Nominal techniques as an Agda library

Murdoch J. Gabbay[1] and Orestis Melkonian[2]

[1] Heriot-Watt University, Scotland
[2] University of Edinburgh, Scotland

Introduction Nominal techniques [8] provide a mathematically principled approach to dealing with names and variable binding in programming languages. However, integrating these ideas in a practical and widespread toolchain has been slow, and we perceive a chicken-and-egg problem: there are no users for nominal techniques, because nobody has implemented them, and nobody implements them because there are no users. This is a pity, but it leaves a positive opportunity to set up a virtuous circle of broader understanding, adoption, and application of this beautiful technology.

This paper explores an attempt to make nominal techniques accessible as a library in the Agda proof assistant and programming language [9], which can be viewed as a port of the first author's Haskell `nom` package [6], although that would be an injustice as its purpose is two-fold:

1. provide a convenient library to use nominal techniques in Your Own Agda Formalisation
2. study the meta-theory of nominal techniques in a rigorous and *constructive* way

A solution to Goal 1 must be ergonomic, meaning that a *technical* victory of implementing nominal ideas is not enough; we further require a *moral* victory that the overhead be acceptable for practical systems. Apart from this being a literate Agda file, our results have been mechanised and are publicly accessible: `https://omelkonian.github.io/nominal-agda/`.

Nominal setup We conduct our development under some abstract type of **atoms**, satisfying certain constraints, namely decidable equality and being infinitely enumerable.[1]. We model this in Agda using *module parameters*, which could be instantiated with a concrete type:

```
module _ (Atom : Type) {{ _ : DecEq Atom }} {{ _ : Enumerable∞ Atom }} where
  И : (Atom → Type) → Type
  И φ = ∃ λ (xs : List Atom) → (∀ y → y ∉ xs → φ y)
```

The И **quantifier** enforces that a predicate holds for all but finitely many atoms, and **swapping** of two atoms can be performed on any type, subject to some laws:

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

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

```
record SwapLaws : Type where
  field 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
```

We only need to provide instances for the base case of *atoms* (whence the decidable equality), and *abstractions* (coming up next). From this we can systematically derive swapping definitions for all user-defined types, using a compile-time macro/tactic (c.f. the case study later on).

One particularly useful family of axioms in equivariant ZFA foundations [5] is that swapping distributes everywhere (constructors, functions, type formers) with the special case for swapping itself being swap-swap. It is consistent to axiomatize this generalized notion of distributivity for swap and we do so by means of a tactic that realises this *axiom scheme*. Most of the time

[1]...also known as "unfiniteness" in a recent nominal mechanization of the locally nameless approach [10].

we are working with types that have **finite support**, expressed using the 'new' quantifier: И² λ 𝕒 𝕓 → swap 𝕓 𝕒 $x \equiv x$. We can then define **equivariant** elements that admit the empty support, as well as an operation to generate fresh atoms freshAtom : $A \to Atom$ (whence the module requirement that atoms are infinitely enumerable). Agda is constructive, so freshAtom is constructive too, which is different from how fresh atoms are used in (non-constructive) set theories. An **abstraction** is just a pair of an atom and an element:

```
Abs A = Atom × A

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

```
instance
  ↔Abs : Swap (Abs A)
  ↔Abs .swap 𝕒 𝕓 (𝕔 , x) = (swap 𝕒 𝕓 𝕔 , swap 𝕒 𝕓 x)
```

Note that we can also provide a *correct-by-construction* and *total* concretion function. In nominal techniques based on Fraenkel-Mostowski set theory [8] this is impossible, and it seems to be a novel observation that in a constructive setup a total concretion function is fine.

Case study Once equipped with all expected nominal facilities, in particular *atoms* and *atom abstractions*, it is easy to define terms in **untyped λ-calculus** without mentioning de Bruijn indices or anything of that sort. For the sake of ergonomics and efficient theorem proving, we provide a meta-programming macro — based on *elaborator reflection* [2] — that is able to automatically derive the implementation of swapping of any type based on its structure.

```
data Term : Type where
  `_    : Atom → Term
  _·_   : Term → Term → Term
  λ_    : Abs Term → Term
unquoteDecl ↔Term =
  DERIVE Swap [ quote Term , ↔Term ]
```

```
data _≈_ : Term → Term → Type where
  ν≈ : ` x ≈ ` x
  ξ≈ : L ≈ L' → M ≈ M' → L · M ≈ L' · M'
  ζ≈ : И (λ 𝕩 → conc f 𝕩 ≈ conc g 𝕩) → λ f ≈ λ g
```

We can naturally express α-equivalence of λ-terms using the И quantifier and manually prove the aforementioned swapping laws and the fact that every λ-term has finite support. However, these all admit a systematic datatype-generic construction and we are currently in the process of automating them. The rest of the development remains identical to the mechanization presented in the PLFA textbook [14], particularly the 'Untyped' chapter. Meanwhile, the gnarly 'Substitution' appendix involving tedious index manipulations is now replaced by the usual nominal presentation of substitution, alongside a few general lemmas about equivariance and support:

```
_[_:=_] : 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 f)    [ 𝕒 := N ] = λ z ⇒ conc f z [ 𝕒 := N ] where z = freshAtom (𝕒 ∷ supp f ++ supp N)
```

We still have a few remaining lemmas to prove to fully cover the PLFA chapter on untyped λ-calculus, but we do not see any inherent obstacles to completing the confluence proof. A good next step would be to formalise a proof of *cut elimination* for first-order logic, since this involves name-abstraction on both terms and proof-trees.

Related work There have been previous nominal mechanizations in Agda that focus on the concrete instance of the untyped λ-calculus and include a proof of confluence [4, 3]. Ours closely matches the non-mechanized formulation in [7], which the Haskell `nom` package [6] then implements. Another representation of nominal sets in Agda [1] is preliminary and we would hope that our approach is more ergonomic and more amenable to scaling up. We treat our Agda library as a complement to other nominal implementations (in FreshML [12], Isabelle/HOL [13], and Nuprl [11]) that is ergonomic, lightweight, accessible, and illustrates the practical compatibility of nominal techniques within a constructive type system.

References

[1] Pritam Choudhury. Constructive representation of nominal sets in Agda. Master's thesis, Robinson College, University of Cambridge, 2015.

[2] David R. Christiansen and Edwin C. Brady. Elaborator reflection: extending Idris in Idris. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 284–297. ACM, 2016.

[3] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. Machine-checked proof of the Church-Rosser theorem for the lambda calculus using the Barendregt variable convention in constructive type theory. In Sandra Alves and Renata Wasserman, editors, *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*, volume 338 of *Electronic Notes in Theoretical Computer Science*, pages 79–95. Elsevier, 2017.

[4] Ernesto Copello, Alvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández. Alpha-structural induction and recursion for the lambda calculus in constructive type theory. In Mario R. F. Benevides and René Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2015, Natal, Brazil, August 31 - September 1, 2015*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 109–124. Elsevier, 2015.

[5] Murdoch J. Gabbay. Equivariant ZFA and the foundations of nominal techniques. *J. Log. Comput.*, 30(2):525–548, 2020.

[6] Murdoch J. Gabbay. The `nom` haskell package, 2020. URL: https://hackage.haskell.org/package/nom.

[7] Murdoch J. Gabbay and Aad Mathijssen. A nominal axiomatization of the lambda calculus. *Journal of Logic and Computation*, 20(2):501–531, 2010.

[8] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002.

[9] Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

[10] Andrew M. Pitts. Locally nameless sets. *Proc. ACM Program. Lang.*, 7(POPL):488–514, 2023.

[11] Vincent Rahli and Mark Bickford. A nominal exploration of intuitionism. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 130–141. ACM, 2016.

[12] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003.

[13] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, 2008.

[14] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.