
        swap-noop b a x λ where 𝟘 → b∉ 𝟘; 𝟙 → a∉ 𝟘

    ¬eq : ∀ a b → a ∈ xs → b ∉ xs → swap b a (` x) ≉ ` x
```

```
      ¬eq a b 𝟘 b∉ rewrite swapʳ b a = λ where ν≡ → b∉ 𝟘

  FinSupp-Term .∀fin (l · m)
    with supˡ , pˡ , ¬pˡ ← ∀fin l
    with supᵐ , pᵐ , ¬pᵐ ← ∀fin m
    = xs , eq , ¬eq -- same as Nominal.Product
    where
      xs = nub (supˡ ++ supᵐ)

      eq : ∀ a b → a ∉ xs → b ∉ xs → swap b a (l · m) ≈ l · m
      eq a b a∉ b∉ =
        ξ≡ (pˡ a b (a∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ˡ) (b∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ˡ))
           (pᵐ a b (a∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ʳ supˡ) (b∉ ∘ ∈-nub⁺ ∘ ∈-++⁺ʳ su

      -- TODO: should not hold, argument might remain unused
      -- *WRONG* the problem only arises when considering _nor
```

```
      postulate ¬eq : ∀ a b → a ∈ xs → b ∉ xs → swap b a (l · m) ≉ l · m
  FinSupp-Term .∀fin t̂@(ƛ x ⇒ t)
    with xs , p , ¬p ← ∀fin t
    = xs′ , eq , ¬eq -- same as Nominal.Abs
    where
      xs′ = filter (¬? ∘ (_≟ x)) xs
      -- TODO: both should be provable
      postulate
        eq : ∀ y z → y ∉ xs′ → z ∉ xs′ → swap z y t̂ ≈ t̂
        ¬eq : ∀ y z → y ∈ xs′ → z ∉ xs′ → swap z y t̂ ≉ t̂

∃supp-var : ∃supp (` x) ≡ [ x ]
∃supp-var = refl

supp-var : supp (` x) ≡ [ x ]
supp-var = refl
```

```
∃supp-ξ : ∃supp (L · M) ≡ ∃supp L ++ ∃supp M
∃supp-ξ = refl

supp-ξ : supp (L · M) ≡ nub (supp L ++ supp M)
supp-ξ = refl

∃supp-ƛ : ∃supp (ƛ x ⇒ N) ≡ x ∷ ∃supp N
∃supp-ƛ = refl

supp-ƛ : supp (ƛ x ⇒ N) ≡ filter (¬? ∘ (_≟ x)) (supp N)
supp-ƛ = refl

∃supp-id : ∃supp (ƛ x ⇒ ` x) ≡ x ∷ x ∷ []
∃supp-id = refl

supp-id : supp (ƛ x ⇒ ` x) ≡ []
```

```
supp-id {x = x} rewrite ≟-refl x = refl
```

```
-- supp-abs⊆ : ∀ (t̂ : Abs Term) {a b} (a∉ : a ∉ supp t̂) (b∉ :
--   (∀fin t̂ .proj₂ a b) a∉ b∉ .proj₁ ⊆ supp t̂
```

ULC/Base.agda

```
open import Prelude.Init; open SetAsType
open import Prelude.DecEq
open import Prelude.General
open import Prelude.Closures
open import Prelude.InferenceRules
open import Prelude.Decidable
open import Prelude.Membership
open import Prelude.Bifunctor

module ULC.Base (Atom : Type) ⦃ _ : DecEq Atom ⦄ where

open import Nominal Atom

-- ** ULC terms.
data Term : Type where
  `_ : Atom → Term
```

```
  _·_ : Op₂ Term
  ƛ_ : Abs Term → Term
{-# TERMINATING #-}
unquoteDecl Swap-Term = DERIVE Swap [ quote Term , Swap-Term ]

infix 30 `_
infixl 20 _·_
infixr 10 ƛ_
infixr 5 ƛ_⇒_
pattern ƛ_⇒_ x y = ƛ abs x y

variable
  x y z w x′ y′ z′ w′ 𝕩 𝕪 𝕫 𝕨 : Atom
  t t′ t″ L L′ M M′ N N′ M₁ M₂ : Term
  t̂ t̂′ t̂″ : Abs Term
```

```
-- ** utilities

data TermShape : Type where
  `■ : TermShape
  ƛ_ : TermShape → TermShape
  _·_ : TermShape → TermShape → TermShape

shape : Term → TermShape
shape = λ where
  (ƛ t) → ƛ shape (t .term)
  (L · M) → shape L · shape M
  (` _) → `■

private
  postulate 𝕒 𝕓 : Atom
```

```
  _ : shape (ƛ 𝕒 ⇒ ƛ 𝕓 ⇒ ` 𝕒 · (` 𝕓 · ` 𝕒))
    ≡ (ƛ ƛ `█ · (`█ · `█))
  _ = refl

isVarShape isLamShape isAppShape : Pred₀ TermShape
isVarShape = λ where
  `█ → ⊤
  _ → ⊥
isLamShape = λ where
  (ƛ _) → ⊤
  _ → ⊥
isAppShape = λ where
  (_ · _) → ⊤
  _ → ⊥

isƛ isApp isVar : Pred₀ Term
```

```
isƛ  = isLamShape ∘ shape
isApp = isAppShape ∘ shape
isVar = isVarShape ∘ shape

unƛ : (t : Term) {_ : isƛ t} → Abs Term
unƛ (ƛ t̂) {tt} = t̂

unApp : (t : Term) {_ : isApp t} → Term × Term
unApp (L · M) {tt} = L , M

unVar : (t : Term) {_ : isVar t} → Atom
unVar (` v) {tt} = v

_≡⦅shape⦆_ = _≡_ on shape

app-shape≡ : (L · M) ≡⦅shape⦆ (L′ · M′) → (L ≡⦅shape⦆ L′) × (M ≡⦅shape
app-shape≡ {L}{M}{L′}{M′} eq
```

```
  with shape L | shape L′ | shape M | shape M′ | eq
... | _ | _ | _ | _ | refl = refl , refl

app-shape≡˘ : (L ≡⟪shape⟫ L′) → (M ≡⟪shape⟫ M′) → (L · M) ≡⟪shape⟫ (L
app-shape≡˘ {L}{M}{L′}{M′} L≡ M≡
  with shape L | shape L′ | L≡
... | _ | _ | refl
  with shape M | shape M′ | M≡
... | _ | _ | refl = refl

lam-shape≡ : (ƛ t̂) ≡⟪shape⟫ (ƛ t̂′) → t̂ .term ≡⟪shape⟫ t̂′ .term
lam-shape≡ {t̂}{t̂′} eq
  with shape (t̂ .term) | shape (t̂′ .term) | eq
... | _ | _ | refl = refl

lam-shape≡˘ : t̂ .term ≡⟪shape⟫ t̂′ .term → (ƛ t̂) ≡⟪shape⟫ (ƛ t̂′)
```

```
lam-shape≡˘ {t̂}{t̂′} eq
  with shape (t̂ .term) | shape (t̂′ .term) | eq
... | _ | _ | refl = refl

swap-shape≡ : ∀ x y t → t ≡⦅shape⦆ swap x y t
swap-shape≡ x y = λ where
  (` _) → refl
  (L · M) → app-shape≡˘ (swap-shape≡ x y L) (swap-shape≡ x y M)
  (ƛ t̂) → lam-shape≡˘ {t̂}{swap x y t̂} $ swap-shape≡ x y (t̂ .term)

swap-shape : ∀ t t′ → t ≡⦅shape⦆ t′ → swap x y t ≡⦅shape⦆ swap x′ y′ t′
swap-shape t t′ = flip trans (swap-shape≡ _ _ t′)
                ∘ trans (sym $ swap-shape≡ _ _ t)

conc-shape : ∀ t̂ t̂′ → t̂ .term ≡⦅shape⦆ t̂′ .term → conc t̂ x ≡⦅shape⦆ co
conc-shape t̂ t̂′ eq = swap-shape (t̂ .term) (t̂′ .term) eq
```

```
conc-shape≡ : ∀ t̂ → t̂ .term ≡⟪shape⟫ conc t̂ x
conc-shape≡ t̂ = swap-shape≡ _ _ (t̂ .term)
```

ULC/Examples.agda

```
module ULC.Examples where

open import Prelude.Init; open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.Nary
open import Prelude.Decidable
open import Prelude.Setoid
open import Prelude.General
open import Prelude.InfEnumerable
open import Prelude.Semigroup

-- ** instantiate atoms to be the natural numbers
record Atom : Type where
  constructor $_
  field un$ : ℕ
```

```
open Atom public
unquoteDecl DecEq-Atom = DERIVE DecEq [ quote Atom , DecEq-Atom ]
instance
  Enum-Atom : Enumerable∞ Atom
  Enum-Atom .enum = Fun.mk↔ {f = un$} {$_} ((λ _ → refl) , (λ _ → refl)
open import Nominal Atom
open import ULC Atom
  as ULC
  hiding (z)

s = $ 0; z = $ 1; m = $ 2; n = $ 3
a = $ 10; b = $ 11; c = $ 12; d = $ 13; e = $ 14

-- ** α-equivalence

_ = (` a) ≡α (` a) ∋ ν≡
```

```
_ : (ƛ a ⇒ ` a) ≡α (ƛ b ⇒ ` b)
_ = ζ≡ (-, qed)
  where qed : ∀ y → y L.Mem.∉ [] → swap y a (` a) ≡α swap y b (` b)
        qed y _ rewrite swapʳ y a | swapʳ y b = ν≡

h : (ƛ a ⇒ ` c · ` a) ≡α (ƛ b ⇒ ` c · ` b)
h = ζ≡ (-, qed)
  where qed : ∀ y → y L.Mem.∉ [ c ] → swap y a (` c · ` a) ≡α swap y b (` c
        qed y y∉ rewrite swapʳ y a | swapʳ y b
                  | swap-noop y a c (λ where (here refl) → y∉ auto; (ther
                  | swap-noop y b c (λ where (here refl) → y∉ auto; (ther
                  = ξ≡ ν≡ ν≡

_ : (ƛ a ⇒ ` c) ≡α (ƛ b ⇒ ` c)
_ = ζ≡ (-, qed)
```

```
  where
    qed : ∀ y → y L.Mem.∉ [ c ] → swap y a (` c) ≡α swap y b (` c)
    qed y y∉
      rewrite swap-noop y a c (λ where (here refl) → y∉ auto; (there (h
        | swap-noop y b c (λ where (here refl) → y∉ auto; (there (here (
      = ν≡

{-
  ¬equiv : ¬ (∀ t t′ 𝕒 𝕓 → t ≡α t′ → swap 𝕒 𝕓 t ≡α swap 𝕒 𝕓 t′
  ¬equiv p = {!absurd!}
    where
        _t  = ƛ a ⇒ ` c
        _t′ = ƛ b ⇒ ` c

        eq : _t ≡α _t′
```

```
      eq = ζ≡ (-, qed)
        where
        qed : ∀ y → y L.Mem.∉ [ c ] → swap y a (` c) ≡α swap
        qed y y∉
          rewrite swap-noop y a c (λ where (here refl) → y∉
                | swap-noop y b c (λ where (here refl) → y
                = ν≡

      absurd : swap a c _t ≡α swap a c _t′
      absurd = p _t _t′ a c eq

_ : (ƛ c ⇒ ƛ a ⇒ ` c · ` a) ≡α (ƛ c ⇒ ƛ b ⇒ ` c · ` b)
_ = ζ≡ (-, qed)
  where
    qed : ∀ y → y L.Mem.∉ [ a ]
```

```
        → swap y c (ƛ a ⇒ ` c · ` a) ≡α swap y c (ƛ b ⇒ ` c
      qed y _ rewrite swapʳ y a | swapʳ y b = {!h!}

  _ : (ƛ c ⇒ ƛ a ⇒ ` c · ` a) ≡α (ƛ d ⇒ ƛ b ⇒ ` d · ` b)
  _ : (ƛ c ⇒ ƛ a ⇒ ` c · ` a) ≢α (ƛ d ⇒ ƛ b ⇒ ` c · ` b)
-}

-- ** finite support

ex : Term
ex = ƛ a ⇒ ` a · ` a

suppEx = Atoms ∋ []
suppEx⁺ = Atoms ∋ [ a ]

finEx : FinSupp ex
```

```
finEx = -, go
  where
    go : ∀ 𝕒 𝕓 → 𝕒 ∉ suppEx⁺ → 𝕓 ∉ suppEx⁺ → swap 𝕓 𝕒 ex ≡α ex
    go 𝕒 𝕓 a∉ b∉
      rewrite swap-noop 𝕓 𝕒 a (λ where 𝟘 → b∉ auto; 𝟙 → a∉ auto)
        = ≡α-refl _

_ = finEx .proj₁ ≡ suppEx⁺
  ∋ refl

_ = supp ex
  ≡ (a ∷ a ∷ a ∷ [])
  ∋ refl

-- ** substitution
```

```
_ = (` a) [ a / ` b ] ≡ ` b
  ∋ refl

_ = (` a) [ a / ` b · ` b ] ≡ ` b · ` b
  ∋ refl

_ = (` a · ` a) [ a / ` b ] ≡ ` b · ` b
  ∋ refl

_ = (` a · ` a) [ a / ` b · ` b ] ≡ (` b · ` b) · (` b · ` b)
  ∋ refl

a' = $ 0 -- fresh in [a, b]

_ = (ƛ a ⇒ ` a) [ a / ` b ] ≡ (ƛ a' ⇒ ` a')
  ∋ refl
```

```
_ = (` a · (ƛ a ⇒ ` a)) [ a / ` b ] ≡ ` b · (ƛ a' ⇒ ` a')
  ∋ refl

b' = $ 0 -- fresh in [b, c]

_ = (ƛ b ⇒ ` b) [ b / ` c ] ≡ (ƛ b' ⇒ ` b')
  ∋ refl

c' = $ 0 -- fresh in [a, b, c]

_ = (` a · (ƛ c ⇒ ` c · ` a)) [ a / ` b ] ≡ (` b · (ƛ c' ⇒ ` c' · ` b))
  ∋ refl

_ = (` a · (ƛ c ⇒ ` c · ` a)) [ a / ` c' ] ≢ (` c' · (ƛ c' ⇒ ` c' · ` c'))
  ∋ λ ()

c'' = $ 1 -- fresh in [a, c, c']
```

```
_ = (` a · (ƛ c ⇒ ` c · ` a)) [ a / ` c' ] ≡ (` c' · (ƛ c'' ⇒ ` c'' · ` c'))
  ∋ refl

-- ** barendregt
a'' = $ 1 -- fresh in [a]

_ = barendregt (ƛ a ⇒ ƛ a ⇒ ` a · ` a) ≡ (ƛ a' ⇒ ƛ a'' ⇒ ` a'' · ` a'')
  ∋ refl

_ = barendregt ((ƛ a ⇒ ` a) · (ƛ a ⇒ ` a)) ≡ ((ƛ a' ⇒ ` a') · (ƛ a' ⇒ ` a'))
  ∋ refl

-- ** grown-up substitution
{-
_ = (abs a $ ` a) ULC.[ ` b ] ≡ (` b)
  ∋ refl
```

```
_ = (abs b $ ` b) ULC.[ ` c ] ≡ (` c)
  ∋ refl

_ = (abs c $ ` c · ` a) ULC.[ ` b ] ≡ (` b · ` a)
  ∋ refl

_ = (abs b $ ` a) ULC.[ ` b ] ≡ (` a)
  ∋ refl

_ = (abs b $ ` a · ` b) ULC.[ ` c ] ≡ (` a · ` c)
  ∋ refl

_ = (abs b $ ƛ a ⇒ ` a) ULC.[ ` b ] ≡ (ƛ c'' ⇒ ` c'')
  ∋ refl
```

```
-}
```

ULC/Measure.agda

```
open import Prelude.Init; open SetAsType
open import Prelude.DecEq
open import Prelude.General
open import Prelude.Closures
open import Prelude.InferenceRules
open import Prelude.Decidable
open import Prelude.Membership
open import Prelude.Setoid
open import Prelude.Bifunctor
open import Prelude.Measurable
open import Prelude.Ord
open import Prelude.InfEnumerable

-- ** Sizes for λ-terms, to be used as termination measures.
module ULC.Measure (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enumerable∞
```

```
open import ULC.Base Atom
open import Nominal Atom

private variable A : Type ℓ

-- module _ {A : Type}
--          ⦃ _ : LawfulSetoid A ⦄ ⦃ _ : Swap A ⦄ ⦃ _ : SwapLa
--          ⦃ _ : FinitelySupported A ⦄
--          ⦃ _ : Measurable A ⦄ where
--   mapAbs′ : (x' : Abs A) → ((x : A) → x <ᵐ x' → A) → Abs A
--   mapAbs′ x' f =
--     let a = fresh-var x'
--     in abs a (f (conc x' a) {!!})

instance
```

```
  Measurable-Abs : ⦃ Measurable A ⦄ → Measurable (Abs A)
  Measurable-Abs .∣_∣ (abs _ t) = suc ∣ t ∣

  Measurable-Shape : Measurable TermShape
  Measurable-Shape .∣_∣ = λ where
    `■ → 1
    (l · r) → ∣ l ∣ + ∣ r ∣
    (ƛ s) → suc ∣ s ∣

  Measurable-Term : Measurable Term
  Measurable-Term .∣_∣ = ∣_∣ ∘ shape

-- swapping does not alter the size of a term
swap≡ : ∀ (t : Term) → ∣ swap x y t ∣ ≡ ∣ t ∣
swap≡ = cong ∣_∣ ∘ sym ∘ swap-shape≡ _ _
```

```
-- concretion reduces the size of a term by one
conc≡ : ∀ (f : Abs Term) x → ∣ conc f x ∣ ≡ Nat.pred ∣ f ∣
conc≡ (abs x t) _ = swap≡ t

-- ⇒ a concretized term is strictly smaller than the original
conc< : ∀ (f : Abs Term) x → ∣ conc f x ∣ < ∣ f ∣
conc< f x rewrite conc≡ f x = Nat.≤-refl

-- the size of a term is always positive
measure⁺ : ∀ (t : Term) → ∣ t ∣ > 0
measure⁺ (` _) = s≤s z≤n
measure⁺ (l · m) with ∣ l ∣ | measure⁺ l | ∣ m ∣ | measure⁺ m
... | suc l′ | _ | suc m′ | _ = s≤s z≤n
measure⁺ (ƛ _) = s≤s z≤n

-- the size of an application's left operand is strictly small
```

```
_·<ˡ_ : ∀ (L M : Term) → L < (L · M)
_ ·<ˡ M = Nat.m<m+n _ (measure⁺ M)
```

```
-- the size of an application's right operand is strictly smal
_·<ʳ_ : ∀ (L M : Term) → M < (L · M)
L ·<ʳ _ = Nat.m<n+m _ (measure⁺ L)
```

ULC/Reduction.agda

```
{-# OPTIONS --allow-unsolved-metas #-}
open import Prelude.Init hiding ([_]); open SetAsType
open L.Mem
open import Prelude.DecEq
open import Prelude.InfEnumerable
open import Prelude.InferenceRules
open import Prelude.Closures
open import Prelude.Decidable
open import Prelude.Functor
open import Prelude.Bifunctor
open import Prelude.Setoid
open import Prelude.Lists.Membership
open import Prelude.Lists.Dec

module ULC.Reduction (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enumerabl
```

```
open import ULC.Base         Atom ⦃ it ⦄ hiding (z; x′)
open import ULC.Measure Atom ⦃ it ⦄
open import ULC.Alpha      Atom ⦃ it ⦄
open import ULC.Substitution Atom ⦃ it ⦄
open import Nominal         Atom ⦃ it ⦄

-- ** Reduction rules.
infix 0 _—→_
data _—→_ : Rel₀ Term where
  β :
      ──────────────────────────────
      (ƛ x ⇒ t) · t′ —→ t [ x / t′ ]
      -- (ƛ t̂) · t → t̂ [ t ] -- "grown-up" substitution

  ζ_ :
```

```
    t —→ t′
    ────────────────────
    ƛ x ⇒ t —→ ƛ x ⇒ t′

  ξ₁_ :
    t —→ t′
    ──────────────────
    t · t″ —→ t′ · t″

  ξ₂_ :
    t —→ t′
    ──────────────────
    t″ · t —→ t″ · t′

postulate
  supp-conc : supp (conc t̂ y) ⊆ y ∷ supp (t̂ .term)
```

```
  supp-conc♯ : t̂ .atom ∉ supp (conc t̂ y)

{-# TERMINATING #-}
supp-[] : supp (t [ x / t′ ]) ⊆ supp t ++ supp t′
supp-[] {` y}{x}{t′}
  with y ≟ x
... | yes refl
  = ∈-++⁺ʳ _
... | no x≠y
  = λ where (here refl) → here refl
supp-[] {L · M}{x}{t′} x∈
  with ∈-++⁻ (supp (L [ x / t′ ])) (∈-nub⁻ x∈)
... | inj₁ x∈ = case ∈-++⁻ (supp L) $ supp-[] {t = L} x∈ of λ where
  (inj₁ x∈) → ∈-++⁺ˡ $ ∈-nub⁺ $ ∈-++⁺ˡ x∈
  (inj₂ x∈) → ∈-++⁺ʳ (nub $ supp L ++ supp M) x∈
```

```
... | inj₂ x∈ = case ∈-++⁻ (supp M) $ supp-[] {t = M} x∈ of λ where
  (inj₁ x∈) → ∈-++⁺ˡ $ ∈-nub⁺ $ ∈-++⁺ʳ (supp L) x∈
  (inj₂ x∈) → ∈-++⁺ʳ (nub $ supp L ++ supp M) x∈
supp-[] {t₀@(ƛ t̂@(abs _ t))}{x}{t′} {x′} x∈
  with y ← freshAtom (x ∷ supp t̂ ++ supp t′)
  with x∈ , x≢ ← ∈-filter⁻ (¬? ∘ (_≟ y)) {xs = supp (conc t̂ y [ x / t′ ]
  with ∈-++⁻ (supp $ conc t̂ y) $ supp-[] {t = conc t̂ y} x∈
... | inj₂ x∈ = ∈-++⁺ʳ (supp t₀) x∈
... | inj₁ x∈
  with x∉ ← supp-conc♯ {t̂ = t̂} {y = y}
  with supp-conc {t̂}{y} x∈
... | 𝟘 = ⊥-elim $ x≢ refl
... | there x∈′
  with x′ ≟ t̂ .atom
... | yes refl = ⊥-elim $ x∉ x∈
```

```
... | no x∌ = ∈-++⁺ˡ {xs = supp t₀}
          $ ∈-filter⁺ (¬? ∘ (_≟ t̂ .atom)) {xs = supp t} x∈′ x∌

∉-[] :
  • y ∉ supp t
  • y ∉ supp t′
    ─────────────────────────
    y ∉ supp (t [ x / t′ ])
∉-[] {t = t} y∉ y∉′ = ∉-++⁺ y∉ y∉′ ∘ supp-[] {t = t}

fresh-⟶ :
  N ⟶ N′
  ─────────────────────────────
  (_∉ supp N) ⊆¹ (_∉ supp N′)
fresh-⟶ (β {t = t}) x∉ =
  let x∉ , x∉′ = ∉-++⁻ $ ∉-nub⁻ x∉
```

```
  in ∉-[] {t = t} (∉-filter⁻ (¬? ∘ (_≟_)) x∉ {!!}) x∉′
fresh-→ (ζ p) x∉ =
  let x∉ = ∉-filter⁻ (¬? ∘ (_≟_)) x∉ {!!}
  in ∉-filter⁺ (¬? ∘ (_≟_)) $ fresh-→ p x∉
fresh-→ (ξ₁ p) x∉ =
  let x∉ , x∉″ = ∉-++⁻ $ ∉-nub⁻ x∉
      x∉′ = fresh-→ p x∉
  in ∉-nub⁺ $ ∉-++⁺ x∉′ x∉″
fresh-→ (ξ₂_ {t″ = t″} p) x∉ =
  let x∉″ , x∉ = ∉-++⁻ {xs = supp t″} $ ∉-nub⁻ x∉
      x∉′ = fresh-→ p x∉
  in ∉-nub⁺ $ ∉-++⁺ x∉″ x∉′

open ReflexiveTransitiveClosure _→_
```

```
appL-cong :
  L —↠ L′
  ---------------
  L · M —↠ L′ · M
appL-cong (L ∎) = L · _ ∎
appL-cong (L —→⟨ r ⟩ rs) = L · _ —→⟨ ξ₁ r ⟩ appL-cong rs

appR-cong :
  M —↠ M′
  ---------------
  L · M —↠ L · M′
appR-cong (M ∎) = _ · M ∎
appR-cong (M —→⟨ r ⟩ rs) = _ · M —→⟨ ξ₂ r ⟩ appR-cong rs

abs-cong :
  N —↠ N′
```

```
  ƛ x ⇒ N —↠ ƛ x ⇒ N′
abs-cong (M ∎) = ƛ _ ⇒ M ∎
abs-cong (L —→⟨ r ⟩ rs) = ƛ _ ⇒ L —→⟨ ζ r ⟩ abs-cong rs

private
  postulate
    s z n m : Atom
    s≢z : s ≢ z

  infixr 2 _—→≡⟨_⟩_ _—→≡⟨⟩_
  _—→≡⟨_⟩_ : (t : Term) → t ≡ t′ → t′ —↠ t″ → t —↠ t″
  _ —→≡⟨ refl ⟩ p = p

  _—→≡⟨⟩_ : (t : Term) → t —↠ t′ → t —↠ t′
  _ —→≡⟨⟩ p = _ —→≡⟨ refl ⟩ p
```

```agda
open import Prelude.General

_ : (ƛ s ⇒ ` s) · ` z —↠ ` z
_ = begin
    (ƛ s ⇒ ` s) · ` z
  —→⟨ β ⟩
    (` s) [ s / ` z ]
  —→≡⟨⟩
    (if s == s then ` z else ` s)
  —→≡⟨ if-true $ cong isYes $ ≟-refl s ⟩
    ` z
  ∎

$z = freshAtom (s ∷ supp (ƛ z ⇒ ` s) ++ (z ∷ []))
```

```
_ : (ƛ s ⇒ ƛ z ⇒ ‘ s) · ‘ z —↠ (ƛ $z ⇒ ‘ z)
_ =
  begin
    (ƛ s ⇒ ƛ z ⇒ ‘ s) · ‘ z
  —→⟨ β ⟩
    (ƛ z ⇒ ‘ s) [ s / ‘ z ]
  —→≡⟨⟩
    (ƛ $z ⇒ conc (abs z $ ‘ s) $z [ s / ‘ z ])
  —→≡⟨⟩
    (ƛ $z ⇒ swap $z z (‘ s) [ s / ‘ z ])
  —→≡⟨ cong (λ ◆ → ƛ $z ⇒ (‘ ◆) [ s / ‘ z ]) $ swap-noop $z z s (λ where
      (here eq) → freshAtom∉ $ here $′ sym eq
      (there (here eq)) → s≠z eq) ⟩
    (ƛ $z ⇒ ‘ s [ s / ‘ z ])
  —→≡⟨ cong (λ ◆ → ƛ $z ⇒ ◆) $ if-true $ cong isYes $ ≟-refl s ⟩
```

```
    (ƛ $z ⇒ ` z)


zeroᶜ = ƛ s ⇒ ƛ z ⇒ ` z
sucᶜ  = ƛ n ⇒ ƛ s ⇒ ƛ z ⇒ ` s · (` n · ` s · ` z)

mkNumᶜ : ℕ → Term
mkNumᶜ = λ where
  zero    → zeroᶜ
  (suc n) → sucᶜ · (mkNumᶜ n)

twoᶜ  = ƛ s ⇒ ƛ z ⇒ ` s · (` s · ` z)
fourᶜ = ƛ s ⇒ ƛ z ⇒ ` s · (` s · (` s · (` s · ` z)))
plusᶜ = ƛ m ⇒ ƛ n ⇒ ƛ s ⇒ ƛ z ⇒ (` m · ` s · (` n · ` s · ` z))
2+2ᶜ : Term
2+2ᶜ = plusᶜ · twoᶜ · twoᶜ
```

```
{-
  _ : 2+2ᶜ —↠ fourᶜ
  _ =
    begin
      (plusᶜ · twoᶜ) · twoᶜ
    ≡⟨⟩
      ( (ƛ m ⇒ ƛ n ⇒ ƛ s ⇒ ƛ z ⇒ (` m · ` s · (` n · ` s · ` z
      · (ƛ s ⇒ ƛ z ⇒ ` s · (` s · ` z))
      ) · twoᶜ
    —→⟨ ξ₁ β ⟩
      let
        n′ = freshAtom (m ∷ n ∷ supp (ƛ s ⇒ ƛ z ⇒ (` m · ` s ·
        s′ = freshAtom (m ∷ {!n ∷ ?!})
        z′ = freshAtom (m ∷ {!!})
```

```
  in
  ( ƛ n′ ⇒ ƛ s′ ⇒ ƛ z′ ⇒ {!!}
      -- ((ƛ s′ ⇒ ƛ z′ ⇒ ` s′ · (` s′ · ` z′)) · ` s′ · (`
  ) · twoᶜ
—→⟨ {!!} ⟩
-- —→⟨ ξ₁ β ⟩
--   ( (ƛ n ⇒ ƛ s ⇒ ƛ z ⇒ (` m · ` s · (` n · ` s · ` z)))
--   ) · twoᶜ
--   (ƛ n ⇒ ƛ s ⇒ ƛ z ⇒ twoᶜ · ` s · (` n · ` s · ` z)) ·
-- —→⟨ β ⟩
--   ƛ s ⇒ ƛ z ⇒ twoᶜ · ` s · (twoᶜ · ` s · ` z)
-- —→⟨ ζ ζ ξ₁ β ⟩
--   ƛ s ⇒ ƛ z ⇒ (ƛ z ⇒ ` s · (` s · ` z)) · (twoᶜ · ` s ·
-- —→⟨ ζ ζ β ⟩
--   ƛ s ⇒ ƛ z ⇒ ` s · (` s · (twoᶜ · ` s · ` z))
```

```
    -- —→⟨ ζ ζ ξ₂ ξ₂ ξ₁ β ⟩
    --   ƛ s ⇒ ƛ z ⇒ ` s · (` s · ((ƛ z ⇒ ` s · (` s · ` z)) ·
    -- —→⟨ ζ ζ ξ₂ ξ₂ β ⟩
      ƛ s ⇒ ƛ z ⇒ ` s · (` s · (` s · (` s · ` z)))
    ≡⟨⟩
      fourᶜ
    ∎
-}

-- ** Specific term forms.
Neutral Normal Value : Pred₀ Term
Neutral = λ where
  (` _) → ⊤
  (L · M) → Neutral L × Normal M
  _ → ⊥
```

```
Normal M = Neutral M ⊎ (case M of λ where
  (ƛ x ⇒ N) → Normal N
  _ → ⊥)
Value = λ where
  (ƛ _ ⇒ _) → ⊤
  _ → ⊥

-- ** Progress.

pattern step_ x = inj₁ x
pattern ⟨+_ x = inj₁ x
pattern done_ x = inj₂ x
pattern +⟩_ x = inj₂ x
infix 0 step_ done_ ⟨+_ +⟩_

progress : (M : Term) → ∃ (M —→_) ⊎ Normal M
```

```
progress (` _) = done auto
progress (ƛ _ ⇒ N) with progress N
... | step (_ , N→) = ⟨+ -, ζ N→
... | done N∅ = +⟩ +⟩ N∅
progress (` _ · N) with progress N
... | step (_ , N→) = ⟨+ -, ξ₂ N→
... | done N∅ = +⟩ ⟨+ auto , N∅
progress ((ƛ _) · _) = ⟨+ -, β
progress (L@(_ · _) · M) with progress L
... | step (_ , L→) = ⟨+ -, ξ₁ L→
... | done (⟨+ L∅) with progress M
...   | step (_ , M→) = ⟨+ -, ξ₂ M→
...   | done M∅ = +⟩ ⟨+ (L∅ , M∅)

-- ** Evaluation.
```

```
Gas = ℕ

eval : Gas → (L : Term) → Maybe (∃ λ N → Normal N × (L —↠ N))
eval 0 L = nothing
eval (suc m) L with progress L
... | done N⌀ = just $ -, N⌀ , (L ∎)
... | step (M , L→) = map₂′ (map₂ (L —→⟨ L→ ⟩_)) <$> eval m M

{- Ctrl-c Ctrl-n "eval 100 2+2ᶜ" -}

-- ** Confluence

infix -1 _⇛_

data _⇛_ : Rel₀ Term where

  -- TODO: adding α-renaming is morally OK
```

```
ν⇛ : ` x ⇛ ` x

ζ⇛ :
    N ⇛ N′
    ──────────────────────
    ƛ x ⇒ N ⇛ ƛ x ⇒ N′
-- И a. N̂ @ a ⇛ N̂′ @ a
-- ──────────────────────
-- ƛ N̂ ⇛ ƛ N̂′
-- or use the dual ∃??

ξ⇛ :
  • L ⇛ L′
  • M ⇛ M′
    ──────────────────
```

```
      L · M ⇛ L′ · M′

  β⇛ :
    • N ⇛ N′
    • M ⇛ M′
      ────────────────────────────────────
      -- (ƛ x ⇒ N) · M ⇛ N′ [ x / M′ ]
      (ƛ x ⇒ N) · M ⇛ N′ [ x / M′ ]

open ReflexiveTransitiveClosure _⇛_
  renaming ( _—→⟨_⟩_ to _⇛⟨_⟩_; _∎ to _⇛∎; _—↠_ to _⇛*_
           ; _—↠⟨_⟩_ to _⇛*⟨_⟩_; —↠-trans to ⇛*-trans
           )

par-refl : M ⇛ M
par-refl {` x} = ν⇛
```

```
par-refl {ƛ N} = ζ⇛ par-refl
par-refl {L · M} = ξ⇛ par-refl par-refl

beta-par :
  M —→ N
  ------
  M ⇛ N
beta-par = λ where
  (ξ₁ r) → ξ⇛ (beta-par r) par-refl
  (ξ₂ r) → ξ⇛ par-refl (beta-par r)
  β      → β⇛ par-refl par-refl
  (ζ r)  → ζ⇛ (beta-par r)

beta-pars :
  M —↠ N
  ------
```

```
  M ⇛* N
beta-pars = λ where
  (_ ∎) → _ ∎
  (L —→⟨ b ⟩ bs) → L ⇛⟨ beta-par b ⟩ beta-pars bs

betas-pars :
  M —↠ N
  ------
  M ⇛* N
betas-pars = λ where
  (_ ∎) → _ ∎
  (_ —→⟨ p ⟩ bs) → _ ⇛⟨ beta-par p ⟩ betas-pars bs

par-betas :
  M ⇛ N
  ------
```

```
  M —↠ N
par-betas (ν⇛ {x = x}) = (` x) ∎
par-betas (ζ⇛ p) = abs-cong (par-betas p)
par-betas {L · M} (ξ⇛ {L = L}{L′}{M}{M′} p₁ p₂) =
  begin L · M   —↠⟨ appL-cong (par-betas p₁) ⟩
        L′ · M  —↠⟨ appR-cong (par-betas p₂) ⟩
        L′ · M′ ∎
par-betas {(ƛ x ⇒ N) · M} (β⇛ {N′ = N′}{M′ = M′} p₁ p₂) =
  begin (ƛ x ⇒ N) · M   —↠⟨ appL-cong (abs-cong (par-betas p₁)) ⟩
        (ƛ x ⇒ N′) · M  —↠⟨ appR-cong (par-betas p₂) ⟩
        (ƛ x ⇒ N′) · M′ —→⟨ β ⟩
        N′ [ x / M′ ]  ∎

pars-betas :
  M ⇛* N
```

```
  ────────
  M —↠ N
pars-betas (_ ⇛∎) = _ ∎
pars-betas (_ ⇛⟨ p ⟩ ps) = —↠-trans (par-betas p) (pars-betas ps)

sub-abs :
  N ⇛ N′
  ──────────────────────
  (ƛ x ⇒ N) ⇛ (ƛ x ⇒ N′)
sub-abs = ζ⇛

sub-swap :
  N ⇛ N′
  ──────────────────────────
  swap x y N ⇛ swap x y N′
  -- Equivariant² _⇛_
```

```
sub-swap ν⇛ = ν⇛
sub-swap (ζ⇛ p) = ζ⇛ (sub-swap p)
sub-swap (ξ⇛ p q) = ξ⇛ (sub-swap p) (sub-swap q)
sub-swap {x = 𝕒}{𝕓} (β⇛ {N}{N′}{M}{M′}{x} p q) =
  {- β⇛ :
        • N ⇛ N′
        • M ⇛ M′
          ────────────────────────────
          (ƛ x ⇛ N) · M ⇛ N′ [ x / M′ ]
   -}
  qed
  where
    a↔b = swap 𝕒 𝕓
    a↔b↓ = (Atom → Atom) ∋ swap 𝕒 𝕓
```

```
-- ƛN⇛ : (ƛ x ⇒ N) ⇛ (ƛ x ⇒ N′)
-- ƛN⇛ = ζ⇛ p

-- N⇛ : a↔b (ƛ x ⇒ N) ⇛ a↔b (ƛ x ⇒ N′)
-- N⇛ = sub-swap ƛN⇛

N⇛ : a↔b N ⇛ a↔b N′
N⇛ = sub-swap p

M⇛ : a↔b M ⇛ a↔b M′
M⇛ = sub-swap q

H : a↔b (ƛ x ⇒ N) · a↔b M ⇛ a↔b N′ [ a↔b↓ x / a↔b M′ ]
H = β⇛ N⇛ M⇛

eq≈ : a↔b (N′ [ x / M′ ]) ≈ a↔b N′ [ a↔b↓ x / a↔b M′ ]
-- eq≈ = equivariant [_/_] 𝕒 𝕓
```

```
    eq≈ = swap-subst 𝕒 𝕓 {N′}{x}{M′}

    eq : a↔b (N′ [ x / M′ ]) ≡ a↔b N′ [ a↔b↓ x / a↔b M′ ]
    eq = {!!}

    qed : a↔b (ƛ x ⇒ N) · a↔b M ⇛ a↔b (N′ [ x / M′ ])
    qed = subst (λ ◆ → a↔b (ƛ x ⇒ N) · a↔b M ⇛ ◆) (sym eq) H

postulate
  sub-swap˘ :
    swap x y N ⇛ swap x y N′
    ────────────────────────
    N ⇛ N′

-- sub-conc : ∀ {f f′ : Abs Term} →
--   ƛ f ⇛ ƛ f′
```

```agda
--    ─────────────────────
--    conc f x ⇛ conc f′ x
-- sub-conc (ζ⇛ p) = sub-swap p

{-# TERMINATING #-}
sub-par :
  • N ⇛ N′
  • M ⇛ M′
    ───────────────────────────
    N [ x / M ] ⇛ N′ [ x / M′ ]

sub-par {x = a} (ν⇛ {x = x}) p
  with x ≟ a
... | yes refl = p
... | no  _ = ν⇛
```

```
sub-par (ξ⇛ L→ M→) p =
  ξ⇛ (sub-par L→ p) (sub-par M→ p)

sub-par {M = M}{M′}{a} (ζ⇛ {N}{N′}{x} p) q =
  {- ζ⇛ :
          N ⇛ N′
          ──────────────────
          ƛ x ⇒ N ⇛ ƛ x ⇒ N′
   -}
  qed
  where
    x′  = freshAtom (a ∷ x ∷ supp N ++ supp M)
    x′′ = freshAtom (a ∷ x ∷ supp N′ ++ supp M′)

    x≡ : x′ ≡ x′′
    x≡ = {!!}
```

```
-- p : N ⇛ N′

↔p : swap x x′ N ⇛ swap x x′ N′
↔p = sub-swap p

s↔p′ : swap x x′ (N [ a / M ]) ⇛ swap x x′ (N′ [ a / M′ ])
s↔p′ = {!sub-par ?p (sub-swap q)!} -- sub-par ↔p (sub-swap q)

ƛs↔p′ : swap x x′ (ƛ x ⇒ N [ a / M ])
      ⇛ swap x x′ (ƛ x ⇒ N′ [ a / M′ ])
ƛs↔p′ = sub-abs s↔p′

-- s↔p : swap x x′ N [ a / M ] ⇛ swap x x′ N′ [ a / M′ ]
-- s↔p = sub-par ↔p q
```

```
-- ƛs↔p : (ƛ x′ ⇒ swap x x′ N [ a / M ]) ⇛ (ƛ x′ ⇒ swap x x
-- ƛs↔p = sub-abs s↔p

-- ƛs↔p′ : (ƛ x′ ⇒ swap x x′ N [ a / M ])
--         ⇛ (ƛ x′′ ⇒ swap x x′′ N′ [ a / M′ ])
-- ƛs↔p′ = subst (λ ◆ → (ƛ x′ ⇒ swap x x′ N [ a / M ])
--                           ⇛ (ƛ ◆ ⇒ swap x ◆ N′ [ a / M′ ]))
--                      x≡ ƛs↔p

-- ƛs↔p′′ : (ƛ swap x x′ x ⇒ swap x x′ N [ a / M ])
--          ⇛ (ƛ swap x x′′ x ⇒ swap x x′′ N′ [ a / M′ ])
-- ƛs↔p′′ rewrite swapˡ x x′ | swapˡ x x′′ = ƛs↔p′

qed : (ƛ x ⇒ N) [ a / M ] ⇛ (ƛ x ⇒ N′) [ a / M′ ]
-- qed : (ƛ N̂) [ a / M ] ⇛ (ƛ N̂′) [ a / M′ ]
qed = {!!} -- sub-swap˘ {!ƛs↔p′!} -- ƛs↔p′
```

```
sub-par {M = X}{X′}{𝕒} (β⇛ {N}{N′}{M}{M′}{x} p q) pq =
  {- β⇛ :
       • N ⇛ N′
       • M ⇛ M′
        ──────────────────────────────
        (ƛ x ⇒ N) · M ⇛ N′ [ x / M′ ]
   -}
  qed
  where
    x′ = freshAtom (𝕒 ∷ supp (ƛ x ⇒ N) ++ supp X)

    _ : ((ƛ x ⇒ N) · M) [ 𝕒 / X ]
      ≡ (ƛ x′ ⇒ swap x′ x N [ 𝕒 / X ]) · (M [ 𝕒 / X ])
    _ = refl
```

```
N⇛ : swap x′ x N [ a / X ] ⇛ swap x′ x N′ [ a / X′ ]
N⇛ = sub-par (sub-swap p) pq

M⇛ : M [ a / X ] ⇛ M′ [ a / X′ ]
M⇛ = sub-par q pq

qed′ : ((ƛ x ⇒ N) · M) [ a / X ]
     ⇛ swap x′ x N′ [ a / X′ ] [ x′ / M′ [ a / X′ ] ]
qed′ = β⇛ N⇛ M⇛

eq≈ : swap x′ x N′ [ a / X′ ] [ x′ / M′ [ a / X′ ] ]
    ≈ N′ [ x / M′ ] [ a / X′ ]
eq≈ =
  ≈-Reasoning.begin
    swap x′ x N′ [ a / X′ ] [ x′ / M′ [ a / X′ ] ]
  ≈⟨ subst-commute {swap x′ x N′} ⟩
```

```
      swap x′ x N′ [ x′ / M′ ] [ 𝕒 / X′ ]
    ≈⟨ cong-subst $ swap∘subst {x′}{x}{N′}{M′} ⟩
      N′ [ x / M′ ] [ 𝕒 / X′ ]
    ≈-Reasoning.∎ where open ≈-Reasoning

  eq : swap x′ x N′ [ 𝕒 / X′ ] [ x′ / M′ [ 𝕒 / X′ ] ]
    ≡ N′ [ x / M′ ] [ 𝕒 / X′ ]
  eq = {!!}

  qed : ((ƛ x ⇒ N) · M) [ 𝕒 / X ] ⇛ N′ [ x / M′ ] [ 𝕒 / X′ ]
  qed = subst (_⇛_) eq qed′

_⁺ : Op₁ Term
_⁺ = λ where
  (` x)      → ` x
  (ƛ x ⇒ M) → ƛ x ⇒ (M ⁺)
```

```
  ((ƛ x ⇒ N) · M) → N ⁺ [ x / M ⁺ ]
  (L · M)  → (L ⁺) · (M ⁺)

par-◇ :
  M ⇛ N
  ----------
  N ⇛ M ⁺
par-◇ = λ where
  ν⇛ → ν⇛
  (ζ⇛ p) → ζ⇛ (par-◇ p)
  (β⇛ p p′) → sub-par (par-◇ p) (par-◇ p′)
  (ξ⇛ {_ · _} p p′) → ξ⇛ (par-◇ p) (par-◇ p′)
  (ξ⇛ {` _} p p′) → ξ⇛ (par-◇ p) (par-◇ p′)
  (ξ⇛ {ƛ _} (ζ⇛ p) p′) → β⇛ (par-◇ p) (par-◇ p′)

par-◁ = par-◇
```

```
par-◇ :
  • M ⇛ N
  • M ⇛ N′
    ─────────────────────────────
    ∃ λ L → (N ⇛ L) × (N′ ⇛ L)
par-◇ {M = M} p p′ = M ⁺ , par-⦉ p , par-⦊ p′

strip :
  • M ⇛ N
  • M ⇛* N′
    ─────────────────────────────
    ∃ λ L → (N ⇛* L) × (N′ ⇛ L)
strip mn (_ ⇛∎) = -, (_ ⇛∎) , mn
strip mn (_ ⇛⟨ mm' ⟩ m'n') =
  let _ , ll' , n'l' = strip (par-⦊ mm') m'n'
```

```
  in  -, (_ ⇛⟨ par-⦊ mn ⟩ ll') , n'l'

par-confluence :
  • L ⇛* M₁
  • L ⇛* M₂
    ────────────────────────────
    ∃ λ N → (M₁ ⇛* N) × (M₂ ⇛* N)
par-confluence (_ ⇛∎) p = -, p , (_ ⇛∎)
par-confluence (_ ⇛⟨ L⇛M₁ ⟩ M₁⇛*M₁′) L⇛*M₂ =
  let _ , M₁⇛*N , M₂⇛N = strip L⇛M₁ L⇛*M₂
      _ , M₁′⇛*N′ , N⇛*N′ = par-confluence M₁⇛*M₁′ M₁⇛*N
   in -, M₁′⇛*N′ , (_ ⇛⟨ M₂⇛N ⟩ N⇛*N′)

confluence :
  • L —↠ M₁
  • L —↠ M₂
```

```
    ∃ λ N → (M₁ —↠ N) × (M₂ —↠ N)
confluence L↠M₁ L↠M₂ =
  let _ , M₁⇛N , M₂⇛N = par-confluence (betas-pars L↠M₁) (betas-pars
  in -, pars-betas M₁⇛N , pars-betas M₂⇛N
```

ULC/Substitution.agda

```
{-# OPTIONS --allow-unsolved-metas #-}
-- {-# OPTIONS --auto-inline #-}
open import Prelude.Init hiding ([_]); open SetAsType
open L.Mem
open import Prelude.General
open import Prelude.DecEq
-- open import Prelude.Lists.Dec
-- open import Prelude.Measurable
open import Prelude.InfEnumerable
open import Prelude.Setoid
open import Prelude.InferenceRules

-- ** Substitution.
module ULC.Substitution (Atom : Type) ⦃ _ : DecEq Atom ⦄ ⦃ _ : Enumer

open import ULC.Base  Atom ⦃ it ⦄
```

```
open import ULC.Measure Atom ⦃ it ⦄ ⦃ it ⦄
open import ULC.Alpha Atom ⦃ it ⦄ ⦃ it ⦄
open import Nominal Atom
open import Nominal.Product Atom

-- enforce the Barendregt convention: no shadowing, distinct b
{-# TERMINATING #-}
barendregt : Op₁ Term
barendregt = go []
  where
    go : List Atom → Op₁ Term
    go xs = λ where
      (` x)   → ` x
      (l · r) → go xs l · go xs r
      (ƛ x ⇒ t) → let x′ = freshAtom (xs ++ supp t)
```

```
              in ƛ x′ ⇒ go (x ∷ xs) (swap x′ x t)

infix 6 _[_/_]
{-# TERMINATING #-}
_[_/_] : Term → Atom → Term → Term
(` x) [ 𝕒 / N ] = if x == 𝕒 then N else ` x
(L · M) [ 𝕒 / N ] =
  let L′ = L [ 𝕒 / N ]
      M′ = M [ 𝕒 / N ]
  in L′ · M′
(ƛ t̂) [ 𝕒 / N ] =
  -- let y = fresh-var (𝕒 , t̂ , N)
  let y = freshAtom (𝕒 ∷ supp t̂ ++ supp N)
  in ƛ y ⇒ conc t̂ y [ 𝕒 / N ]

infix 6 _[_]
```

```
_[_] : Abs Term → Term → Term
(abs x t) [ s ] = (ƛ x ⇒ t) [ x / s ]

infix 6 _[_/_]↑
_[_/_]↑ : Abs Term → Atom → Term → Abs Term
(abs 𝕒 t) [ x / N ]↑ = unƛ $ (ƛ 𝕒 ⇒ t) [ x / N ]

{- ** well-founded version
t₀ [ 𝕒 / s ] = <-rec _ go t₀
  module |Substitution| where
    go : ∀ x → (∀ y → y < x → Term) → Term
    go x rec with x
    ... | ` x   = if x == 𝕒 then s else ` x
    ... | l · m = rec l (l ·<ˡ m) · rec m (l ·<ʳ m)
    -- Cannot simply use `ƛ (mapAbs go f)` here; need well-fou
    -- ... | ƛ f   = ƛ mapAbs-Term f (λ t t< → rec t t<)
```

```
    ... | ƛ f =
      let y , _ = fresh (nub $ 𝕒 ∷ supp f ++ supp s)
      in  ƛ y ⇒ rec (conc f y) (conc< f y)
-}

-- ** postulate equivariance of substitution for now...
postulate swap-subst : Equivariant _[_/_]
-- swap-subst = ? -- equivariant _[_/_]

-- ** we will also need the following lemmas for proving Reduc

subst-commute : N [ x / L ] [ y / M [ x / L ] ] ≈ N [ y / M ] [ x / L ]
subst-commute {` n} {x} {L} {y} {M}
  with n ≟ x | n ≟ y
... | yes refl | yes refl
  -- exclude with x ≠ y
```

```
  = {!subst-commute !}
... | yes refl | no n≢y
  rewrite ≟-refl n
  = {!!}
  -- prove with y ♯ L
... | no n≢x | yes refl
  rewrite ≟-refl n
  = ≈-refl
... | no n≢x | no n≢y
  rewrite dec-no (n ≟ x) n≢x .proj₂
    | dec-no (n ≟ y) n≢y .proj₂
    = ≈-refl
subst-commute {Nˡ · Nʳ} {x} {L} {y} {M}
  = ξ≡ (subst-commute {Nˡ}) (subst-commute {Nʳ})
```

```
subst-commute {ƛ t̂} {x} {L} {y} {M}
  with xˡ ← freshAtom (x ∷ supp t̂ ++ supp L)
  -- (ƛ xˡ ⇒ conc t̂ xˡ [ x / L ]) [ y / M [ x / L ] ]
  with yˡ ← freshAtom (y ∷ supp (abs xˡ $ conc t̂ xˡ [ x / L ]) ++ supp (M
  -- ƛ yˡ ⇒ conc (abs xˡ $ conc t̂ xˡ [ x / L ]) yˡ [ y / M [ x
  --       ≡ conc t̂ yˡ [ x / L ] [ y / M [ x / L ] ]

  with yʳ ← freshAtom (y ∷ supp t̂ ++ supp M)
  -- (ƛ yʳ ⇒ conc t̂ yʳ [ y / M ]) [ x / L ]
  with xʳ ← freshAtom (x ∷ supp (abs yʳ $ conc t̂ yʳ [ y / M ]) ++ supp L)
  -- ƛ xʳ ⇒ conc (abs yʳ $ conc t̂ yʳ [ y / M ]) xʳ [ x / L ]
  --       ≡ conc t̂ xʳ [ y / M ] [ x / L ]
  = ζ≡ ( {!!} , (λ z z∉ → {!!} ))

postulate cong-subst : t ≈ t′ → t [ x / M ] ≈ t′ [ x / M ]
```

```
-- {-# TERMINATING #-}
swap∘subst : swap y x N [ y / M ] ≈ N [ x / M ]
swap∘subst {y} {x} {` n} {M}
  with n ≟ x | n ≟ y
... | yes refl | yes refl
  rewrite ≟-refl y
  = ≈-refl
... | yes refl | no n≢y
  rewrite ≟-refl y
  = ≈-refl
... | no n≢x | yes refl
  rewrite dec-no (x ≟ y) (≢-sym n≢x) .proj₂
  = {!!} -- prove with y ♯ N
... | no n≢x | no n≢y
```

```
    rewrite dec-no (n ≟ y) n≢y .proj₂
    = ≈-refl
  swap∘subst {y} {x} {L · R} {M}
    = ξ≡ (swap∘subst {N = L}) (swap∘subst {N = R})
  swap∘subst {y} {x} {ƛ t̂} {M}
  {-
  swap y x (ƛ z ⇒ t) [ y / M ]
  ≡ (ƛ swap y x z ⇒ swap y x t) [ y / M ]
  ≡ let zˡ = freshAtom (y ∷ supp (swap y z (ƛ z ⇒ t) ++ supp M)
    in ƛ zˡ → conc (swap y x $ abs z t) zˡ [ y / M ]
             ≡ conc (swap y x $ abs z t) zˡ [ swap y x x / M ]
             ≡ conc (swap y x $ abs z t) (swap y x zˡ) [ swap y x

  ≡ let zʳ = freshAtom (x ∷ z ∷ supp t ++ supp M)
    in ƛ zʳ → conc (ƛ z ⇒ t) zʳ [ x / M ]
```

```
≡ (ƛ z ⇒ t) [ x / M ]
∎

conc (ƛ zˡ → conc (ƛ swap y x z ⇒ swap y x t) zˡ [ y / M ]) w
≡ swap w zˡ $ conc (ƛ swap y x z ⇒ swap y x t) zˡ [ y / M ]
≡ swap w zˡ $ conc (ƛ swap y x z ⇒ swap y x t) zˡ [ swap y x x

≡ conc (swap w zˡ $ ƛ swap y x z ⇒ swap y x t) w [ y / M ]
≡ conc (ƛ swap w zˡ (swap y x z) ⇒ swap w zˡ $ swap y x t) w [
≡ conc (ƛ swap w zˡ (swap y x z) ⇒ swap w zˡ $ swap y x t) w [

≈?
≡ swap w z t [ x / M ]
≡ conc (ƛ z ⇒ t) w [ x / M ]
≡ conc (swap w zʳ $ ƛ z ⇒ t) w [ x / M ]
```

```
≡ swap w zʳ $ conc (ƛ z ⇒ t) zʳ [ x / M ]
conc (ƛ zʳ → conc (ƛ z ⇒ t) zʳ [ x / M ]) w
-}
  = ζ≡ ( {!!} , λ w w∉ → {!!} )
```