Nominal techniques as an Agda library

Murdoch J. Gabbay, Orestis Melkonian

20 June 2023, ProgLog seminar @ Chalmers

' While nominal techniques provide a general approach to define and reason about syntax with binders, I actually hesitate to include it here because it seems to require special language support to use effectively, which Agda does not have. '
—Jesper Cockx's blog: 1001 Representations of Syntax with Binding

- Explore another point in the design space of already existing nominal implementations (e.g. Nominal Isabelle)
- Provide a constructive perspective on nominal techniques
- Do this without changing the system itself — as an Agda library
- Make it ergonomic for the user to use the library as a tool for dealing with names (e.g. working on some syntax with binding)
- Mechanise existing (but also new?) meta-theoretical results

The nominal universe

```
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
```

```
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
```

```
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)
```

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

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

FinSupp : Pred A _
FinSupp x = ∀² 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
```

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

```
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

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

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

```
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)
```

```
_[_/_] : Term → Atom → Term → Term
(` x)   [ a / N ] = if x == a then N else ` x
(L · M) [ a / N ] = L [ a / N ] · M [ a / N ]
(λ t̂)   [ a / N ] = λ 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 ]
```

```
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 : (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
```

- More meta-programming automation to minimise overhead
  - corresponding laws and equivariance lemmas follow the same type-directed structure as the swap operation itself
- Another case study on *cut elimination* for first-order logic
  - need to work with entities that are not finitely supported
  - also includes name abstraction over proof trees
- Formalise the constructive *total* concretion function, which seems novel

QUESTIONS?

https://omelkonian.github.io/nominal-agda