

NOMINAL TECHNIQUES AS AN ĀGDA 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 ( $\ell$   $\cup$   $\ell$ ) 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

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

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

THE NOTION OF FINITE SUPPORT

```
module ... { _ : Enumerable $\infty$  Atom } where
```

```
FinSupp : Pred A _
```

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

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

```
Equivariant' x =  $\exists$   $\lambda$  (fin-x : FinSupp x)  $\rightarrow$  fin-x .proj1  $\equiv$  []
```

```
record FinitelySupported : Type $w$  where
```

```
  field  $\forall$  fin : Unary.Universal FinSupp
```

```
  supp : A  $\rightarrow$  Atoms
```

```
  supp = proj1  $\circ$   $\forall$  fin
```

```
  fresh $\notin$  : (a : A)  $\rightarrow$   $\exists$  (_ $\notin$  supp a)
```

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

instance

FinSupp-Atom : FinitelySupported Atom

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

instance

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

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

$$x :: xs , \lambda y \ z \ y \notin z \notin \rightarrow$$

begin

$$((z \leftrightarrow y)) (\text{abs } x \ t)$$
 $\equiv \langle \rangle$

$$\text{abs } ((z \leftrightarrow y) \ x) \ ((z \leftrightarrow y) \ t)$$

$$\equiv \langle \text{cong } (\lambda \diamond \rightarrow \text{abs } \diamond \ ((z \leftrightarrow y) \ t))$$

$$\ \$ \text{swap-noop } z \ y \ x \ (\lambda \text{where } \emptyset \rightarrow z \notin \emptyset ; \mathbb{1} \rightarrow y \notin \emptyset) \ \rangle$$

$$\text{abs } x \ ((z \leftrightarrow y) \ t)$$

$$\approx \langle \text{cong-abs } \$ \ p \ y \ z \ (y \notin \circ \text{there}) \ (z \notin \circ \text{there}) \ \rangle$$

$$\text{abs } x \ t$$

■ 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
```

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

data $_ \equiv \alpha _$: Term \rightarrow Term \rightarrow Type₀ where

$v \approx$: $x \approx y$

$$\lambda x \equiv \alpha \lambda y$$

$\xi \equiv$: • $L \equiv \alpha L'$
• $M \equiv \alpha M'$

$$(L \cdot M) \equiv \alpha (L' \cdot M')$$

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

$$(\lambda f) \equiv \alpha (\lambda g)$$

NOMINAL SUBSTITUTION

$_[-/_-]$: $\text{Term} \rightarrow \text{Atom} \rightarrow \text{Term} \rightarrow \text{Term}$
 $(`x) [a / N] = \text{if } x == a \text{ then } N \text{ else } `x$
 $(L \cdot M) [a / N] = L [a / N] \cdot M [a / N]$
 $(\lambda \hat{t}) [a / N] = \lambda y \Rightarrow \text{conc } \hat{t} y [a / N]$
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 : $\frac{}{(\lambda x \Rightarrow t) \cdot t' \rightarrow t [x / t']}$

ζ_{-} : $t \rightarrow t'$

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

ξ_{1-} : $t \rightarrow t'$

$\frac{}{t \cdot t'' \rightarrow t' \cdot t''}$

ξ_{2-} : $t \rightarrow t'$

$\frac{}{t'' \cdot t \rightarrow t'' \cdot t'}$

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

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 $L \Rightarrow M_1 \ L \Rightarrow M_2 =$

let

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

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

in

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

FUTURE WORK

- 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>