Nominal techniques as an Agda library

Murdoch J. Gabbay, Orestis Melkonian

14 June 2023, TYPES @ Valencia

Motivation

-
-

The nominal universe

Swapping

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

record Swap (A : Type ℓ) : Type ℓ where
  field swap : Atom → Atom → A → A
  ⦅_↔_⦆_ = swap

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

Swapping laws

```
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

instance
  SwapLaws-Atom : SwapLaws Atom
```

Nominal abstraction

```
record Abs (A : Type ℓ) : Type ℓ where
  constructor abs
  field atom : Atom
        term : A

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)

SwapLaws-Abs : SwapLaws (Abs A)
```

The "new" (И) quantifier

```
И : Pred (Pred Atom ℓ) ℓ
И φ = ∃ λ (xs : List Atom) → (∀ y → y ∉ xs → φ y)
```

The notion of finite support

```
module … ⦃ _ : Enumerable∞ Atom ⦄ where
FinSupp : Pred A _
FinSupp x = И² λ 𝕒 𝕓 → swap 𝕓 𝕒 x ≈ x

Equivariant′ : Pred A _
Equivariant′ x = ∃ λ (fin-x : FinSupp x) → fin-x .proj₁ ≡ []

record FinitelySupported : Typeω where
  field ∀fin : Unary.Universal FinSupp

  supp : A → Atoms
  supp = proj₁ ∘ ∀fin

  fresh∉ : (a : A) → ∃ (_∉ supp a)
  fresh∉ = minFresh ∘ supp
```

Finitely supported atoms

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

```
instance
  FinSupp-Abs : ⦃ FinitelySupported A ⦄ → 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)
    ≡⟨⟩
      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
```

Case study: the untyped λ-calculus

λ-TERMS, NOMINALLY

```
data Term : Type where
  `_ : Atom → Term
  _·_ : Term → Term → Term
  ƛ_ : Abs Term → Term
pattern ƛ_⇒_ x y = ƛ abs x y

unquoteDecl Swap-Term = DERIVE Swap [ quote Term , Swap-Term ]
```

α-EQUIVALENCE, NOMINALLY

```
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≡_ : И (λ 𝕩 → conc f 𝕩 ≡α conc g 𝕩)
        ──────────────────────────────
        (ƛ f) ≡α (ƛ g)

pattern ν≡ = ν≈ refl
```

Nominal substitution

```
_[_/_] : Term → Atom → Term → Term
(` x)   [ a / N ] = if x == a then N else ` x
(L · M) [ a / N ] = L [ a / N ] · M [ a / N ]
(ƛ t̂) [ a / N ] = ƛ y ⇒ conc t̂ y [ a / N ]
  where y = fresh-var (a , t̂ , N)

swap-subst    : Equivariant _[_/_]
subst-commute : N [ x / L ] [ y / M [ x / L ] ] ≈ N [ y / M ] [ x / L ]
cong-subst    : t ≈ t′ → t [ x / M ] ≈ t′ [ x / M ]
swap∘subst    : swap y x N [ y / M ] ≈ N [ x / M ]
```

Reduction

```
data _⟶_ : Rel₀ Term where
  β   : ─────────────────────────────
        (ƛ x ⇒ t) · t′ ⟶ t [ x / t′ ]

  ζ_  : t ⟶ t′
        ───────────────────
        ƛ x ⇒ t ⟶ ƛ x ⇒ t′

  ξ₁_ : t ⟶ t′
        ─────────────────
        t · t″ ⟶ t′ · t″

  ξ₂_ : t ⟶ t′
        ─────────────────
        t″ · t ⟶ t″ · t′

open ReflexiveTransitiveClosure _⟶_ using (_⟶*_)
```

Progress

```
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∅)
```

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

Future Work

-
-

Questions?