

NOMINAL TECHNIQUES AS AN ÅGDA LIBRARY

Murdoch J. Gabbay, Orestis Melkonian

14 June 2023, TYPES @ Valencia

NO NEED FOR SPECIAL LANGUAGE SUPPORT?

‘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

MOTIVATION

- 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

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 rel ℓ) where
  field
    cong-swap :  $x \approx y \rightarrow ((a \leftrightarrow b) x \approx (a \leftrightarrow b) y)$ 
    swap-id    :  $((a \leftrightarrow a) x \approx x)$ 
    swap-rev   :  $((a \leftrightarrow b) x \approx (b \leftrightarrow a) x)$ 
    swap-sym   :  $((a \leftrightarrow b) ((b \leftrightarrow a) x) \approx x)$ 
    swap-swap  :  $((a \leftrightarrow b) ((c \leftrightarrow d) x) \approx (( (a \leftrightarrow b) c \leftrightarrow (a \leftrightarrow b) d ) (a \leftrightarrow b) x)$ 

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” (\mathbb{N}) QUANTIFIER

$\mathbb{N} : \text{Pred} (\text{Pred } \text{Atom } \ell) \ell$

$\mathbb{N} \varphi = \exists \lambda (xs : \text{List } \text{Atom}) \rightarrow (\forall y \rightarrow y \notin xs \rightarrow \varphi y)$

THE NOTION OF FINITE SUPPORT

```
module ... { _ : Enumerable∞ Atom } where
```

```
FinSupp : Pred A _
```

```
FinSupp x =  $\mathbb{N}^2 \rightarrow \text{swap } b\ a\ x \approx x$ 
```

```
Equivariant' : Pred A _
```

```
Equivariant' x =  $\exists \lambda (fin-x : \text{FinSupp } x) \rightarrow fin-x.\text{proj}_1 \equiv []$ 
```

```
record FinitelySupported : Typew where
```

```
  field  $\forall fin : \text{Unary.Universal FinSupp}$ 
```

```
  supp : A → Atoms
```

```
  supp =  $\text{proj}_1 \circ \forall fin$ 
```

```
  fresh $\notin$  : (a : A) →  $\exists (\_ \notin \text{supp } a)$ 
```

```
  fresh $\notin$  = minFresh ∘ supp
```

instance

FinSupp-Atom : FinitelySupported Atom

FinSupp-Atom . $\forall \text{fin } a = [a], \lambda _ _ y \notin z \notin \rightarrow$
swap-noop $_ _ _ \lambda$ where $0 \rightarrow z \notin 0; 1 \rightarrow y \notin 0$

FINITELY SUPPORTED ABSTRACTIONS

instance

FinSupp-Abs : { FinitelySupported A } \rightarrow FinitelySupported (Abs A)

FinSupp-Abs . \forall fin (abs x t) = let xs , p = \forall fin t in

x :: xs , $\lambda y z y \not\in z \not\in \rightarrow$

begin

($\llbracket z \leftrightarrow y \rrbracket$) (abs x t)

$\equiv \langle \rangle$

abs ($\llbracket z \leftrightarrow y \rrbracket$ x) ($\llbracket z \leftrightarrow y \rrbracket$ t)

$\equiv \langle$ cong ($\lambda \blacklozenge \rightarrow$ abs \blacklozenge ($\llbracket z \leftrightarrow y \rrbracket$ t))

\$ swap-noop z y x (λ where $\emptyset \rightarrow z \not\in \emptyset$; $\mathbb{1} \rightarrow y \not\in \emptyset$) \rangle

abs x ($\llbracket z \leftrightarrow y \rrbracket$ t)

$\approx \langle$ cong-abs \$ p y z ($y \not\in \circ$ there) ($z \not\in \circ$ there) \rangle

abs x t

■ where open \approx -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
```

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

data $_ \equiv \alpha _ : \text{Term} \rightarrow \text{Term} \rightarrow \text{Type}_0$ where

$v \approx : x \approx y$

$$\frac{}{\lambda x \equiv \alpha \lambda y}$$

$\xi \equiv : \bullet L \equiv \alpha L'$

$\bullet M \equiv \alpha M'$

$$\frac{}{(L \cdot M) \equiv \alpha (L' \cdot M')}$$

$\zeta \equiv _ : \mathbb{N} (\lambda x \rightarrow \text{conc } f \ x \equiv \alpha \text{conc } g \ x)$

$$\frac{}{(\lambda x f) \equiv \alpha (\lambda x g)}$$

NOMINAL SUBSTITUTION

$_[-/_-] : \text{Term} \rightarrow \text{Atom} \rightarrow \text{Term} \rightarrow \text{Term}$
 $(\backslash x) \ [a / N] = \text{if } x == a \text{ then } N \text{ else } \backslash x$
 $(L \cdot M) \ [a / N] = L \ [a / N] \cdot M \ [a / N]$
 $(\lambda \hat{t}) \ [a / N] = \lambda y \Rightarrow \text{conc } \hat{t} \ y \ [a / N]$
 where $y = \text{fresh-var } (a, \hat{t}, N)$

$\text{swap-subst} \quad : \text{Equivariant } _[-/_-]$
 $\text{subst-commute} : N \ [x / L] \ [y / M \ [x / L]] \approx N \ [y / M] \ [x / L]$
 $\text{cong-subst} \quad : t \approx t' \rightarrow t \ [x / M] \approx t' \ [x / M]$
 $\text{swap} \circ \text{subst} \quad : \text{swap } y \ x \ N \ [y / M] \approx N \ [x / M]$

REDUCTION

data $_ \rightarrow _ : \text{Rel}_0 \text{ Term}$ where

β : $\frac{}{(\lambda x \Rightarrow t) \cdot t' \rightarrow t [x / t']}$

$\zeta_ :$ $t \rightarrow t'$
 $\frac{}{\lambda x \Rightarrow t \rightarrow \lambda x \Rightarrow t'}$

$\xi_{1_} :$ $t \rightarrow t'$
 $\frac{}{t \cdot t'' \rightarrow t' \cdot t''}$

$\xi_{2_} :$ $t \rightarrow t'$
 $\frac{}{t'' \cdot t \rightarrow t'' \cdot t'}$

open ReflexiveTransitiveClosure $_ \rightarrow _$ using $(_ \twoheadrightarrow _)$

PROGRESS

```
progress : (M : Term) → ∃ (M → _) ⊔ Normal M
progress (`_) = done auto
progress (λ _ ⇒ 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 ((λ _) . _) = ⟨+ -, 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 :

- $L \twoheadrightarrow M_1$
- $L \twoheadrightarrow M_2$

$$\exists \lambda N \rightarrow (M_1 \twoheadrightarrow N) \times (M_2 \twoheadrightarrow N)$$

confluence $L \Rightarrow M_1 \quad L \Rightarrow M_2 =$

let

$L \Rightarrow^* M_1 \quad , \quad L \Rightarrow^* M_2 \quad = \text{betas-pars } L \Rightarrow M_1 \quad , \quad \text{betas-pars } L \Rightarrow M_2$

$- \quad , \quad M_1 \Rightarrow N \quad , \quad M_2 \Rightarrow N = \text{par-confluence } L \Rightarrow^* M_1 \quad L \Rightarrow^* M_2$

in

$- \quad , \quad \text{pars-betas } M_1 \Rightarrow N \quad , \quad \text{pars-betas } M_2 \Rightarrow N$

- 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>