Nominal Sets in Agda
A Fresh and Immature Mechanization

Miguel Pagano*
FAMAF - Universidad Nacional de Córdoba
Córdoba, Argentina
miguel.pagano@unc.edu.ar

José E. Solsona
Facultad de Ingeniería - Universidad ORT Uruguay
Montevideo, Uruguay
solsona@ort.edu.uy

In this paper we present our current development on a new formalization of nominal sets in Agda. Our first motivation in having another formalization was to understand better nominal sets and to have a playground for testing type systems based on nominal logic. Not surprisingly, we have independently built up the same hierarchy of types leading to nominal sets. We diverge from other formalizations in how to conceive finite permutations: in our formalization a finite permutation is a permutation (i.e. a bijection) whose domain is finite. Finite permutations have different representations, for instance as compositions of transpositions (the predominant in other formalizations) or compositions of disjoint cycles. We prove that these representations are equivalent and use them to normalize (up to composition order of independent transpositions) compositions of transpositions.

1 Introduction

Nominal sets were introduced to Computer Science by Gabbay and Pitts to give an adequate mathematical universe that permits the definition of inductive sets with binding [8]. Instead of taking equivalence classes of inductively defined sets (as in a formal treatment of, say, the Lambda Calculus) or a particular representation of the variables (as in the de Bruijn approach to Lambda Calculus), nominal sets have a notion of name abstraction that ensures all the properties expected for binders; in particular, alpha-equivalent lambda terms are represented by the same element of the nominal set of lambda terms.

In this paper we present a new mechanization [10] of nominal sets. Most of the current mechanizations of nominal sets represent finite permutations as compositions of transpositions, where transpositions are represented by pairs of atoms and compositions as lists. In contrast, our starting point is permutations (i.e. bijective functions); finite permutations are permutations that can be represented by composition of transpositions. Moreover they conflate the set of atoms mentioned in a list with the domain of the (represented) permutation. Pondering about this issue, we decided to develop a "normalization" procedure for representations of finite permutations; in order to prove its correctness, we were driven to introduce a cycle notation.

The rest of this paper is structured into four sections. In Sect. 2 we summarize the fundamentals of Nominal Sets; in Sect. 3 we explain the different representations of finite permutations and their equivalence; then, in Sect. 4 we present the most salient aspects of our mechanization in Agda; and finally in Sect. 5 we conclude by mentioning related works and contrasting them with our approach, indicating also our next steps. We assume some knowledge of Agda, but also hope that the paper can be followed by someone familiar with any other language based on type theory.

*Most of this work was done in a research leave in ORT Uruguay, financed by Agencia Nacional de Investigación e Innovación (ANII) of Uruguay.

P. Fontaine, D. Nantes-Sobrinho (Eds.):
Logical and Semantic Frameworks with Applications 2022 (LSFA'22)
EPTCS 376, 2023, pp. 67–80, doi:10.4204/EPTCS.376.7

2 Fundamentals of Nominal Sets

In this section we summarize the main concepts underlying the notion of Nominal Sets; for a more complete treatment we refer the reader to [13]. We repeat the basic definitions of group and group action. A *group* is a set G with a distinguished element ($\varepsilon \in G$, the *unit*), a binary operation ($\_\cdot\_ \colon G \times G \to G$, the *multiplication*), and a unary operation ($\_^{-1} \colon G \to G$, the *inverse*), satisfying the following axioms:

$$\begin{array}{rrll} \text{Associativity:} & g_1 \cdot (g_2 \cdot g_3) & = (g_1 \cdot g_2) \cdot g_3 & ,\forall g_1, g_2, g_3 \in G \\ \text{Inverse element:} & g \cdot (g^{-1}) & = \varepsilon = g^{-1} \cdot g & ,\forall g \in G \\ \text{Identity element:} & \varepsilon \cdot g & = g = g \cdot \varepsilon & ,\forall g \in G \end{array}$$

Although a group is given by the tuple $(G, \varepsilon, \_\cdot\_, \_^{-1})$ (and the proofs that these operations satisfy the axioms) we will refer to the group simply by G. A sub-group of G is a subset $H \subseteq G$ such that $\varepsilon \in H$ and H is closed under the inverse and multiplication.

Let G be a group. A *G-set* is a set X with an operation $\_\bullet\_ \colon G \times X \to X$ (called the *action*) satisfying:

$$\begin{array}{rrll} \text{Identity:} & \varepsilon \bullet x & = x & ,\forall x \in X \\ \text{Compatibility:} & g_1 \bullet (g_2 \bullet x) & = (g_1 \cdot g_2) \bullet x & ,\forall g_1, g_2 \in G, \forall x \in X \end{array}$$

A morphism between G-sets X and Y is a function $F : X \to Y$ that commutes with the actions:

$$F\,(g \bullet x) \;=\; g \bullet F\,x \qquad ,\forall g \in G, \forall x \in X$$

These are called *equivariant* functions. Since id_X is equivariant and the composition of equivariant functions yields an equivariant function we can talk of the category of G-Sets.

Any set X can be seen as a G-set by letting $g \bullet x = x$; such a G-set is called the *discrete* G-set. Moreover any group acts on itself by the multiplication.

One can form the (in)finitary product of G-sets by defining the action of G on a tuple in a pointwise manner:

$$g \bullet \langle x_1, x_2 \rangle \;=\; \langle g \bullet x_1, g \bullet x_2 \rangle \qquad ,\forall g \in G, \forall x_1 \in X_1, \forall x_2 \in X_2$$

The projections and the product morphism $\langle F, H \rangle$ are equivariant, assuming that F and H are also equivariant. G-set, as a category, also has co-products.

If X and Y are G-sets one can endow the set Y^X of functions from X to Y with the *conjugate* action:

$$(g \bullet F)\,x \;=\; g \bullet (F\,(g^{-1} \bullet x)) \qquad ,\forall g \in G, \forall x \in X \;\; .$$

G-sets over the Permutation Group The group of symmetries over a set X consists of $G = Sym(X)$, where $Sym(X)$ is the set of bijections on X; the multiplication of $Sym(X)$ is composition, the inverse is the inverse bijection, and the unit is the identity.

Let $Perm(X)$ be the subset of $Sym(X)$ of bijections that changes only finitely many elements; i.e., $f \in Perm(X)$ if $\mathrm{supp}(f) = \{x \in X \mid f\,x \neq x\}$ is finite. It is straightforward to prove that $Perm(X)$ is a sub-group of $Sym(X)$. Of course, if X is finite, then $Perm(X) = Sym(X)$. Notice that X itself is a $Perm(X)$-set with the action being function application: $\pi \bullet x \;=\; \pi\,x$.

In particular, the *transposition* (or *swapping*) of a pair of elements $x, y \in X$ is the finite permutation $(x\,y) \in Perm(X)$ given by

$$(x\,y)\,z \;=\; \begin{cases} y & \text{if } z = x \\ x & \text{if } z = y \\ z & \text{otherwise} \end{cases}$$

A basic result (in [9] is proved as Theorem 6.3 and Corollary 6.5) is that every $\pi \in Perm(X)$ can be expressed as a composition of *disjoint cycles*

$$\pi \;=\; (x_1\; x_2\; \ldots\; x_n) \circ \cdots \circ (z_1\; z_2\; \ldots\; z_k)$$

and every cycle can be expressed as a composition of transpositions

$$(x_1\; x_2\; \ldots\; x_n) \;=\; (x_1\; x_2) \circ (x_2\; x_3) \circ \cdots \circ (x_{n-1}\; x_n)$$

Therefore every $\pi \in Perm(X)$ can be expressed as a composition of transpositions. We elaborate on the equivalence of the representations in Sect. 3. Let us exhibit this with a concrete example.

Example 1. *Let $f\colon \mathbb{N} \to \mathbb{N}$ be defined as*

$$f\,x \;=\; \begin{cases} (x+2)\ mod\ 6 & \text{if } x \leq 5 \\ x & \text{else} \end{cases}$$

Function f is a finite permutation, because it has finite support: $\{x \in \mathbb{N} \mid 0 \leq x \leq 5\}$. Therefore it can be expressed as the composition of two cycles: $(1\ 3\ 5) \circ (0\ 2\ 4)$, or alternatively, it can also be expressed as a composition of four transpositions: $(1\ 3) \circ (3\ 5) \circ (0\ 2) \circ (2\ 4)$.

Nominal Sets If we let X be the set of variables for the lambda calculus, then a permutation on X is a renaming; such a permutation can be lifted to an action over the set of lambda terms (taking care of the bound variables). In the nominal parlance one says that X is the set of *atoms* or that variables are atomic names: an atomic name has no structure in itself. We only assume that a set of atoms is a countable infinite set with decidable equality; from now on we will use $\mathbb{A}$ to refer to a set of atoms.

Let X be a $Perm(\mathbb{A})$-set. We say that $x \in X$ *is supported by* $A \subseteq \mathbb{A}$ if

$$\forall \pi.\ (\forall a \in A.\ \pi\, a = a) \implies \pi \bullet x = x\ .$$

We say that X is a *nominal set* if each element of X is supported by some finite subset of $\mathbb{A}$. Since each finite permutation can be decomposed as a composition of transpositions, then one can prove that the above definition is equivalent to

$$\forall a, a' \in \mathbb{A} \setminus A.\ (a\ a') \bullet x = x\ .$$

The following are some examples of nominal sets:

- The discrete $Perm(\mathbb{A})$-set X is nominal, because any $x \in X$ is supported by $\emptyset$.
- $\mathbb{A}$ itself is nominal once equipped with the action $\pi \bullet a = \pi\, a$, because any $a \in \mathbb{A}$ is supported by $\{a\}$. More in general, any $S \subseteq \mathbb{A}$ containing name a is a support for a.
- The set $\lambda Term$ of λ-calculus terms, inductively defined by $t ::= V(a) \mid A(t,t) \mid L(a,t)$ where $a \in \mathbb{A}$, equipped with the action $\_ \bullet \_ : Perm(\mathbb{A}) \times \lambda Term \to \lambda Term$ such that

$$\begin{aligned} \pi \bullet V(a) &= V(\pi\, a) \\ \pi \bullet A(t_1,t_2) &= A(\pi \bullet t_1, \pi \bullet t_2) \\ \pi \bullet L(a,t) &= L(\pi\, a, \pi \bullet t) \end{aligned}$$

  is nominal because any $t \in \lambda Term$ is supported by $supp(t) = FreeVars(t)$.

In his book [13] Pitts uses classical logic to prove that if x is supported by some finite set A, then there exists a least supporting set, called *the* support of x. As shown by Swan [14] one cannot define the least support in a constructive setting; therefore a formalization in a constructive type theory should ask for "some" finite support. This affects the notion of freshness: in classical logic we have

$$x \text{ is fresh for } y \;\Leftrightarrow\; \mathrm{supp}(x) \cap \mathrm{supp}(y) = \emptyset,$$

with $x \in X$ and $y \in Y$ being elements of different nominal sets; but in a constructive setting one has to limit this relation to atoms, that is

$$a \in \mathbb{A} \text{ is fresh for } x \in X \;\Leftrightarrow\; a \notin \mathrm{supp}(x),$$

where $\mathrm{supp}(x)$ is the set supporting x, not necessarily the least one. Notice that the definition is the same ("there exists some finite support for each element"), but in classical logic that is sufficient to obtain the least support.

3 Finite Permutations

As we have already said, a finite permutation on a set A can be explicitly given by:

1. a bijection $f : A \to A$ together with its support $\mathrm{supp}(f) \subseteq_{fin} A$; i.e., $a \in \mathrm{supp}(f)$ if and only if $f\,a \neq a$;
2. a composition of disjoint cycles; concretely, we can think of this as a finite set $R \subseteq_{fin} A^*$ of disjoint cycles, each of them without repeated elements;
3. a composition of transpositions; that is, a finite sequence of pairs $p : (A \times A)^*$.

We present our proof that these definitions are equivalent. It basically boils down to define a predicate on sequences of elements in A not containing repeated elements ensuring that they are cycles for f. We use the usual notation $(a\ b)$ to denote the bijection $\{(a,b),(b,a)\}$.

Definition 1 (List of transpositions from a cycle)**.** *We define* $toFP\colon A \times A^* \to (A \times A)^*$.

$$toFP(a,\rho) = \begin{cases} [] & \text{if } \rho = [] \\ (a,b) : toFP(b,\rho') & \text{if } \rho = b : \rho' \end{cases}$$

If we know that $\rho = a : \rho'$*, then we also write* $toFP(\rho)$ *to mean* $toFP(a,\rho')$.

Definition 2 (Permutation from a list of transpositions)**.** *Let* $as : (A \times A)^*$*, then* $[\![as]\!] : A \to A$ *is defined by recursion on as:*

$$[\![as]\!] = \begin{cases} id & \text{if } as = [] \\ (a\ b) \cdot [\![as']\!] & \text{if } as = (a,b) : as' \end{cases}$$

Definition 3 (Prefixes)**.** *We say that a non-empty sequence* $\rho = [a_1,\ldots,a_n] : A^*$ *is a* prefix with head a_0 for *bijection* f *if:*

1. $a_0 \in \mathrm{supp}(f)$,
2. $f\,a_i = a_{i+1}$, *and*
3. $a_0 \notin \rho$.

A prefix ρ is closed *if $f\, a_n = a_0$. Since ρ is non-empty, we denote with $last(\rho)$ its last element.*

From this simple definition we can deduce:

Lemma 1 (Properties of prefixes)**.** *Let ρ be a prefix with head a.*

1. *If ρ' is a prefix with head $last(\rho)$, then its concatenation $\rho\rho'$ is a prefix with head a.*
2. *ρ has no duplicates.*
3. *If ρ is closed and $b \in (a : \rho)$, then $f\, b = [\![toFP(a,\rho)]\!]\, b$.*
4. *If $b \notin (a : \rho)$, then $[\![toFP(a,\rho)]\!]\, b = b$.*

We can extend this definition to a sequence of sequences: let $R = [(a_1,\rho_1),\ldots,(a_m,\rho_m)] : (A \times A^*)^*$, then R is a list of prefixes, with its head, if each ρ_i is a prefix and $\rho_i \cap \rho_j = \emptyset$.

Lemma 2 (Correctness of prefixes)**.** *Let $R = [(a_1,\rho_1),\ldots,(a_m,\rho_m)]$ be a list of closed prefixes, then $[\![toFP(a_1,\rho_1)\ldots toFP(a_m,\rho_m)]\!]\, a = f\, a$.*

This proves that from a representation with cycles one can get a representation with transpositions. If we can produce a list of closed prefixes from a finite permutation (as a bijection with its support explicitly given) then we have the equivalence. First we define a function $cycle_f \colon \mathbb{N} \times A \to A^*$ such that $cycle_f(n,a)$ computes a prefix with head a of length at most $n+1$ by recursion on n:

$$cycle_f(0,a) = [f\, a]$$

$$cycle_f(n+1,a) = \begin{cases} \rho & \text{if } f\, b = a \\ \rho\,[f\, b] & \text{otherwise} \end{cases}$$

$$\text{where } \rho = cycle_f(n,a) \text{ and } b = last(\rho)$$

We can extend this definition to compute a list of prefixes from a list of atoms:

$$cycles_f(n,[],R) = R$$

$$cycles_f(n,a:as,R) = \begin{cases} cycles_f(n,as,R) & \text{if } a \in \bigcup R \\ cycles_f(n,as,\rho : R) & \text{otherwise} \end{cases}$$

$$\text{where } \rho = a : cycle_f(n,a)$$

Lemma 3 (Correctness of computed cycles)**.** *If $f \colon A \to A$ is a bijection and $a \in \mathrm{supp}(f)$, then $cycle_f(n,a)$ is a prefix with head a, for all $n \in \mathbb{N}$. Moreover if $|\mathrm{supp}(f)| \leqslant n$, then $cycle_f(n,a)$ is closed.*

If R is a list of prefixes and $as \subseteq \mathrm{supp}(f)$, then $cycles_f(n,as,R)$ is a list of prefixes; if $|\mathrm{supp}(f)| \leqslant n$, then $cycles_f(n,as,R)$ is a list of closed prefixes.

Theorem 1. *If $f \colon A \to A$ is a bijection, then $R = cycles_f(|\mathrm{supp}(f)|, \mathrm{supp}(f), [])$ is a list of closed prefixes. Therefore $[\![toFP^*(R)]\!]\, a = f\, a$, for all $a \in A$.*

Notice that a composition of transpositions might mention elements that are not in the support of the induced permutation; for example, both $(1\ 1)$ and $(1\ 2)(2\ 1)$ are equal to the identity permutation. One can get a "normalized" representation by composing our functions. As a matter of fact, this was our motivation to formalize cycles.

Corollary 1 (Normalization of transpositions)**.** *Let p be a list of transpositions and $ats = \mathrm{supp}(toFP(p))$. Moreover, let $R = cycles_{toFP(p)}(|ats|, ats, [])$. Then $[\![toFP^*(R)]\!] = [\![as]\!]$; moreover every atom in $toFP^*(R)$ is in its support.*

4 Our Formalization in Agda

Our formalization is developed on top of the Agda's standard library v1.7 [15]. Figure 1 shows a high level view of the project. The standard library includes an algebraic hierarchy going beyond groups; it lacks, however, a formalization of group actions. The module `GroupAction` includes G-Sets, equivariant functions and constructions like products and co-products. We also have a `Permutation` module which includes the concepts of finite permutations, cycles, normalization and the permutation group. And last, in the module `Nominal` we formalize the concepts of support, nominal set, equivalence between different notions of support, normalization and again constructions like products and co-products.

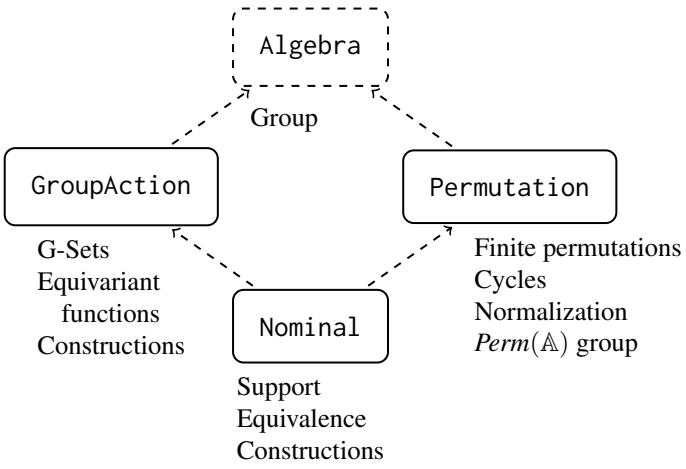

Figure 1: High level view of the modular organization in the project.

We first present the definition of Group in the standard library in order to introduce some terminology and concepts:

```
record Group c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier : Set c
    _≈_     : Rel Carrier ℓ
    _·_     : Op₂ Carrier
    ε       : Carrier
    _⁻¹     : Op₁ Carrier
    isGroup : IsGroup _≈_ _·_ ε _⁻¹
```

A Group is a *bundle* where the components of its definition (the carrier set, the unit, the inverse, the composition) are explicitly mentioned plus a proof, given by *isGroup*, that they satisfy the axioms. Notice that one of the fields is a relation `_≈_`; that relation should be an equivalence relation over the carrier: essentially this amounts to say that the `Carrier` has a setoid structure. Setoids allows for greater flexibility as they enable to work with a notion of equality that is not the propositional equality; `Func X Y` is the set of functions between setoids `X` and `Y` that preserve the equality; sometimes these functios are called *respectful*.

G-Sets Our first definition is the *structure* that collects the equations required for an action. In the following, we are under a module parameterized by `G : Group`.

```
record IsAction (F : Func (G.setoid ×ₛ X) X) : Set _ where
  _•_ : Carrier G → Carrier X → Carrier X
  g • x = Func.f F (g , x)
  field
    idₐ : ∀ x → ε ·ₐ x ≈X x
    compₐ : ∀ g' g x → g' • g • x ≈X (g' · g) • x
```

Notice that the record-type `IsAction` is a predicate over respectful functions from the setoid $G \times X$ to X. The definition of G-Set is straightforward and follows the pattern of the standard library

```
record G-Set : Set _ where
  field
    set : Setoid ℓ₁ ℓ₂
    action : Func (G.setoid ×ₛ set) set
    isAction : IsAction action
```

In order to introduce the notion of equivariant function[1] we first introduce the predicate `IsEquivariant` stating when `H : Func X Y` is equivariant for respectful functions `FX` and `FY`.

```
IsEquivariant :
 {X : Setoid ℓ₁ ℓ₂} →
 {Y : Setoid ℓ₃ ℓ₄} →
 (FX : Func (G.setoid ×ₛ X) X) →
 (FY : Func (G.setoid ×ₛ Y) Y) →
 (H : Func X Y) → Set (ℓ₁ ⊔ ℓ₄ ⊔ cℓ)
IsEquivariant {Y = Y} FX FY H = ∀ x g → F.f (g •X x) ≈Y (g •Y F.f x)
  where _•X_ = _•_ {F = FX} ; _•Y_ = _•_ {F = FY} ; _≈Y_ = _≈_ Y
        open module F = Func H
```

Now we pack a respectful function between the setoids of G-Sets together with a proof of it being equivariant.

```
record Equivariant (X : G-Set) (Y : G-Set) : Set _ where
  field
    F : Func (set X) (set Y)
    isEquivariant : IsEquivariant (action X) (action Y) F
```

In the following snippet we show how to construct binary products of G-Sets (we use copatterns to define record objects).

```
variable X Y : G-Set G
private
  open module GX = G-Set X ; open module GY = G-Set Y
G-Set-× : G-Set G
set G-Set-× = GX.set ×ₛ GY.set
f (action G-Set-×) (g , (x , y)) = g GX.• x  , g GY.• y
cong (action G-Set-×) (g=g' , (x=x' , y=y')) =
  Func.cong GX.action (g=g' , x=x') ,  Func.cong GY.action (g=g' , y=y')
idₐ (isAction (G-Set-×)) (x , y) = GX.idₐ x , GY.idₐ y
compₐ (isAction (G-Set-×)) g g' (x , y) = GX.compₐ g g' x , GY.compₐ g g' y
```

[1] We note that both Choudhury and Paranhos define equivariant functions only for the group of finite permutations.

We now prove that the first projection is equivariant; notice that `G-Set-×` is the product of `X` and `Y` introduced with the **`variable`** keyword.

```
π₁ : Equivariant G G-Set-× X
f (F π₁) = proj₁
cong (F π₁) = proj₁
isEquivariant π₁ _ _ = refl (set X)
```

Permutations Now we focus on the module `Permutation`. We start by introducing the group $Sym(\mathbb{A})$ using the definitions of inverses from the standard library; notice that the equivalence relation is given by the point-wise (or extensional) equality of functions.

```
-- In this context A-setoid is a Setoid (not necessarily decidable).
A = Carrier A-setoid ; _≈A_ = _≈_ A-setoid
Perm = Inverse A-setoid A-setoid
_≈ₚ_ : Rel Perm _
F ≈ₚ G = (a : A) → f F a ≈A f G a

Sym : Group (ℓ ⊔ ℓ') (ℓ ⊔ ℓ')
Carrier Sym = Perm
_≈_ Sym = _≈ₚ_
_·_ Sym = _∘ₚ_          -- composition of Perm, from the stdlib
ε Sym = idₚ A-setoid    -- identity Perm, from the stdlib
_⁻¹ Sym = _⁻¹           -- inverse permutation, from the stdlib
isGroup Sym = record { ... } -- ommited
```

If we ask the setoid `A-setoid` to be decidable, then we can define the swapping permutation.

```
module Perm (A-setoid : DecSetoid ℓ ℓ') where
  open DecSetoid A-setoid renaming (Carrier to A)
  transp : A → A → A → A
  transp a b c with does (c ≟ a)
  ... | true = b
  ... | false with does (c ≟ b)
  ... | true = a
  ... | false = c

  transp-perm : (a b : A) → Perm
  transp-perm a b = record {
    f = transp a b ; f⁻¹ = transp a b
    ; cong₁ = transp-respects-≈ a b ; cong₂ = transp-respects-≈ a b
    ; inverse = transp-involutive a b , transp-involutive a b
  }
```

Our next goal is to define the group $Perm(\mathbb{A})$ of finite permutations of atoms. As we explained before, finite permutation can be given by a bijective map, as a composition of transpositions, or as a composition of disjoint cycles.

In other works the group of finite permutations is explicitly defined as lists of pairs, where each pair represents a transposition and the empty list is the identity permutation: appending a pair (a,b) to a list

p amounts to compose the transposition $(a\ b)$ to the permutation denoted by p. Concatenation of lists p and p' also induces their composition. This choice has the advantage of being explicit and avoids having alternative expressions for composing permutations. On the other hand it still allows different representatives for the same permutation; in fact, $[(a,a)]$, $[(b,a),(a,b)]$, and $[]$ are all representations of *the* identity permutation. It is clear that the setoid of finite permutations should equate those three versions of the identity, therefore the equivalence relation used is that of inducing the same permutation.

We started with the following syntactic representation of Finite Permutations, which is close to that of lists but in terms of S-expressions; since we cannot ensure canonicity with lists, why not to be more liberal also on associativity?

```
data FinPerm : Set ℓ where
  Id : FinPerm
  Swap : (a b : A) → FinPerm
  Comp : (p q : FinPerm) → FinPerm
```

The permutation associated with a `FinPerm` is given by

```
⟦_⟧ : FinPerm → Perm
⟦ Id ⟧ = id_p setoid
⟦ Swap a b ⟧ = transp-perm a b
⟦ Comp p q ⟧ = ⟦ q ⟧ ∘_p ⟦ p ⟧
```

Before introducing our concrete formalization of $Perm(\mathbb{A})$ let us exploit the fact that we have a decidable setoid of atoms to prove that the equivalence of finite permutation is also decidable. In order to do that, we define a relation `_⊆ₛ_` on `FinPerm`; `p ⊆ₛ q` holds when `q` coincides with `p` in the support of the latter. Since we can compute the support of `FinPerm`s and the equality of atoms is decidable, then we can decide `_⊆ₛ_`.

```
_⊆ₛ_ : Rel FinPerm (ℓ ⊔ ℓ')
p ⊆ₛ q = All (λ a → f ⟦ p ⟧ a ≈ f ⟦ q ⟧ a) (support p)

?⊆ₛ : ∀ p q → Dec (p ⊆ₛ q)
?⊆ₛ p q = all? (λ a → f ⟦ p ⟧ a ≟ f ⟦ q ⟧ a) (support p)
```

Moreover we can prove that the mutual containment is equivalent to denoting the same permutation; thus we can decide the equality of finite permutations as given by `FinPerm`:

```
_≈ₛ_ : Rel FinPerm (ℓ ⊔ ℓ')
p ≈ₛ q = p ⊆ₛ q × q ⊆ₛ p

≈ₛ-dec : ∀ p q → Dec (p ≈ₛ q)
≈ₛ-dec p q = (?⊆ₛ p q) ×-dec (?⊆ₛ q p)
-- We omit the proofs of these lemmas.
≈ₛ⇒≈_p : ∀ p q → p ≈ₛ q → ⟦ p ⟧ ≈_p ⟦ q ⟧
≈_p⇒≈ₛ : ∀ p q → ⟦ p ⟧ ≈_p ⟦ q ⟧ → p ⊆ₛ q
_≟_p_ : ∀ p q → Dec (⟦ p ⟧ ≈_p ⟦ q ⟧)
```

Furthermore we can normalize a `FinPerm` to have an equivalent permutation where every occuring atom is in its support.

Let us first revisit the Example 1 now in Agda where we see how to encode a finite permutation as a composition of cycles.

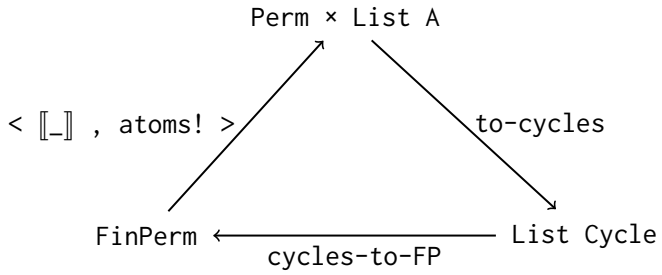

Figure 2: The mappings between different representations of permutations.

```
f : ℕ → ℕ
f x with x ⩽? 5
... | yes p = (x + 2) mod 6
... | no ¬p = x
```

We represent cycles simply as lists of atoms; we certainly could also have used fresh lists to represent cycles. A composition of cycles is a list of cycles.

```
Cycle = List A

cycle₀ cycle₁ : Cycle
cycle₀ = 1 :: 3 :: 5 :: []
cycle₁ = 0 :: 2 :: 4 :: []
f-cycles : List Cycle
f-cycles = cycle₀ :: cycle₁ :: []
```

Or alternatively, it can also be expressed as a composition of four transpositions:

```
f-swaps : FinPerm
f-swaps = Comp (Comp (Swap 1 3) (Swap 3 5)) (Comp (Swap 0 2) (Swap 2 4))
```

In Figure 2 we show the three representations of finite permutations. The normalization of `FinPerm` is simply the composition of the mappings:

```
norm : FinPerm → FinPerm
norm = cycles-to-FP ∘ cycles-from-FP
```

The functions `cycles-to-FP` maps lists of disjoint cycles to `FinPerm` and `cycles-from-FP` goes in the reverse direction, producing a list of disjoint cycles from a `FinPerm` (this is the composition of the diagonal arrows in Fig. 2).

The correctness of the normalization follows the proof presented in Sec. 3. Although we do not enforce neither freshness for cycles nor disjointness of cycles we keep that as an invariant when we compute the cycles in `to-cycles`.

```
module Thm (p : FinPerm) where
  ats = atoms! p -- Fresh list of the atoms in the support of p.
  -- from-atom-˜* is the proof of Lemma 3.
  rel = from-atoms-˜* ⟦ p ⟧ ats []* (fp-supp p) (dom⊇atoms! p)
  -- the representation as composition of cycles
  ρs = to-cycles ⟦ p ⟧ (length ats) ats []
```

```
  -- This property follows from Lemma 3.
  ∈-dom⇒∈ρs : (_∈-dom ⟦ p ⟧) ⊆ (_∈ concat ρs)

  norm-corr : ⟦ p ⟧ ≈ₚ ⟦ norm p ⟧
  norm-corr x with x ∈? concat ρs
  ... | yes x∈at = ~*-out ⟦ p ⟧ rel x∈at      -- Item 3 of Lemma 1.
  ... | no x∉at = trans                        -- f ⟦ p ⟧ x = x = f ⟦ norm p ⟧ x
      (¬∈-dom⇒∉-dom {⟦ p ⟧} (contraposition ∈-dom⇒∈ρs x∉at))
      (~*-out-fresh ⟦ p ⟧ rel x∉at)           -- Item 4 of Lemma 1.
```

We also have other correctness result to prove that the `FinPerm` obtained from a `Perm` and its support is equivalent to it:

```
module Thm' (F : Perm) {ats : List A} (is-sup : ats is-supp-of F)
  (incl : (_∈ ats) ⊆ (_∈-dom F)) where

  ρs = to-cycles p (length ats) ats []
  norm-corr : F ≈ₚ ⟦ cycles-to-FP ρs ⟧
```

Let us remark that `FinPerm` is just a representation and the set of finite permutation, `PERM`, is the subset of `Perm` corresponding to the image of $[\![\_]\!]$:

```
PERM : Set _
PERM = Σ[ p ∈ Perm ] (Σ[ q ∈ FinPerm ] (p ≈ₚ ⟦ q ⟧))
```

A disadvantage of using this encoding is that we need to deal with triples; for instance, the identity `PERM` is represented by `Id`.

```
ID : PERM
ID = idₚ setoid , Id , λ _ → refl
```

The group $Perm(\mathbb{A})$ is explicity defined as:

```
Perm-A : Group (ℓ ⊔ ℓ') (ℓ ⊔ ℓ')
Carrier Perm-A = PERM
_≈G_ Perm-A = _≈ₚ_ on proj₁
_·_ Perm-A = _○P_
ε Perm-A = ID
_⁻¹ Perm-A = _⁻¹P
isGroup Perm-A = record { ... }
```

We alleviate the burden of working with triples by proving lemmas characterizing the action of `PERM`s in terms of the finite permutation, for instance for `Id`:

```
-- In this context the group acting on G-Sets is Perm-A.
module Act-Lemmas {X-set : G-Set {ℓ₁ = ℓx} {ℓ₂ = ℓx'}} where
  _≈X_ = Setoid._≈_ set
  id-act : ∀ (π : PERM) (x : X) → proj₁ π ≈ₚ ⟦ Id ⟧ → (π • x) ≈X x
  id-act π x eq = trans (cong¹ {π} {ID} x eq) (idₐ x)
```

Nominal Sets Remember that a subset $A \subseteq \mathbb{A}$ is a support for x if every permutation fixing every element of A fixes x, through the action. A subset of a setoid `A` can be defined either as a predicate or as pairs (just as in `PERM` where the predicate is `λ p → Σ[ q ∈ FinPerm ] (p ≈ₚ ⟦ q ⟧)`) or as another type, say `B`, together with an injection `ι : Injection B A`.

```
variable
  X : G-Set
  P : SetoidPredicate A-setoid
is-supp : Pred X _
is-supp x = (π : PERM) → (predicate P ⊆ _∉-dom (proj₁ π)) → (π • x) ≈X x
```

The predicate `λ a → f (proj₁ π) a ≈A a` is `_∉-dom (proj₁ π)`; therefore, if `P a` iff $a \in A$, then `predicate P ⊆ _∉-dom (proj₁ π)` is a correct formalization of $\forall a \in A.\ \pi\, a = a$.

Our official definition of support is the following:

```
_supports_ : Pred X _
_supports_ x = ∀ {a b} → a #ₛ P → b #ₛ P → SWAP a b • x ≈X x
```

Here `SWAP` is a `PERMutation` equal to $[\![$`Swap a b`$]\!]$. We formally proved that both definitions are equivalent, which is stated by the mutual implications:

```
is-supp⊆supports : ∀ x → is-supp x → _supports_ x
supports⊆is-supp : _supports_ ⊆ is-supp
```

Let us note that the second implication uses explicitly the normalization of finite permutations and its correctness.

In order to define nominal sets we need to choose how to say that a subset is finite; as explained by Coquand and Spiwak [7] there are several possibilities for this. We choose the easiest one: a predicate is finite if there is a list that enumerates all the elements satisfying the predicate.

```
finite : Pred (SetoidPredicate setoid) _
finite P = Σ[ as ∈ List Carrier ] (predicate P ⊆ (_∈ as))
```

A G-Set is nominal if all the elements of the underlying set are finitely supported.

```
record Nominal (X : G-Set) : Set _ where
  field
    sup : ∀ x → Σ[ P ∈ SetoidPredicate setoid ] (finite P × P supports x)
```

It is easy to prove that various constructions are nominals; for instance any discrete G-Set is nominal because every element is supported by the empty predicate $\bot_s$:

```
Δ-nominal : (S : Setoid _ _) → Nominal (Δ S)
sup (Δ-nominal S) x = ⊥ₛ , ⊥-finite , (λ _ _ → S-refl {x = x})
  where open Setoid S renaming (refl to S-refl)
```

We have defined `G-Set-⇒ X Y` corresponding to the G-Set of equivariant functions from `X` to `Y`; now we can prove that `G-Set-⇒ X Y` is nominal, again with $\bot_s$ as the support for any `F : Equivariant X Y`.

```
→-nominal : Nominal (G-Set-⇒ X Y)
sup (→-nominal) F = ⊥ₛ , ⊥-finite , λ _ _ → supported
  where supported : ∀ {a b} x → f ((SWAP a b) ·→ F) x ≈Y f F x
```

5 Conclusion

Nominal techniques have been adopted in various developments. We distinguish developments borrowing some concepts from nominal techniques to be applied in specific use cases (e.g. formalization of languages with binders like the λ or π calculus with their associated meta-theory) [2, 6, 5, 4] from more

general developments aiming to formalize at least the core aspects of the theory of nominal sets. We are more concerned with the later type.

The nominal datatype package for Isabelle/HOL [16] developed by Urban and Berghofer implements an infrastructure for defining languages involving binders and for reasoning conveniently about alpha-equivalence classes. This Isabelle/HOL package inspired Aydemir et al. [1] to develop a proof of concept for the Coq proof assistant, however it had no further development. In his Master thesis [3], Choudhury notes that none of the previous developments following the theory of nominal sets were based on constructive foundations. He showed that a considerable portion (most of the first four chapters of Pitts book [13]) of the theory of nominal sets can also be developed constructively by giving a formalization in Agda. Pitts original work is based on classical logic, and depends heavily on the existence of the smallest finite support for an element of a nominal set. However, Swan [14] has shown that in general this existence cannot be constructively guaranteed, as it would imply the law of the excluded middle.

Choudhury works with the notion of *some non-unique support*. In order to formalize the category of Nominal Sets, Choudhury preferred setoids instead of postulating functional extensionality. As far as we know, Choudhury is still the most comprehensive mechanization in terms of instances of constructions having a nominal structure.

Recently Paranhos and Ventura [11] presented a constructive formalization in Coq of the core notions of nominal sets: support, freshness and name abstraction. They follow closely Choudhury's work in Agda [3], acknowledging the importance of working with setoids. They claim that by using Coq's type class and setoid rewriting mechanism, much shorter and simpler proofs are achieved, circumventing the "setoid hell" described by Choudhury. In his master thesis [12] Paranhos further developed the library.

Both of those two formalizations in type theory take a very pragmatic approach to finite permutations: a finite permutation is a list of pairs of names. In our approach, we start with the more general notion of bijective function from which the finite permutations are obtained as a special case; moreover having different representations allowed us to state and prove some theorems that cannot even be stated in the other formalizations. So far, our main contributions are: the representation of finite permutations and the normalization of composition of transpositions; the equivalence between two definitions of the relation "A supports the element x"; and proving that the extension of every container type can be enriched with a group action (notice that this cover lists, trees, etc.).

Our next steps are the definition of freshness. We are studying an alternative notion of support that would admit having a freshness relation between elements of two nominal sets (in contrast with other mechanization that only consider "the atom a is fresh for x") and name abstraction. In parallel we hope to be able to prove that extensions of finite containers on nominal sets are also nominal sets. We also hope to streamline further some rough corners of our development.

Acknowledgments

This formalization grew up from discussions with the group of the research project "Type-checking for a Nominal Type Theory": Maribel Fernández, Nora Szasz, Álvaro Tasistro, and Sebastián Urciouli. We thank Cristian Vay for discussions about group theory. This work was partially funded by Agencia Nacional de Investigación e Innovación (ANII) of Uruguay.

References

[1] Brian E. Aydemir, Aaron Bohannon & Stephanie Weirich (2007): *Nominal Reasoning Techniques in Coq: (Extended Abstract)*. *Electron. Notes Theor. Comput. Sci.* 174(5), pp. 69–77,

doi:10.1016/j.entcs.2007.01.028.

[2] Jesper Bengtson & Joachim Parrow (2009): *Formalising the pi-calculus using nominal logic*. *Log. Methods Comput. Sci.* 5(2), doi:10.2168/lmcs-5(2:16)2009. Available at `http://arxiv.org/abs/0809.3960`.

[3] Pritam Choudhury (2015): *Constructive Representation of Nominal Sets in Agda*. Master's thesis, Cambridge University.

[4] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2018): *Formalisation in Constructive Type Theory of Barendregt's Variable Convention for Generic Structures with Binders*. *Electronic Proceedings in Theoretical Computer Science* 274, doi:10.4204/EPTCS.274.2.

[5] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2018): *Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory*. *Electronic Notes in Theoretical Computer Science* 338, pp. 79–95, doi:10.1016/j.entcs.2018.10.006.

[6] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove & Maribel Fernández (2016): *Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory*. *Electronic Notes in Theoretical Computer Science* 323, pp. 109–124, doi:10.1016/j.entcs.2016.06.008. Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015).

[7] Thierry Coquand & Arnaud Spiwack (2010): *Contribuciones científicas en honor de Mirian Andrés Gómez*, chapter Constructively finite?, pp. 217–230.

[8] M. J. Gabbay & A. M. Pitts (2002): *A New Approach to Abstract Syntax with Variable Binding*. *Formal Aspects of Computing* 13, pp. 341–363, doi:10.1007/s001650200016.

[9] Thomas William. Hungerford (1974): *Algebra*. Graduate texts in mathematics, Springer, New York, doi:10.1007/978-1-4612-6101-8.

[10] Miguel Pagano & José E. Solsona (2022): *Nominal Sets in Agda*. `https://github.com/miguelpagano/nominal-sets/`.

[11] Fabrício S. Paranhos & Daniel Ventura: *Towards a Formalization of Nominal Sets in Coq*. `https://popl22.sigplan.org/details/CoqPL-2022-papers/4/Towards-a-Formalization-of-Nominal-Sets-in-Coq`. Online; accessed 1 May 2022.

[12] Fabrício Sanches Paranhos (2022): *Uma Formalização da Teoria Nominal em Coq*. Master thesis, Universidade Federal de Goiás (UFG), Brasil. Available at `http://repositorio.bc.ufg.br/tede/handle/tede/12314`.

[13] Andrew M. Pitts (2013): *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, England, doi:10.1017/CBO9781139084673.

[14] Andrew Swan (2017): *Some Brouwerian Counterexamples Regarding Nominal Sets in Constructive Set Theory*, doi:10.48550/ARXIV.1702.01556.

[15] The Agda Team (2021): *The Agda standard library, version 1.7*. `https://github.com/agda/agda-stdlib`.

[16] Christian Urban & Stefan Berghofer (2006): *A Recursion Combinator for Nominal Datatypes Implemented in Isabelle/HOL*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, *Lecture Notes in Computer Science* 4130, Springer, pp. 498–512, doi:10.1007/11814771_41.