University of Cambridge

MPhil Thesis

Constructive Representation of Nominal Sets in Agda

Author:
Pritam Choudhury
Robinson College

Supervisor:
Prof. Andrew M. Pitts
The Computer Laboratory

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

June 12, 2015

Declaration of Authorship

I, Pritam Choudhury of Robinson College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 13758 (excluding all mathematical symbols)

Signed: ____________________________________

Date: ____________________________________

All trademarks used in this dissertation are hereby acknowledged.

Abstract

The theory of nominal sets provide a mathematical analysis of names that is based upon symmetry. It formalizes the informal reasoning we employ while working with languages involving name binding operators. The central ideas of the theory are support, freshness and name abstraction, which respectively encapsulate the ideas of name dependence, name independence and alpha equivalence. This theory has been developed within the framework of classical logic. Certain notions of this classical theory, like the smallest finite support for an element of a nominal set, are non-constructive. In this dissertation, we show that with appropriate modifications, a considerable portion of the theory of nominal sets can also be developed constructively. We show this by building from scratch, the theory of nominal sets, upto and including name abstraction sets, in the dependent type-theoretic environment of the programming language Agda. In our development, we replace the notion of unique smallest finite support with that of non-unique some finite support. In addition, all throughout our development, we work with setoids. This helps us in recovering extensionality of functions and in working with alpha-equivalence classes of terms. Though extensionality of functions could have been easily postulated in Agda, we refrained from using any postulate whatsoever at any level, since we cannot ensure constructivity of the whole development in case it rests even on just a single postulate. Our work can be extended and developed further to produce a nominal version of Agda which would make working with binding constructs in Agda much more easier.

"A journey of a thousand miles must begin with a single step."

Lao Tzu

Events, as they turned out

After completing my Bachelor's degree in electrical engineering from Indian Institute of Technology, Roorkee, India, when I came to Cambridge, I was supposed to study natural language processing. But after attending classes for a week, I realized, natural language processing was not my cup of tea. The one thing that I could not digest all my life is ambiguity and handling ambiguity is one of the very important concerns of natural language processing. So, I decided to switch over to the theoretical modules, in retrospect, with incommensurate trust on my mathematical abilities. It was not long before I found out that I literally knew nothing. The veracity of this statement can be judged from the fact that at that time, I did not know what a lambda term is. But I did not lose hope. Because though I could hardly follow the lectures technically, I got the intuitions. The Category theory course, by Prof. Andrew M. Pitts, in particular, opened unexplored territories in my mind. I liked all the 'general abstract non-sense'. So then, when I approached Prof. Pitts to suggest some mathematical project for my MPhil ACS, he proposed that I work on 'Constructive representation of nominal sets in Agda'. At that time, I neither knew what 'constructive' is, nor did I know what nominal sets are. Not to say anything about my knowledge of Agda, since I did not know any programming language other than C++ and MATLAB. But I was determined to work on the project. I argued with myself, this is the time to take risks, if not now, then never. When I wrote the project proposal with the help of Prof. Pitts in late November last year, I was hardly aware of the intensity of the challenges that lay ahead of me. The first blow came when I started reading some Agda code developed by others. I was totally confused. What is a type? What is the difference between a data type and a function type? Why do propositions on a type map the elements of that type to 'Set'? Prof. Pitts answered these questions, but at that time, a new term was leading me to a whole bunch of other even newer terms and I could see no end to this. The other problem was that information about Agda is scattered across various tutorials, papers, code snippets, etc. There is no single comprehensive guide to Agda covering the language in all its details. So I was reading everything I could find, but all this reading did not lead to an understanding of the language. With all these problems, two months passed. It was mid February and I was almost at the point where I began, other than having known a few technical terms here and there. It was clear that things were not working. At this point, I changed all my plans. May be the top-down approach would not work. May be I cannot possibly comprehend the profundity of dependent type theory, constructive logic and Agda in a month or two and that any attempt towards that would just increase confusion. But may be, I can still start working on Agda and Agda itself will teach me how to deal with it. So then, I tried the bottom up approach. Slowly, but surely, it worked. Starting from the point where writing a line or two in Agda required hours of

thought amidst constant tussle with the type-checker to make it accept the definition I was writing, I gradually moved to the point where I could just sit and start writing as if I were doing maths on pen and paper, without really bothering about the type-checker, out of confidence that if I can satisfy myself, the type-checker too will be satisfied. And from that point onwards, things flowed, though not without occasional hiccups, but those were petty things when compared with the challenges I had to overcome in the initial days of the project.

This project then, was a medium of transformation for me, not just academically but also from a broader perspective. I learnt how to face my fears, I learnt why not to fight with them, but to swim along with them, for it's fear that leads to all great truths, when followed incessantly. I believe this lesson will help me face my future in a more resourceful way.

And none of these would have been possible without Prof. Pitts. All throughout these months, he has been an academic mentor and a moral supporter. Though I could not gather everything that he had said in my initial supervisions, still I believe that those meetings were very helpful, because they gave me the faith that may be things are not that difficult, may be, like Andrew, I too can understand them some day. My thanks are also due to Dr. Jeremy Yallop and Dr. Leo White, lecturers of the Advanced Functional Programming course. Being completely new to functional programming, I could not have worked with Agda, if this course were not there. And I cannot thank Jeremy and Leo enough for how well they taught the course. My thanks also to other lecturers and professors and other members of the Computer Laboratory, who have helped in different ways at different times.

Contents

Declaration of Authorship **ii**

Abstract **iv**

Events, as they turned out **vi**

Contents **ix**

Symbols **xi**

1 Introduction **1**
- 1.1 What's in a name? 1
- 1.2 Names and Symmetry 2
- 1.3 Names in computer science 2
- 1.4 Literature review 3
- 1.5 Motivation 4
- 1.6 Related Work 6
- 1.7 Constructive logic 6
- 1.8 Intuitionistic or Dependent type theory 7
- 1.9 Agda 8
- 1.10 An example 10
- 1.11 Structure of the thesis 14

2 Permutations **16**
- 2.1 Atoms 16
- 2.2 Finite permutations 17
- 2.3 Permutations are bijective functions 17
- 2.4 Equivalence relation on permutations 20
- 2.5 Group 20
- 2.6 Finite permutations form a group 21
- 2.7 Group Action 23
- 2.8 Examples of Group Action 23
- 2.9 Perm-Set 26
- 2.10 Equivariant functions 26
- 2.11 Example of an equivariant function 27
- 2.12 Useful properties of permutations 27

2.12.1 Atoms unchanged under action 27
2.12.2 A permutation and a name-swap 28

3 A category theoretic perspective **32**
3.1 Nominal Set 32
3.2 Examples of Nominal Sets 34
3.3 Functions between nominal sets 36
3.4 **Nom** and $\mathbf{Nom_{fs}}$ 38
3.5 Products in **Nom** and $\mathbf{Nom_{fs}}$ 40
3.6 Coproducts in **Nom** and $\mathbf{Nom_{fs}}$ 40
3.7 Exponentials in **Nom** and $\mathbf{Nom_{fs}}$ 41
3.7.1 Exponential object 42
3.7.2 Application morphism 43
3.7.3 Currying in **Nom** 43
3.7.4 Currying in $\mathbf{Nom_{fs}}$ 45
3.8 Pullbacks in **Nom** 45
3.9 Nom-sets and Perm-sets 47
3.10 Powerset 49

4 Freshness **51**
4.1 Definition 51
4.2 Some/any theorem 52
4.3 Freshness in finitely supported functions 53
4.4 Separated product functor 54

5 Name Abstraction **57**
5.1 Alpha-equivalence 57
5.2 Name abstraction sets 59
5.3 Name abstraction functor 60
5.4 Properties of name abstraction functor 61
5.5 Freshness condition for binders 67

6 Conclusion **73**

Bibliography **75**

Symbols

$\mathbb{A}$	The set of atomic names
Perm	The set of all finite permutations on $\mathbb{A}$
Nom	The category of nominal sets and equivariant functions
$\mathbf{Nom_{fs}}$	The category of nominal sets and finitely supported functions
$S_X(x)$	Some finite support for $x \in X$
$a\#x$	Atom a is fresh for x
$x \approx_\alpha y$	x is alpha-equivalent to y
$[\mathbb{A}]X$	The set $\mathbb{A} \times X$ quotiented by $\approx_\alpha$
$(a \quad b) \cdot x$	Action of swapping names a and b on x
$\pi \cdot x$	Permutation π acts on x
$x \approx y$	x is related to y by an equivalence relation $\approx$
$l_1 + l_2$	Concatenation of lists l_1 and l_2

Chapter 1

Introduction

1.1 What's in a name?

Imagine a Primordial Observer (PO), an entity which considers everything to be atomic and thus unrelated to everything else. So, to PO, a certain table at a certain time instant has no relation whatsoever to the same table at the very next moment. Indeed, PO has no sense of meaning because any meaning attributed to any object would be restricted to that very object at that very instant and thus would be of no use. But if PO decides that certain instances of a given object can be unified under some property, thus yielding a single object whose different instances are related to one another by that property, then PO must coin a name for that object. Thus, any name represents a level of abstraction relating different instances of the thing named. This is true even in the case of algebraic variables used as names, since, for example, while using 'x' as a name for the number of apples in a given problem, we implicitly apply the convention that all occurrences of the variable 'x' in that problem refer to the number of apples. Here the variable 'x' by itself is of no particular interest, since we could have used any other variable 'y' in lieu of 'x', but the important thing is the unification of all the occurrences of the variable. Now, if in some other problem, we have 'x' and 'y' as the number of apples and oranges respectively, then unlike the previous case, we cannot unilaterally substitute 'y' for 'x', since then we lose the distinction between the number of apples and oranges. Put symbolically, $\lambda x.x \approx_\alpha \lambda y.y$, but $\lambda xy.xy \not\approx_\alpha \lambda yy.yy$. Thus, names help us in unifying certain instances and in distinguishing the unified ones from other such instances.

1.2 Names and Symmetry

Symmetry helps us in abstract characterization of an object, since by giving us the actions performed on the object under which it remains invariant, it reveals the different unified instances of the object, thus giving us an idea of the abstract property which unifies those instances and which truly gives birth to the object. The actions should be such that they keep the properties of the whole ensemble of objects unchanged. Now all such actions under which a given object belonging to the ensemble remains invariant can be thought of as a characterization of that object. This idea can help us in characterizing names as well. For this, let us consider a countably infinite set of names and actions given by finite name-swappings. Any finite name-swapping leaves the set of names, considered as a whole, unchanged. Now, the set of finite name-swappings under which a given name remains unchanged may be said to characterize that name. This logic can be extended from a single name to sets of names to structures made up of names. For example, under any finite name-swapping (say, ns), $ns(\lambda x.x) \approx_\alpha \lambda x.x$ and $ns(\lambda xy.xy) \approx_\alpha \lambda xy.xy$, but this is not so for $\lambda x.xy$, since under the name-swapping $(x\ \ y)$, we have $(x\ \ y)\cdot(\lambda x.xy) = \lambda y.yx \not\approx_\alpha \lambda x.xy$. Thus, while all finite name-swappings leave $\lambda x.x$ and $\lambda xy.xy$ unchanged, there are certain finite name-swappings which change $\lambda x.xy$. What then, would be the best way to characterize the symmetries of $\lambda x.xy$, i.e. how can we describe the finite name-swappings which leave $\lambda x.xy$ unchanged? As we can see, any name-swapping that leaves y unchanged also leaves $\lambda x.xy$ unchanged. So, we can describe the symmetries of a term made up of names in terms of a set of names, each of which if and when remains unchanged by a finite name-swapping, the term also remains unchanged. So the term may be thought of as depending on the identities of the names in this set. This idea also brings forth a complementary notion of non-dependence. For example, the term $\lambda x.x$ does not depend on any name. This then gives us an alternative characterization of the symmetries of a term, since any finite name-swapping that only swaps names on which a term does not depend, also leaves the term unchanged.

1.3 Names in computer science

A formal language is specified by a set of symbols, a grammar stating the ways of valid combination of the symbols, certain axioms specifying relations between the valid terms of the language and inference rules for carrying out deductions. If the grammar of a formal language has variable binding operators, like 'λ' in untyped-lambda calculus, then in general, we work in such a language with alpha-equivalence classes of terms and not the raw-terms as such. Alpha-equivalence classes are classes of terms which differ only in the

names of the bound variables, like $\lambda x.x$ and $\lambda y.y$. Any member of an alpha-equivalence class may be taken as a representative of the class, provided we change over to another suitable member of the class in case there is conflict during substitution of free variables. For example, if we have to substitute y with x in the lambda-term $\lambda x.xyw$, we can rename the bound variable x to anything other than x and the free variables in $\lambda x.xyw$ viz. y and w. Say, we rename x as v to obtain $\lambda v.vyw \approx_\alpha \lambda x.xyw$. Now, we can simply substitute x for y, i.e. $[x/y](\lambda x.xyw) = \lambda v.vxw$. But we can make things a bit easier by letting the name-swapping $(x\ \ y)$ instead of $[x/y]$ act on $\lambda x.xyw$. This automatically respects alpha-equivalence since $(x\ \ y) \cdot (\lambda x.xyw) = \lambda y.yxw \approx_\alpha [x/y](\lambda x.xyw)$. Name swappings then, give us an alternative way of dealing with substitution and alpha-equivalence. Though the technicalities of capture-avoiding substitution are at times relegated to the periphery with the umbrella statement that we identify terms upto alpha-equivalence, a formal treatment of this informal approach is necessary to avoid any unwanted error while defining functions on alpha-equivalence classes of terms by structural recursion and proving properties of those functions by structural induction.

1.4 Literature review

Induction and recursion are powerful tools by which complex systems can be built out of very basic ingredients. Given a few basic terms and some rules for generation of new terms from the previous ones, we can build an infinite number of terms and prove properties of all those terms by structural induction and recursion. Burstall (1969) [1] applies the idea of structural induction to computer programs, deriving proofs which are 'very similar to the programs to which they refer' by 'cutting down some of the trivial but tedious manipulation'. This is made possible by working on abstract syntax trees (ASTs) of programs rather than on their concrete syntax. Such syntax trees, also known as first order syntax trees, do away with all the details unnecessary for the semantics of the program like brackets, punctuation, etc and simply shows the dependence structure of the constituents of the program on one another. These ASTs then represent the 'meaning' of a program and we would therefore expect that any two programs having the same 'meaning' should have the same AST. But this is not true of first order abstract syntax trees of languages involving variable binders where two alpha-equivalent programs, i.e. programs differing only in the names of bound variables, may be represented by different syntax trees. One way of overcoming this problem may be by replacing bound variables with indexes, similar to de Bruijn (1972) [2]. But one then needs to operate on de Bruijn indexes and the flexibility and intuition of working with variables is lost. Another way of overcoming the problem is through Higher Order Abstract Syntax (HOAS), developed by Pfenning and Elliot (1988) [3].

In HOAS, the bound variables of object programs are represented by meta-variables in a metalanguage so that ultimately any two alpha-equivalent programs have the same representation in the metalanguage. But as Honsell, Miculan and Scagnetto (1998) [4] observes, delegating properties like substitution, alpha-conversion and freshness of names to the metalanguage makes proving properties, involving these mechanisms, in the object language impossible and thus such properties need to be postulated, so that some of the details that HOAS delegates to the meta-level are reified at the object level. Yet another approach towards solving the original problem is through permutation model of set theory with atoms, originally proposed by Fraenkel and later used by Mostowski, and in the current context, rediscovered by Gabbay and Pitts (2002) [5] to give a semantics of abstract syntax modulo alpha-equivalence. Sets in Fraenkel-Mostowski universe are called nominal sets. The main advantage of working with nominal sets as opposed to HOAS is that the former gives the user 'the ability to manipulate names of bound variables explicitly in computation and proof' [5]. The key ideas of nominal sets are support, freshness and name abstraction. Support captures the idea of name dependence while freshness the idea of name independence, as discussed in the previous section. Name abstraction models what it means for a variable to be bound in a term. But even out of these three ideas, support occupies the most important position since without finite support, there are no nominal sets. The theory of nominal sets as presented by Pitts (2013) [6] relies heavily on the existence of a least finite support. But Swan (2014) [7] has shown that constructively, least finite support 'cannot be guaranteed to exist'. This is so because the existence of least finite support entails the assumption of the Weak Limited Principle of Omniscience (WLPO), which says that, given $f : \mathbb{N} \to \mathbb{B}$, where $\mathbb{N}$ is the set of natural numbers and $\mathbb{B} = \{0, 1\}$ is the set of Booleans, the statement $f(n) = 0$, for all n, is decidable in nature. WLPO is a weaker notion of the law of the excluded middle, which states that any proposition is either true or false. WLPO and the law of the excluded middle, in general, do not hold constructively. This thesis tries to formalize the theory of nominal sets, upto and including most of the first four chapters of Pitts (2013) [6] in the constructive type-theoretic environment of the dependently-typed programming language Agda.

1.5 Motivation

Human reasoning is subject to human frailties. As we design more and more complex systems, it becomes a necessary evil for us to forget the intricate low-level details so that we might implement the system without getting overwhelmed, but as the saying goes, the devil often lies in the detail. This has been proved time and again, with the loss of NASA's $125M Mars Climate Orbiter in 1999 due to 'the failure to use metric units in

the coding of a ground software file' (Page 16, [8]) or with the North-east blackout of 2003 partially caused due to a race condition in the control software [9]. This shows that mere testing is not good enough to ensure robustness of critical systems. As Dijkstra (1997) [10] puts it, 'program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence'. Thus, to ensure correctness, we need formal proofs. But owing to the complexity of the systems involved, such proofs cannot be developed manually. This is where proof assistants come in. Proof assistants can not only be used for the verification of hardware systems but also for the verification of software systems. The proof assistant acts like a meta-level language where we can express and prove properties of object-level languages, which may well be a mathematical theory or a hardware circuit description. If a certain property of an object-level language is verified by a proof assistant, we can say that the property holds in the given language, provided the proof assistant works with a sound underlying logic. Thus, the correctness of all statements verified by a proof assistant boils down to the correctness of the proof assistant itself. In this way, we can be more confident of claims which are verified by reliable proof assistants. This is one of our motivations behind developing the theory of nominal sets constructively in Agda. Now, since proof assistants become the most vital part of the process of formal verification, we should try to improve them on all aspects, whether by making the underlying logic more and more robust or by providing end-level user-friendly features that automate the straightforward technical manipulations. As mentioned in the previous section, when working with languages involving binding operators, one generally works at the level of alpha abstraction, but currently in proof assistants like Coq, Agda, one cannot do that. Due to this, the workload of any development in a language involving binders is substantially increased, since properties of freshness, substitution and alpha-equivalence needs to be proved explicitly in each and every case. An idea of the increased workload may be obtained from the following statement by Hirschkoff (1997) [11], who formalized π-calculus theory in the calculus of constructions. 'Of our 800 proved lemmas, about 600 are concerned with operators on free names'. Since working with binding constructs in proof assistants presents such prominent challenges, finding representations that allow reasoning about inductively defined data structures (with binders) with minimum technical overhead was the central theme of the POPLMark challenges, a set of benchmark problems designed to test the strength of proof assistants in formalizing the principles of programming languages. But the proposed solutions to the problems leave a lot of scope for improvement, and as Aydemir *et al.* (2005) puts it, 'none (of the solutions) emerge as clear winners' (Section 2.3, [12]). Thus, the problem of representing binding constructs is far from being truly solved. In such a scenario, a nominal version of Agda, where one could work at the level of alpha abstraction, without worrying about the intricate technicalities of freshness, substitution, etc, would be very helpful. Exploring the possibility

of providing nominal techniques to an Agda user is one of the other motivations behind this project.

1.6 Related Work

Nominal techniques have been developed in some programming languages. Shinwell (2005) [13] added nominal features to the Objective Caml language to develop Fresh Objective Caml. Urban and Berghofer (2006) [14] implemented a structural recursion operator for nominal datatypes in Isabelle/HOL, this enables one to work conveniently at the level of alpha abstraction (in languages involving binders) in Isabelle/HOL. Aydemir *et al.* (2007) [15] developed nominal reasoning techniques in Coq using an axiomatized approach. But we are not aware of any previous work towards constructive development of the theory of nominal sets.

1.7 Constructive logic

The notions of truth and proof occupy a fundamental position in logic and by extension, in mathematics. But what constitutes a proof of a proposition is not easy to answer. In traditional or classical mathematics, proof is given by a sequence of steps, each supported directly or indirectly by axioms and rules of inference, leading to the conclusion beginning from the premises. To prove the validity of the process, one then just needs to show that the axioms and the rules of inference, considered as a whole, are logically consistent. Along with consistency, if one can ensure completeness, i.e. the truth value of all statements that can be formulated in the language concerned is decidable, one then has a 'solid foundation' for the language. Giving such a foundation to mathematics, considered as a formal language, by 'disposing of the foundational questions, once and for all' was the primary aim of Hilbert's program but Gödel's theorems showed that such a goal cannot be realized (Zach, 2006, [16]). Meanwhile, there were others, most notably, L. E. J. Brouwer, who believed that mathematics is an activity of the mind which rests upon intuition and as such, needs no foundation whatsoever [17]. According to Brouwer, every mathematical object is a mental construction and a proof is an intuitive justification for a proposition, given by a construction which realizes the proposition. As such, showing that the negation of a statement is false is not constructively equivalent to proving the statement. An interesting account of the differences between classical and constructive or intuitionistic mathematics can be found in Heyting (1956) [18]. The ideas of Brouwer were subsequently developed, among others, by Heyting, Kolmogorov, Bishop and Martin-Löf. This resulted in a full-fledged alternative form of mathematics

and logic, known as constructive mathematics and logic. One of the great advantages of constructive logic is that it lends itself easily to machine level formalizations. As Martin-Löf (1985) [19] puts it, 'If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute, then it no longer seems possible to distinguish the discipline of programming from constructive mathematics'. It is precisely because of this reason that Martin-Löf's intuitionistic theory of types (1971) [20], originally developed to formalize constructive mathematics, may well be viewed as a programming language.

1.8 Intuitionistic or Dependent type theory

Type theory was first introduced by Russell (1908) [21] to resolve the set-theoretic paradoxes of self-membership. Russell's theory had a ramified hierarchy of types, according to which every type is a term of some other type belonging to a higher level in the hierarchy, this hierarchical design circumvented the problem of self-membership since a type and the terms inhabiting the type now became members of different universes. Church (1940) [22] gave a formulation of the simple theory of types which incorporated certain features of his λ-calculus. The simple types were extended by Girard (1972) [23] to polymorphic types and then type operators were added to give System F and System Fω respectively. Within System F, we have terms that depend on types while System Fω makes possible the construction of types depending upon other types. Martin-Löf's dependent type theory (1971) [20] goes a step further by allowing types to depend upon terms. Thus, in Martin-Löf type theory, we can express propositions about terms by means of types very easily. Oury and Swierstra (2008) [24] show the power of dependent types by embedding three domain specific languages within the dependently typed programming language Agda.

The main idea behind Martin-Löf's dependent type theory is the Curry-Howard correspondence, whereby propositions are seen as types and proofs as terms inhabiting those types. Thus the judgement $a \in A$ can be read in any of the following ways:-

- a is a term of type A
- a is a proof of the proposition A
- a is a program for the specification A
- a is a solution to the problem A

This correspondence can be carried to the logical connectives $\wedge, \vee, \Rightarrow, \forall, \exists$, in the following way:-

Type	Proposition
$(\Pi x \in A)B(x)$	$(\forall x \in A)B(x)$
$(\Sigma x \in A)B(x)$	$(\exists x \in A)B(x)$
$(\Pi x \in A)B$	$A \Rightarrow B$
$(\Sigma x \in A)B$	$A \wedge B$
$A + B$	$A \vee B$

Other than these, this type theory also contains the empty type, the unit type, the natural number type and type universes.

As Nordström, Petersson and Smith (2000) [25] notes, an advantage of using type theory for program construction is that it is possible to express specifications, programs for those specifications and proofs that the programs indeed satisfy the specifications within the same formalism, along with the guarantee that the evaluation of a well-typed program always terminates. We can therefore use type theory as a programming logic. To assist in the formalization of proofs within type theory, several automated proof assistants like Coq, Agda, etc have been developed. Agda, in particular, is based on Martin-Löf type theory.

1.9 Agda

Agda was developed by Norell (2007) [26] with the aim of building 'a practical programming language based on type theory'. It is both a programming language and a proof assistant. Its important features are that it allows inductive and co-inductive data-type definitions, dependent functions, parametrised modules, dependent record types, pattern matching on intermediate results using the `with` construct, dot pattern, absurd pattern, implicit arguments, etc (Norell, 2009, [27]). We now give a few examples to give the reader a taste of Agda, in which all the proofs in this thesis have been developed.

One of the important data types used throughout is the identity type. For any $x : A$, we know that $x = x$, but that knowledge is propositional and not definitional. Hofmann (1995) [28] points out that propositional equality can be expressed as an equivalence relation between two terms of a type, whereby the relation is inhabited if and only if the two terms are identical. We give below the definition of propositional equality in Agda.

Definition 1.1

```
data _≡_ {a : Level}{A : Set a}(x : A) : A → Set a  where
 refl : x ≡ x

reflex : {A : Set} → (a : A) → a ≡ a
reflex a = refl

sym : {ℓ : Level}{A : Set ℓ} → {a b : A} → a ≡ b → b ≡ a
sym refl = refl

_≡<_>_ : {ℓ : Level} {A : Set ℓ} → (a : A) → { b c : A} → a ≡ b
                                                          → b ≡ c → a ≡ c
a ≡< refl > refl = refl
```

The equality relation then gives a setoid, a set equipped with an equivalence relation. Setoids are very useful since by using setoids, we can even recover several extensional features like function extensionality in the intensional type theoretic environment of Agda. This is one of our prime motivations in working with setoids rather than with sets, since we require function extensionality throughout this work. It is important to note that though Agda provides mechanisms for postulating axioms so that function extensionality could have been postulated, we refrained from using any postulate at any level whatsoever, since the purpose of this project is to constructively develop the theory of nominal sets from scratch and we cannot ensure constructivity of the whole development in case it rests even on just a single postulate.

Below we give definitions for some of the basic data types: the empty type, the unit type, the Boolean type, the natural number type, the list type, the vector type, the decidable equality type and a dependent record type.

Definition 1.2

```
data ⊥ : Set  where  --the empty type


record ⊤ : Set where  --the unit type
  constructor 𝟙


data 𝔹 : Set where   --boolean type
```

```
 true : 𝔹
 false : 𝔹


data ℕ : Set where   --natural number type
 zero : ℕ
 succ : ℕ → ℕ


data List (A : Set) : Set where  --list type
 [] : List A
 _::_ : A → List A → List A


data Vec (A : Set) : ℕ → Set where   --vector type
 [] : Vec A zero
 _::_ : ∀ {n} → (a : A) → (as : Vec A n) → Vec A (succ n)


data Dec {ℓ : Level} (X : Set ℓ) : Set ℓ where
                             --decidable equality type
  yes : X → Dec X
  no : (X → ⊥) → Dec X


record Σ {a b}(A : Set a)(B : A → Set b) : Set (a ⊔ b) where
                            --dependent record type
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

1.10 An example

In this section, we emphasize the difference between classical logic and constructive logic with the help of an example.

Lemma 1.3 Given an infinite sequence of distinct elements $\mathbb{A} = \{a_0, a_1, \ldots, a_n, \ldots\}$ with decidable equality and a finite set $S \subset \mathbb{A}$, there exists some $a \in \mathbb{A}$ such that $a \notin S$.

Proof. **A classical solution:-** If possible, let there be no such a such that $a \in \mathbb{A}$ and $a \notin S$. Hence, $a \in \mathbb{A} \Rightarrow a \in S$. So $\mathbb{A} \subseteq S$. But $S \subset \mathbb{A}$. Hence, $S = \mathbb{A}$. Therefore S must be infinite. But this is not true. So we have a contradiction and thus, our original assumption was false. The lemma follows. □

Proof. **A constructive solution:-** The classical solution uses the law of the excluded middle to give the proof. But this is not valid constructively. To show that the lemma holds, we need to construct an $a \in A$ such that $a \notin S$. But for this, we need to give precise definitions of $\mathbb{A}$, $\neq$ and $_ \notin _$.

Definition 1.4

```
data Atom : Set where
  root : Atom
  next : Atom → Atom

_≠_ : {A : Set}(x x’ : A) → Set
x ≠ x’ = (x ≡ x’ → ⊥)

data _∉_ (a : Atom) : List Atom → Set where
  a∉[] : a ∉ []
  a∉as : ∀ {b as} → a ∉ as →  ( b ≠ a ) → a ∉ (b :: as)
```

To find an $a \notin S$, we find an a which is greater than all atoms in S. For this, we need to define the greater than relation, both between two atoms and between an atom and a list.

Definition 1.5

```
data _<’_  : Atom → Atom → Set where
 r<n : ∀ {a} → root <’ next a
 n₁<n₂ : ∀ {a b} → a <’ b → next a <’ next b

data _<_ : List Atom → Atom → Set where
 []< : ∀ {a} → [] < a
 as< : ∀ {a b as} → as < b → a <’ b → (a :: as) < b
```

Now, given two atoms, we provide an atom which is greater than both and prove that it is so.

Definition 1.6

```
greater : Atom → Atom → Atom
greater root root = next (root)
greater (next a) root = next (next a)
greater root (next a) = next (next a)
greater (next a) (next b) = next(greater a b)
```

Lemma 1.7

```
b<nb : (b : Atom) → b <' next b
b<nb root = r<n
b<nb (next b) = n₁<n₂ (b<nb b)

a>bc⇒a>b : (b c : Atom) → b <' greater b c
a>bc⇒a>b root root = r<n
a>bc⇒a>b root (next c) = r<n
a>bc⇒a>b (next b) root  = b<nb (next b)
a>bc⇒a>b (next b) (next c)  = n₁<n₂ (a>bc⇒a>b b c)

a>bc⇒a>c : (b c : Atom) → c <' greater b c
a>bc⇒a>c root root = r<n
a>bc⇒a>c root (next c) = b<nb (next c)
a>bc⇒a>c (next b) root  = r<n
a>bc⇒a>c (next b) (next c)  = n₁<n₂ (a>bc⇒a>c b c)
```

Now, we find an atom that does not belong to S, represented here as a list.

Definition 1.8

```
outside : List Atom → Atom
outside [] = root
outside (a :: as) = greater a (outside as)
```

Now, we need to show that greater than is transitive and as such definition (1.8) gives something greater than S.

Lemma 1.9

```
a<'b<'c : (a b c : Atom) → a <' b → b <' c → a <' c
a<'b<'c _ _ root a<'b = λ ()
a<'b<'c _ root (next _) = λ ()
a<'b<'c root (next b) (next c) _ _ = r<n
a<'b<'c (next a) (next b) (next c) (n1<n2 a<'b) (n1<n2 b<'c) =
                                n1<n2 (a<'b<'c a b c a<'b b<'c)

as<a<'b : (a b : Atom) → (as : List Atom) → as < a → a <' b → as < b
as<a<'b a b [] as<a a<'b = []<
as<a<'b a b (x :: xs) (as< xs<a x<'a) a<'b = as<
           (as<a<'b a b xs xs<a a<'b)(a<'b<'c x a b x<'a a<'b)

outside< : (xs : List Atom) → xs < (outside xs)
outside< [] = []<
outside< (a :: as) = as< (as<a<'b (outside as)(greater a (outside as))
                          as (outside< as) (a>bc⇒a>c a (outside as)))
                                           (a>bc⇒a>b a (outside as))
```

And finally, we need to show that greater than is different from being equal.

Lemma 1.10

```
<≠ : {a b : Atom} → a <' b → a ≡ b → ⊥
<≠ r<n = λ ()
<≠ (n1<n2 a<'b) = a≠b⇒na≠nb (<≠ a<'b)

<∉ : ∀ {a as} → as < a → a ∉ as
<∉ []< = a∉[]
<∉ (as< as<b a<'b) = a∉as (<∉ as<b) (<≠ a<'b)

outside∉ : (xs : List Atom) → (outside xs) ∉ xs
outside∉ xs = <∉ (outside< xs)
```

The lemma follows. □

This example shows two things. First, the constructive solution to a problem is not always the same as the classical one. Secondly, in Agda, we need to work out everything in explicit detail so much so that even re-bracketing of terms needs a justification in terms of an explicit mention of the associativity of the operator concerned.

So in the following chapters, we skip some of the steps in our proofs for brevity, steps without which our Agda programs won't run but which otherwise are fairly clear to any reader. In case of any doubt, the reader may please refer to the program corresponding to the specification concerned.

1.11 Structure of the thesis

Chapter 2 discusses permutations, permutation actions, equivariant functions and properties of permutations. Chapter 3 gives a category theoretic treatment to nominal sets. Chapter 4 develops properties of the freshness relation. Chapter 5 introduces name abstraction sets and formalizes their properties. Chapter 6 concludes the dissertation.

Chapter 2

Permutations

One of the central ideas of the theory of nominal sets is that of finite permutations and permutation actions. Given any set X, finite or infinite, we can define a finite permutation P on X as a bijective function from X to X such that $X' = \{x \mid P(x) \neq x\} \subset X$ is finite. The set of all permutations, finite or infinite, on a set X form a group, known as the symmetric group on X. The set of finite permutations is a subgroup of the symmetric group. A finite permutation can also be represented by cycles, for example, for the set $X = \{1, 2, 3, 4\}$, the permutation $P(1) = 1, P(2) = 3, P(3) = 4, P(4) = 2$ can be represented as $(2 \;\; 3 \;\; 4)$, meaning $2 \to 3 \to 4 \to 2$. The representation by means of cycles is not unique, since we can also represent P as $(2 \;\; 3)(2 \;\; 4)$ meaning $2 \to 3, 3 \to 2 \to 4, 4 \to 2$, i.e. the action is given by the composition of actions acting from left to right. Now, if we wish to represent P by means of 2-cycles only, even then we cannot ensure a unique representation since P can also be represented as $(1 \;\; 4)(2 \;\; 3)(1 \;\; 4)(2 \;\; 4)$. We shall have to accommodate this fact while defining permutations. In case of nominal sets, finite permutations are nothing but bijective functions from a countably infinite set of atomic names or Atoms to itself.

2.1 Atoms

Atom ($\mathbb{A}$), as defined in definition (1.4), is a countably infinite set of atomic names with decidable equality, meaning given any two atoms $a, a' \in \mathbb{A}$, we can decide whether $a = a'$ or $a \neq a'$. An atom does not contain any element and it is not empty either.

2.2 Finite permutations

A finite permutation can be defined as a list of pairs of atoms, i.e. as a list of 2-cycles.

Definition 2.1

```
Perm = List (Atom × Atom)
```

Any pair of atoms in Perm represents a name swap. For any three atoms $a, b, c \in \mathbb{A}$, the action of name swap $(a \quad b)$ on c is given by:-

$$(a \quad b) \cdot c = \begin{cases} b & \text{if } c = a \\ a & \text{if } c = b \\ c & \text{if } c \neq a, c \neq b \end{cases} \tag{2.1}$$

The action of a permutation can then be given as:-

Definition 2.2

```
PermAct : Perm → Atom → Atom
PermAct [] a = a
PermAct (aa :: the_rest) a = PermAct the_rest (swap aa a)
```

2.3 Permutations are bijective functions

Our definition of a permutation as a list of pairs of atoms is equivalent to a bijective function from $\mathbb{A}$ to itself. We can prove this by showing that the action of Perm on $\mathbb{A}$ is both injective and surjective. First, we prove injectivity.

- Injectivity

  Lemma 2.3

```
a≠b⇒πa≠πb : (a b : Atom) → (p : Perm) → (a ≡ b → ⊥)
                                → ((PermAct p a ≡ PermAct p b) → ⊥)
```

Proof. The base case for empty list is trivially true. Let this be true for lists of length n and we show this to be true for lists of length $n+1$. Let $p = (m\ \ n) :: q$ be a list of length $n+1$. For the following cases, we get a contradiction: $(a = m) \ \wedge\ (b = m)$, $(a = n) \ \wedge\ (b = n)$, $(a = m) \ \wedge\ (b = n) \ \wedge\ (m = n)$, $(a = n) \ \wedge\ (b = m) \ \wedge\ (m = n)$.
Now we consider the other cases:

– $(a = m) \ \wedge\ (b = n) \ \wedge\ (m \neq n)$

$$
\begin{aligned}
&((m\ \ n) :: q) \cdot a \\
&= q \cdot n \quad \text{[By definition]} \\
&\neq q \cdot m \quad \text{[By induction hypothesis]} \\
&= ((m\ \ n) :: q) \cdot b \quad \text{[By definition]}
\end{aligned}
$$

Similarly, this holds when $(a = n) \ \wedge\ (b = m) \ \wedge\ (m \neq n)$.

– $a \notin \{m, n\} \wedge b \notin \{m, n\}$

$$
\begin{aligned}
&((m\ \ n) :: q) \cdot a \\
&= q \cdot a \quad \text{[By definition]} \\
&\neq q \cdot b \quad \text{[By induction hypothesis]} \\
&= ((m\ \ n) :: q) \cdot b \quad \text{[By definition]}
\end{aligned}
$$

□

- Surjectivity

Now, we prove that for any $p \in \text{Perm}$ and $a \in \mathbb{A}$, there exists some $b \in \mathbb{A}$ such that $p \cdot b = a$. To find such a b for any a and p, we define the inverse function.

Definition 2.4

```
invPerm : Perm → Perm
invPerm []  = []
invPerm ((a , b) :: the_rest) =  (invPerm (the_rest)) ++ [(a , b)]
-- _++_ is the list concatenation operator and [ x ] = x :: []
```

Before proceeding further, we need to prove two lemmas.

Lemma 2.5

```
p₁++p₂≡p₂p₁ : ∀ {a} → (p₁ p₂ : Perm) → PermAct (p₁ ++ p₂) a
                                     ≡ PermAct p₂ (PermAct p₁ a)
```

Proof. We prove this by induction on the length of p_1. The base case is trivially true. Let this be true for p_1 of length n. We show this to be true for p_1 of length $n+1$. Let $p_1 = (m \;\; n) :: q$.

$$
\begin{aligned}
&(((m \;\; n) :: q) + p_2) \cdot a \\
&= (q + p_2) \cdot ((m \;\; n) \cdot a) \quad \text{[By definition]} \\
&= p_2 \cdot q \cdot ((m \;\; n) \cdot a) \quad \text{[By induction hypothesis]} \\
&= p_2 \cdot (((m \;\; n) :: q) \cdot a) \quad \text{[By definition]}
\end{aligned}
$$

□

Lemma 2.6

```
p∘p⁻¹Act : (p : Perm) → ∀ {a} →  PermAct (p ++ (invPerm p)) a ≡ a
```

Proof. We prove this by induction on the length of p. The base case is trivially true. Let this be true for p of length n. We show this to be true for p of length $n+1$. Let $p = (m \;\; n) :: q$.

$$
\begin{aligned}
&(((m \;\; n) :: q) + ((m \;\; n) :: q)^{-1}) \cdot a \\
&= (q + (q^{-1} + (m \;\; n))) \cdot ((m \;\; n) \cdot a) \quad \text{[By definition]} \\
&= ((q + q^{-1}) + (m \;\; n)) \cdot ((m \;\; n) \cdot a) \quad \text{[By associativity]} \\
&= (m \;\; n) \cdot (q + q^{-1}) \cdot ((m \;\; n) \cdot a) \quad \text{[By lemma (2.5)]} \\
&= (m \;\; n) \cdot (m \;\; n) \cdot a \quad \text{[By induction hypothesis]} \\
&= a \quad \text{[By idempotence of swapping]}
\end{aligned}
$$

□

Now we can show that:

Lemma 2.7

```
pp⁻¹a≡a : (p : Perm) → (a : Atom) →
          (PermAct p (PermAct (invPerm p) a)) ≡ a
```

Proof.

$$
\begin{aligned}
& p \cdot p^{-1} \cdot a \\
&= (p^{-1} + p) \cdot a \quad \text{[By lemma (2.5)]} \\
&= (p^{-1} + (p^{-1})^{-1}) \cdot a \quad [\because (p^{-1})^{-1} = p] \\
&= a \quad \text{[By lemma (2.6)]}
\end{aligned}
$$

□

This proves that permutations represented as lists are equivalent to bijective functions.

2.4 Equivalence relation on permutations

To accommodate for the fact that two different lists may in fact represent the same permutation intensionally, we define a relation whereby any two permutations that are intensionally same are related to one another, whereas ones that are not are unrelated.

Definition 2.8

```
pEquiv : Perm → Perm → Set
pEquiv p₁ p₂ = ∀ {x} → (PermAct p₁ x) ≡ (PermAct p₂ x)
```

This relation, in fact, captures the idea of permutations as bijective functions. It is easy to check that pEquiv is an equivalence relation. The reflexive, symmetric and transitive properties follow from the reflexive, symmetric and transitive properties respectively of $_ \equiv _$. Hence, Perm along with this relation forms a setoid.

Now we show that Perm along with the composition operation forms a group.

2.5 Group

A group is a set G along with an identity element e and a binary operation $_ \bullet _$ and an unary operation $_^{-1}$ such that the following axioms are satisfied:

$$\forall g_1, g_2, g_3 \in G, \qquad g_1 \bullet (g_2 \bullet g_3) = (g_1 \bullet g_2) \bullet g_3 \tag{G1}$$
$$\forall g \in G, \qquad g \bullet g^{-1} = e \tag{G2}$$
$$\forall g \in G, \qquad g^{-1} \bullet g = e \tag{G3}$$

But since we are working with setoids instead of sets, we need to modify the equality relation in the above examples with an equivalence relation and we also need to add a few more axioms,

$$\forall g_1, g_2, g_3, g_4 \in G, \qquad g_1 \approx g_3 \wedge g_2 \approx g_4 \Rightarrow g_1 \bullet g_2 \approx g_3 \bullet g_4 \tag{G4}$$
$$\forall g_1, g_2 \in G, \qquad g_1 \approx g_2 \Rightarrow g_1^{-1} \approx g_2^{-1} \tag{G5}$$

Thus the record type Group is given as :-

Definition 2.9

```
record Group : Set₁ where
  field
    Gₛ       : Set                              --underlying set
    _≈ₛ_     : Rel Gₛ                           --relation
    eq≈ₛ     : isEquivalence Gₛ _≈ₛ_            --proof of equivalence
    _•_      : Op₂ Gₛ                           --binary operation
    e        : Gₛ                               --identity element
    _⁻¹      : Op₁ Gₛ                           --unary operation
    asso     : assoc {Gₛ}{_•_}{_≈ₛ_}            -- axiom (G1)
    id       : idax {Gₛ}{_•_}{_⁻¹}{_≈ₛ_} e      -- axioms (G2) & (G3)
    cong•    : _•_ Preserves₂ _≈ₛ_ ⟶ _≈ₛ_ ⟶ _≈ₛ_ -- axiom (G4)
    cong⁻¹   : _⁻¹ Preserves _≈ₛ_ ⟶ _≈ₛ_        -- axiom (G5)
```

2.6 Finite permutations form a group

Perm along with the composition operation forms a group.

Theorem 2.10

```
PermG : Group
```

Proof. That Perm obeys axioms (G2) and (G3) follow from lemmas (2.6) and (2.7) respectively. So we just need to verify axioms (G4) and (G5).

Lemma 2.11

```
cong++ : _++_ Preserves₂ pEquiv ⟶ pEquiv ⟶ pEquiv
```

Proof. For $p_1, p_2, q_1, q_2 \in$ Perm, $p_1 \approx p_2, q_1 \approx q_2$ and $x \in \mathbb{A}$, we have,

$$\begin{aligned}
&(p_1 + q_1) \cdot x \\
&= q_1 \cdot (p_1 \cdot x) \quad \text{[By lemma (2.5)]} \\
&= q_1 \cdot (p_2 \cdot x) \quad [\because p_1 \approx p_2] \\
&= q_2 \cdot (p_2 \cdot x) \quad [\because q_1 \approx q_2] \\
&= (p_2 + q_2) \cdot x \quad \text{[By lemma (2.5)]}
\end{aligned}$$

□

Lemma 2.12

```
p≈q⇒p⁻¹≈q⁻¹ : (p q : Perm) → (pEquiv p q) →
                   (pEquiv (invPerm p) (invPerm q))
```

Proof. For $x \in \mathbb{A}$, we have:

$$\begin{aligned}
&p^{-1} \cdot x \\
&= (q + q^{-1}) \cdot p^{-1} \cdot x \quad \text{[By lemma (2.6)]} \\
&= q^{-1} \cdot q \cdot p^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= q^{-1} \cdot p \cdot p^{-1} \cdot x \quad [\because p \approx q] \\
&= q^{-1} \cdot x \quad \text{[By lemma (2.7)]}
\end{aligned}$$

□

Hence Perm forms a group. □

Once we have proved that Perm is a group, we can now prove that the permutation action on atoms is in fact, a group action.

2.7 Group Action

An action of a group G on a set X is given by a function $_\cdot_ : G \times X \to X$ such that the following axioms are satisfied:

$$\forall g_1, g_2 \in G, x \in X, \qquad g_1 \cdot (g_2 \cdot x) = (g_1 \bullet g_2) \cdot x \tag{GA1}$$

$$\forall x \in X, \qquad e \cdot x = x \tag{GA2}$$

For setoids, we need to add an additional axiom:

$$\forall g_1, g_2 \in G, x_1, x_2 \in X, \quad g_1 \approx g_2 \quad \wedge x_1 \approx x_2 \Rightarrow g_1 \cdot x_1 \approx g_2 \cdot x_2 \tag{GA3}$$

This gives us the following record type for a G-setoid X:

Definition 2.13

```
record GAct (G : Group) : Set₁ where
 field
    AₛP : Set                                        --underlying set
    ≈ₐP : Rel AₛP                                    --relation on X
    eq≈ₐP : isEquivalence AₛP ≈ₐP                    --proof of equivalence
    ActP : (Gₛ G) → AₛP → AₛP                          --group action
    resP : ActP Preserves₂ (_≈ₛ_ G) ⟶ ≈ₐP ⟶ ≈ₐP  --Axiom GA3
    p₁p₂↠P : p₁p₂Act {Gₛ G}{AₛP}{_•_ G}{≈ₐP}{ActP} --Axiom GA1
    ι↠P : ιAct {Gₛ G}{AₛP}{≈ₐP}{ActP} (e G)      --Axiom GA2
```

2.8 Examples of Group Action

We look at a few examples of group action.

- Perm action on $\mathbb{A}$
  The permutation action on $\mathbb{A}$, as defined in definition (2.2) is a group action since it satisfies axiom (GA2) and axiom (GA3) by virtue of definition and axiom (GA1) by lemma (2.5).

- Perm action on Perm (composition)
  Perm can act on itself by composition, i.e. the function f : Perm $\times$ Perm $\to$ Perm defined as $f(\sigma, \pi) = \pi + \sigma$ is a group action. Here, axiom (GA3) holds by

lemma (2.11) , axiom (GA1) holds due to associativity whereas axiom (GA2) is satisfied definitionally.

- Perm action on Perm (conjugation)
  Perm can also act on itself by conjugation, i.e. the function f : Perm × Perm → Perm defined as $f(\sigma,\pi) = \sigma^{-1} + (\pi + \sigma)$ is also a group action. We prove this below.

Lemma 2.14

```
PermGP : GAct PermG
```

Proof. First, let us define the action.

Definition 2.15

```
PPAct : Perm → Perm → Perm
PPAct p p' = (invPerm p) ++ (p' ++ p)
```

Now, in this case, axiom (GA2) holds definitionally. We need to verify axioms (GA1) and (GA3). First we verify axiom (GA1). But to do so, we need to prove a lemma first.

Lemma 2.16

```
p₁p₂⁻¹≈p₂⁻¹p₁⁻¹ : (p₁ p₂ : Perm) → pEquiv (invPerm (p₁ ++ p₂))
                                   ((invPerm p₂) ++ (invPerm p₁))
```

Proof. For any $x \in \mathbb{A}$, we have:

$$
\begin{aligned}
& (p_1 + p_2)^{-1} \cdot x \\
&= (p_1 + p_1^{-1}) \cdot (p_1 + p_2)^{-1} \cdot x \quad \text{[By lemma (2.6)]} \\
&= p_1^{-1} \cdot p_1 \cdot (p_1 + p_2)^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= p_1^{-1} \cdot (p_2 + p_2^{-1}) \cdot p_1 \cdot (p_1 + p_2)^{-1} \cdot x \quad \text{[By lemma (2.6)]} \\
&= p_1^{-1} \cdot p_2^{-1} \cdot p_2 \cdot p_1 \cdot (p_1 + p_2)^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= p_1^{-1} \cdot p_2^{-1} \cdot (p_1 + p_2) \cdot (p_1 + p_2)^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= p_1^{-1} \cdot p_2^{-1} \cdot x \quad \text{[By lemma (2.7)]} \\
&= (p_2^{-1} + p_1^{-1}) \cdot x \quad \text{[By lemma (2.5)]}
\end{aligned}
$$

□

Now, we can verify axiom (GA1).

Lemma 2.17

```
p₁++p₂≡p₂p₁p : (p₁ p₂ : Perm) → (p : Perm) →
   pEquiv (PPAct (p₁ ++ p₂) p) (PPAct p₂ (PPAct p₁ p))
```

Proof. For $x \in \mathbb{A}$, we have,

$$
\begin{aligned}
&((p_1 + p_2)^{-1} + (p + (p_1 + p_2))) \cdot x \\
&= (p + (p_1 + p_2)) \cdot (p_1 + p_2)^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= (p + (p_1 + p_2)) \cdot (p_2^{-1} + p_1^{-1}) \cdot x \quad \text{[By lemma (2.16)]} \\
&= (p + (p_1 + p_2)) \cdot p_1^{-1} \cdot p_2^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= ((p + p_1) + p_2) \cdot p_1^{-1} \cdot p_2^{-1} \cdot x \quad \text{[By associativity]} \\
&= p_2 \cdot (p + p_1) \cdot p_1^{-1} \cdot p_2^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= p_2 \cdot (p_1^{-1} + (p + p_1)) \cdot p_2^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= ((p_1^{-1} + (p + p_1)) + p_2) \cdot p_2^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= (p_2^{-1} + ((p_1^{-1} + (p + p_1)) + p_2)) \cdot x \quad \text{[By lemma (2.5)]}
\end{aligned}
$$

□

Now, we verify axiom (GA3).

Lemma 2.18

```
resPP : (π₁ π₂ p₁ p₂ : Perm) → (π₁≈π₂ : pEquiv π₁ π₂)
                          → (p₁≈p₂ : pEquiv p₁ p₂) →
                       pEquiv (PPAct π₁ p₁) (PPAct π₂ p₂)
```

Proof. For $x \in \mathbb{A}$, we have,

$$
\begin{aligned}
&(\pi_1^{-1} + (p_1 + \pi_1)) \cdot x \\
&= (p_1 + \pi_1) \cdot \pi_1^{-1} \cdot x \quad \text{[By lemma (2.5)]} \\
&= (p_1 + \pi_1) \cdot \pi_2^{-1} \cdot x \quad \text{[By lemma (2.12)]} \\
&= (p_2 + \pi_2) \cdot \pi_2^{-1} \cdot x \quad \text{[By lemma (2.11)]} \\
&= (pi_2^{-1} + (p_2 + \pi_2)) \cdot x \quad \text{[By lemma (2.5)]}
\end{aligned}
$$

□

This shows that the conjugation action on Perm defines a group action. □

2.9 Perm-Set

A Perm-Set is a set with a permutation action, i.e.

Definition 2.19

```
PermSet = GAct PermG
```

As the previous section shows, $\mathbb{A}$ and Perm are Perm-Sets. In fact, we can define a Perm action on any set X given by $f : \text{Perm} \times X \to X$, where for all $\pi \in \text{Perm}, x \in X, f(\pi, x) = x$. Thus with this discrete action, $\mathbb{B}$, $\mathbb{N}$ are all Perm-Sets.

2.10 Equivariant functions

Given two Perm-Sets X and Y, a function $f : X \to Y$ is equivariant if it satisfies the following condition:

$$\forall p \in \text{Perm}, x \in X, \quad p \cdot (fx) = f(p \cdot x) \tag{EqP1}$$

For setoids, we must ensure that

$$\forall x, x' \in X, \quad x \approx x' \Rightarrow fx \approx fx' \tag{EqP2}$$

So we have the following record type:

Definition 2.20

```
record EquivarP (X Y : PermSet) : Set where
 field
  pufun : AₛP X → AₛP Y                                --the function
  peqi≈ : pufun Preserves (≈ₐP X) ⟶ (≈ₐP Y)            --axiom (EqP2)
  pequiv : (p : Perm) → (x : AₛP X) → (≈ₐP Y) ((ActP Y) p (pufun x))
                            (pufun ((ActP X) p x))  --axiom (EqP1)
```

2.11 Example of an equivariant function

We look at an example of an equivariant function. The permutation action on any Perm-Set X is itself an equivariant function, when the Perm action on Perm is defined by conjugation. To see this, first we note that the cartesian product of two Perm-Sets is again a Perm-Set, with the permutation action defined point-wise. Now, we prove that the action indeed gives an equivariant function.

Lemma 2.21

```
GX→XEq : (X : PermSet) → EquivarP (ProdP PermGP X) X
          --ProdP is the cartesian product function
```

Proof. Here, axiom (EqP2) holds due to axiom (GA3). We just need to verify axiom (EqP1). For $p \in \text{Perm}$, $(g, x) \in \text{Perm} \times X$, we have,

$$
\begin{aligned}
& p \cdot g \cdot x \\
&= (g + p) \cdot x \quad \text{[By Axiom (2.7)]} \\
&= (g + p) \cdot e \cdot x \quad \text{[By Axiom (2.8)]} \\
&= (g + p) \cdot (p + p^{-1}) \cdot x \quad \text{[By lemma (2.6)]} \\
&= (g + p) \cdot p^{-1} \cdot p \cdot x \quad \text{[By lemma (2.5)]} \\
&= (p^{-1} + (g + p)) \cdot p \cdot x \quad \text{[By lemma (2.5)]} \\
&= (p \cdot g) \cdot (p \cdot x) \quad \text{[By definition]}
\end{aligned}
$$

□

2.12 Useful properties of permutations

We now state and prove a few properties of permutations which will be useful later.

2.12.1 Atoms unchanged under action

Since we are working with finite permutations, given any $\pi \in \text{Perm}$, we can always find $a \in \mathbb{A}$ such that $\pi \cdot a = a$. To find such an a, we observe that because of our definition of Perm as lists of pairs of atoms, any a not present in π definitely remains unchanged under the action of π. Put precisely,

Definition 2.22

```
flatten : Perm → List Atom
flatten [] = []
flatten (p :: ps) = (proj₁ p) :: ((proj₂ p) :: (flatten ps))
```

Lemma 2.23

```
pa≡a : (p : Perm) → (a : Atom) → (a ∉ (flatten p)) → PermAct p a ≡ a
```

Proof. The base case for empty list is trivially true. Let this be true for lists of length n. We show that this holds for lists of length $n+1$. Let $p = (m \;\; n) :: q$. We have,

$$\begin{aligned} & ((m \;\; n) :: q) \cdot a \\ &= q \cdot (m \;\; n) \cdot a \quad \text{[By definition]} \\ &= q \cdot a \quad [\because a \neq m \wedge a \neq n \text{ by definition (1.4)}] \\ &= a \quad \text{[By induction hypothesis]} \end{aligned}$$

□

2.12.2 A permutation and a name-swap

The composition of a permutation and a name-swap gives us the following useful lemma.

Lemma 2.24

```
π+πab≈abπ : (π : Perm) → (a b : Atom) →
  pEquiv (π ++ [(PermAct π a) , (PermAct π b)]) ([(a , b)] ++ π)
```

Proof. For any $x \in \mathbb{A}$, we have the following cases,

- $x = a$

$$\begin{aligned} & (\pi + (\pi \cdot a \;\; \pi \cdot b)) \cdot a \\ &= (\pi \cdot a \;\; \pi \cdot b) \cdot \pi \cdot a \quad \text{[By lemma (2.5)]} \\ &= \pi \cdot b \quad \text{[By definition]} \\ &= \pi \cdot (a \;\; b) \cdot a \quad \text{[By definition]} \\ &= ((a \;\; b) + \pi) \cdot a \quad \text{[By lemma (2.5)]} \end{aligned}$$

Similarly, we can show that the lemma holds for $x = b$.

- $x \neq a, x \neq b$

$$
\begin{aligned}
& (\pi + (\pi \cdot a \quad \pi \cdot b)) \cdot x \\
= & (\pi \cdot a \quad \pi \cdot b) \cdot \pi \cdot x \quad \text{[By lemma (2.5)]} \\
= & \pi \cdot x \quad [\because \pi \cdot x \neq \pi \cdot a, \pi \cdot x \neq \pi \cdot b \text{ by lemma (2.3)}] \\
= & \pi \cdot (a \quad b) \cdot x \quad \text{[By definition]} \\
= & ((a \quad b) + \pi) \cdot x \quad \text{[By lemma (2.5)]}
\end{aligned}
$$

□

Now, we have the following corollaries from the above lemma.

Corollary 2.25

```
πabπ≈πab : (π : Perm) → (a b : Atom) → (b ∉ (flatten π)) →
        pEquiv (π ++ [((PermAct π a) , b)]) ([(a , b)] ++ π)
```

Proof. By lemma (2.24) and lemma (2.23). □

Corollary 2.26

```
ab∉pcom : (a b : Atom) → (p : Perm) → (a ∉ (flatten p)) →
 (b ∉ (flatten p)) → pEquiv ([( a , b )] ++ p) (p ++ [( a , b )])
```

Proof. By lemma (2.24) and lemma (2.23). □

Corollary 2.27

```
πsuppAx : (p : Perm) → (b c : Atom) →
   (b ∉ (flatten p)) → (c ∉ (flatten p)) → (a : Atom) →
   PermAct ([( b , c)] ++ (p ++ [( b , c )])) a ≡ PermAct p a
```

Proof. We have,

$$
\begin{aligned}
& ((b\ \ c) + (p + (b\ \ c))) \cdot a \\
&= ((b\ \ c) + ((b\ \ c) + p)) \cdot a \quad \text{[By lemma (2.26)]} \\
&= (((b\ \ c) + (b\ \ c)) + p) \cdot a \quad \text{[By associativity]} \\
&= p \cdot a \quad \text{[By idempotence of swapping]}
\end{aligned}
$$

□

Corollary 2.28

```
ac≈ab+bc+ab : (a b c : Atom) → (c ≡ a → ⊥) → (c ≡ b → ⊥) →
   pEquiv ([(a , c)]) ([(a , b)] ++ ([(b , c)] ++ [(a , b)]))
```

Proof. This can be easily seen to be true by a case analysis. We prove this in a different way. For $\pi \in \text{Perm}$, $a', b', x \in \mathbb{A}$, we have, by lemma (2.24),

$$
\begin{aligned}
(\pi + (\pi \cdot a'\ \ \pi \cdot b')) \cdot \pi^{-1} \cdot x &= ((a'\ \ b') + \pi) \cdot \pi^{-1} \cdot x \\
(\pi \cdot a'\ \ \pi \cdot b') \cdot \pi \cdot \pi^{-1} x &= (\pi^{-1} + ((a'\ \ b') + \pi)) \cdot x \quad \text{[By lemma (2.5)]} \\
(\pi \cdot a'\ \ \pi \cdot b') \cdot x &= (\pi^{-1} + ((a'\ \ b') + \pi)) \cdot x \quad \text{[By lemma (2.7)]}
\end{aligned}
$$

Putting $\pi = (a\ \ b), a' = b, b' = c$, we have,

$$
\begin{aligned}
((a\ \ b) \cdot b\ \ (a\ \ b) \cdot c) \cdot x &= ((a\ \ b) + ((b\ \ c) + (a\ \ b))) \cdot x \\
(a\ \ c) x &= ((a\ \ b) + ((b\ \ c) + (a\ \ b))) \cdot x \quad \text{[By definition]}
\end{aligned}
$$

□

We can now build the theory of nominal sets on top of the theory of permutations and Perm-Sets.

Chapter 3

A category theoretic perspective

'Just as group theory is the abstraction of the idea of a system of permutations of a set or symmetries of an object', category theory is the abstraction of the idea of functions between mathematical objects. As the symmetries of an object abstractly characterize it, the functions (or morphisms) of a category abstractly characterize the objects of the category. Since functions are ubiquitous in mathematics, category theory 'turns out to be a kind of universal mathematical language like set theory'(Awodey, 2006, [29]). Ever since its introduction by Eilenberg and Mac Lane (1945) [30], category theory has been applied to many different fields of mathematics including geometry, algebra and logic. In this chapter, we shall try to develop the theory of nominal sets in the language of category theory.

3.1 Nominal Set

A Perm-Set X is a nominal set if every $x \in X$ is finitely supported, i.e. given $x \in X$, we can find a finite subset $S_X(x) \subset A$ such that for all $\pi \in \mathrm{Perm}$,

$$((\forall a \in S_X(x)), \pi \cdot a = a) \Rightarrow \pi \cdot x = x \tag{S}$$

The support of an element is a set of names, each of which when remains unchanged by a permutation, the element itself also remains unchanged. For example, the support of an atomic name $a \in \mathbb{A}$ is any finite subset of $\mathbb{A}$ containing a. Thus, $\{a\}, \{a, b\}, \{a, b, c\}$ for $b, c \in \mathbb{A}$ are all supports for a. Classically, one can show that if $S_X(x)$ and $S'_X(x)$ are finite supports for $x \in X$, then so is $S_X(x) \cap S'_X(x)$ (Proposition 2.3, [6]). As such, one can work with a unique smallest support instead of any support. The smallest support

can be defined as (Page 30, [6]):

$$\text{supp}_X x \triangleq \cap \{A \in \text{Pow } \mathbb{A} \mid A \text{ is a finite support for } x\} \tag{SUPP}$$

where Pow $\mathbb{A}$ is the powerset of $\mathbb{A}$. But as pointed out in section (1.4), we cannot find $\text{supp}_X x$ constructively. So instead, we work with the notion of some non-unique finite support ($S_X(x) \supseteq \text{supp}_X x$). Since $S_X(x)$ is an overestimate of the support for x, there may be permutations π such that for all $a \in S_X(x), \pi \cdot a = a$ does not hold, yet we have $\pi \cdot x = x$. But axiom (S) still holds true. Now, since we are working with permutations represented as lists, we can express axiom (S) as:

$$\forall b, c \in \mathbb{A}, \qquad b \notin S_X(x) \;\wedge\; c \notin S_X(x) \Rightarrow (b \;\; c) \cdot x = x \tag{N}$$

If $b, c \notin S_X(x)$, then we know for sure that for all $a \in S_X(x), (b \;\; c) \cdot a = a$. By induction then, it can be shown that for any $\pi \in$ Perm, if (flatten π) $\cap S_X(x) = \varnothing$, then for all $a \in S_X(x), \pi \cdot a = a$. So axiom (N) is equivalent to:

$$\forall \pi \in \text{Perm}, \qquad (\text{flatten } \pi) \cap S_X(x) = \varnothing \Rightarrow \pi \cdot x = x \tag{SSUPP}$$

We have the following record type for nominal sets:

Definition 3.1

```
record Nominal : Set₁ where
  field
    Aₛ : Set                                      --underlying set
    ≈ₐ : Rel Aₛ                                   --relation on set
    eq≈ₐ : isEquivalence Aₛ ≈ₐ                   --proof of equivalence
    Act : Perm → Aₛ → Aₛ                          --Perm action
    res : Act Preserves₂ pEquiv ⟶ ≈                 --axiom (GA3)
    p₁p₂↠ : p₁p₂Act {Perm}{Aₛ}{_++_}{≈ₐ}{Act}   --axiom (GA1)
    ι↠ : ιAct {Perm}{Aₛ}{≈ₐ}{Act} ι             --axiom (GA2)

    some_supp : Aₛ → List Atom                   --some support
    suppAx : (a : Aₛ) → (b c : Atom) → (b ∉ some_supp a) →
             (c ∉ some_supp a) → ≈ₐ (Act [(b , c)] a) a  --axiom (N)
```

For any nominal set X and $x \in X$, we shall denote any finite support of x by $S_X(x)$. In case the nominal set referred to is clear from the context, we may abbreviate the

notation as $S(x)$. Since we are working with some support, $S(x)$ doesn't refer to a unique subset of $\mathbb{A}$.

3.2 Examples of Nominal Sets

We now look at several examples of nominal sets.

- $\mathbb{A}$
  $\mathbb{A}$ is a nominal set with $S_{\mathbb{A}}(a) = \{a\}$. That this satisfies axiom (N) follows by definition.
- Perm (conjugation)
  Perm with conjugation action is a nominal set with $S_{\text{Perm}}(\pi) = \text{flatten } \pi$. That this satisfies axiom (N) follows from corollary (2.27).
- Discrete Nominal Sets
  $\mathbb{B}, \mathbb{N}$ with the discrete action are nominal sets, where each element has an empty support. Axiom (N) is satisfied trivially.
- Finite subsets of $\mathbb{A}$
  We can show that the set of finite subsets of any nominal set is also a nominal set. Here, we show this for the nominal set $\mathbb{A}$. For this, we represent a subset of $\mathbb{A}$ as a list of atoms. Now if we define the setoid relation as the strict propositional equality, then two equal subsets of $\mathbb{A}$ may be treated as different since lists may contain multiple instances of the same element so that two lists may contain the same set of elements but some of them occurring different number of times. In order to avoid such misclassifications, we define the following subset relation:

Definition 3.2

```
data _<_ {A : Set} (a : A) : List A → Set where
  a<a : ∀ {as} → a <  (a :: as)
  a<as : ∀ {as}{b : A} → a < as → a < (b :: as)

record <rel (A : Set)(L1 : List A)(L2 : List A) : Set where
  constructor L1◯L2
  field
    L1⊆L2 : ∀ {a : A} → (a < L1) → (a < L2)
    L2⊆L1 : ∀ {a : A} → (a < L2) → (a < L1)
```

Now, the equivalence property of the relation can be easily verified. With the permutation action defined point-wise on each element of the list, we get a nominal set where every list is supported by itself. That axioms (GA1), (GA2), (N) hold follow from the corresponding properties of $\mathbb{A}$. To verify axiom (GA3), we need two subordinate lemmas, which we state without proof.

Lemma 3.3

```
a<l⇒πa<πl : {a : Atom}{l : List Atom} → (π₁ π₂ : Perm) →
     (pEquiv π₁ π₂) → a < l → (PermAct π₁ a) < (ActList π₂ l)
                              --ActList is PermAct on List
```

Lemma 3.4

```
πa<πl⇒a<l : {a : Atom}{L : List Atom} → (π : Perm) →
       a < (ActList π L) → (PermAct (invPerm π) a) < L
```

Now, we verify axiom (GA3).

Lemma 3.5

```
ActListRes : {L₁ L₂ : List Atom}{π₁ π₂ : Perm} →
        (π₁≈π₂ : pEquiv π₁ π₂) → (L₁≈L₂ : <rel Atom L₁ L₂) →
                  <rel Atom (ActList π₁ L₁) (ActList π₂ L₂)
```

Proof. For any $a \in \pi_1 \cdot L_1$, we have,

$$\begin{aligned} & a < \pi_1 \cdot L_1 \\ &= \pi_1^{-1} \cdot a < L_1 \quad \text{[By lemma (3.4)]} \\ &= \pi_1^{-1} \cdot a < L_2 \quad [\because L_1 \approx L_2] \\ &= \pi_1 \cdot \pi_1^{-1} \cdot a < \pi_2 \cdot L_2 \quad [\text{By lemma (3.3)}, \because \pi_1 \approx \pi_2] \\ &= a < \pi_2 \cdot L_2 \end{aligned}$$

This shows that $\pi_1 \cdot L_1 \subseteq \pi_2 \cdot L_2$. Similarly, we can prove that $\pi_2 \cdot L_2 \subseteq \pi_1 \cdot L_1$. The lemma follows. □

- Algebraic signature

  A single-sorted algebraic signature Sig with variables from a set A can be defined as:

  Definition 3.6

```
data Sig (A : Set) : Set where
 base : (a : A) → Sig A
 build : (n : ℕ) → Vec (Sig A) n → Sig A
```

For any nominal set N, we can show that Sig N is again a nominal set. The action and support can be defined as follows:

Definition 3.7

```
VecAct : (N : Nominal) → Perm → (n : ℕ) → Vec (Sig (As N)) n
                                           → Vec (Sig (As N)) n
SigAct : (N : Nominal) → Perm → Sig (As N) → Sig (As N)

VecAct N π zero [] = []
VecAct N π (succ n) (x :: xs) = (SigAct N π x) :: (VecAct N π n xs)
SigAct N π (base a) = base ((Act N) π a)
SigAct N π (build n xs) = build n (VecAct N π n xs)
```

```
Vecsupp : (N : Nominal) → (n : ℕ) → Vec (Sig (As N)) n → List Atom
Sigsupp : (N : Nominal) → Sig (As N) → List Atom

Vecsupp N zero [] = []
Vecsupp N (succ n) (x :: xs) = (Sigsupp N x) ++ (Vecsupp N n xs)
Sigsupp N (base a) = (some_supp N) a
Sigsupp N (build n xs) = Vecsupp N n xs
```

That the axioms are obeyed by Sig N follow from the fact that they are obeyed by N.

3.3 Functions between nominal sets

Now we look at functions which preserve the nominal set structure. Just as equivariant functions preserve G-Set structure, finitely supported functions preserve nominal set structure. In fact, equivariant functions are also finitely supported functions, where the support is the empty set. Given nominal sets X and Y, a function $f : X \to Y$ is finitely supported, if there exists a finite subset $S_{X\to Y}(f) \subset \mathbb{A}$ such that for all $\pi \in$ Perm,

$$((\forall a \in S_{X\to Y}(f)), \pi \cdot a = a) \Rightarrow (\forall x \in X), \pi^{-1} \cdot f(\pi \cdot x) = fx \qquad \text{(FS)}$$

Here again, we shall work with some support instead of the smallest support. In line with our definition for nominal set, we modify axiom (FS) as:

$$\forall b, c \in \mathbb{A}, \quad b \notin S_{X \to Y}(f), c \notin S_{X \to Y}(f) \Rightarrow (\forall x \in X), (b \;\; c) \cdot f((b \;\; c) \cdot x) = fx \qquad \text{(FS1)}$$

But since we are working with setoids, we need to add an additional axiom:

$$\forall x_1, x_2 \in X, \quad x_1 \approx x_2 \Rightarrow fx_1 \approx fx_2 \qquad \text{(FS2)}$$

So, we have the following record type:

Definition 3.8

```
record Funfs (X Y : Nominal) : Set where
   field
      ffun :  As X → As Y                              --the function
      fsupp : List Atom                                --support
      fsuppAx : (b c : Atom) → (b ∉ fsupp) → (c ∉ fsupp) →
      (∀ (x : (As X)) → (≈a Y) ((Act Y) [( b , c )]
      (ffun ((Act X) [( b , c )] x))) (ffun x))
                                                      --axiom (FS1)
      feqi≈ : ffun Preserves (≈a X) ⟶ (≈a Y)          --axiom (FS2)
```

Now let us look at an example of a finitely supported function. For any $\pi \in$ Perm, the function $f : \mathbb{A} \to \mathbb{A}$ given by $f(a) = \pi \cdot a$ is a finitely supported function. The support for this function can be taken to be (flatten π). Axiom (FS1) then follows from corollary (2.27)and axiom (FS2) follows by definition, showing that f is a finitely supported function.

The definition for an equivariant function between nominal sets remains same as that between the corresponding Perm-sets (given in definition (2.20)). It's easy to check that an equivariant function is also a finitely supported function. The support for an equivariant function is the empty list. Axiom (FS2) is obeyed due to axiom (EqP2). We can now check that axiom (FS1) is also obeyed.

$$\begin{aligned} & (b \;\; c) \cdot f((b \;\; c) \cdot x) \\ = & (b \;\; c) \cdot (b \;\; c) \cdot (fx) \quad \text{[By equivariance property, axiom (EqP1)]} \\ = & fx \quad \text{[By idempotence of swapping]} \end{aligned}$$

Once we have got nominal sets and functions between nominal sets, we can now look at the category of nominal sets.

3.4 Nom and $\mathbf{Nom_{fs}}$

A category $\mathbb{C}$ consists of objects and morphisms between those objects such that the following axioms are satisfied:

$$\text{Given } A \xrightarrow{f} B \xrightarrow{g} C \text{ in } \mathbb{C}, \text{ we have } A \xrightarrow{g\circ f} C \tag{C1}$$

$$\text{Given } A \in \text{Obj } \mathbb{C}, \text{ we have } \text{id}_A \in \mathbb{C}(A,A) \tag{C2}$$

$$\text{Given } A \xrightarrow{f} B, \quad f \circ \text{id}_A = f \tag{C3}$$

$$\text{Given } A \xrightarrow{g} B, \quad \text{id}_B \circ g = g \tag{C4}$$

$$\text{Given } A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D, \quad h \circ (g \circ f) = (h \circ g) \circ f \tag{C5}$$

To recover extensionality, we replace the equality relation between morphisms with an equivalence relation and demand that the composition operation respect this equivalence relation, i.e.

$$f_1, f_2 \in \mathbb{C}(A,B), g_1, g_2 \in \mathbb{C}(B,C), \qquad f_1 \approx f_2 \wedge g_1 \approx g_2 \Rightarrow g_1 \circ f_1 \approx g_2 \circ f_2 \tag{C6}$$

Therefore we have the following record type:

Definition 3.9

```
record Cat : Set₂ where
 field
  Obj : Set₁                                                   --objects
  _⇒_ : Obj → Obj → Set                                        --morphisms
  _≈_ : ∀ {M N} → Rel (M ⇒ N)                                  --relation
  isEq : ∀ {M N} →  isEquivalence (M ⇒ N) _≈_                  --equivalence
  id : ∀ {N} → N ⇒ N                                           --axiom (C2)
  _◎_ : ∀ {M N O} → M ⇒ N → N ⇒ O → M ⇒ O                      --axiom (C1)
  id1 : ∀ {M N} → (f : M ⇒ N) → (id ◎ f) ≈ f                   --axiom (C3)
  id2 : ∀ {M N} →  (g : M ⇒ N)  → (g ◎ id) ≈ g                 --axiom (C4)
  asso : ∀ {M N O P} → (f : M ⇒ N) → (g : N ⇒ O) →
         (h : O ⇒ P) → ((f ◎ g) ◎ h) ≈ (f ◎ (g ◎ h))          --axiom (C5)
```

```
resp :  ∀ {M N O} → (_◎_ {M}{N}{O}) Preserves₂ _≈_ ⟶ _≈_ ⟶ _≈_
                                                    --axiom (C6)
```

Now, we can show that nominal sets and finitely supported functions form a category $\mathbf{Nom_{fs}}$. The equivalence relation is defined as extensional functional equality. The identity morphism is given by emptily supported identity function. Axioms (C3) and (C4) are obeyed definitionally. The composition of two finitely supported functions $f : X \to Y$ and $g : Y \to Z$ is again a finitely supported function $g \circ f : X \to Z$ whose support is given by $S_{X\to Z}(g \circ f) = S_{X\to Y}(f) + S_{Y\to Z}(g)$. This obeys axiom (FS1) as for all $x \in X$, if $b, c \notin S_{X\to Z}(g \circ f)$, we have:

$$
\begin{aligned}
& (b \;\; c) \cdot (g \circ f)(b \;\; c) \cdot x \\
= & \, (b \;\; c) \cdot g((b \;\; c) \cdot (fx)) \quad [\because b, c \notin S_{X\to Y}(f)] \\
= & \, g(fx) \quad [\because b, c \notin S_{Y\to Z}(g)]
\end{aligned}
$$

Axioms (C5) and (C6) are obeyed by definition. Hence, $\mathbf{Nom_{fs}}$ is a category.

It is interesting to note that not only the nominal sets and the finitely supported functions form a category $\mathbf{Nom_{fs}}$, but also the nominal sets and the equivariant functions form a category **Nom**. $\mathbf{Nom_{fs}}$ and **Nom** have the same objects, but the morphisms of **Nom** form a proper subset of the morphisms of $\mathbf{Nom_{fs}}$. Thus **Nom** is a wide or lluf subcategory (term first used by Freyd, 1991, [31]) of $\mathbf{Nom_{fs}}$. We now check that **Nom** is indeed a category. The relation on morphisms is again extensional functional equality. Axioms (C2), (C3) and (C4) are easy to verify. We just show that axiom (C1) is satisfied and axioms (C5) and (C6) will then follow. For equivariant functions $f : X \to Y$ and $g : Y \to Z$, we have, for any $\pi \in \text{Perm}$ and $x \in X$,

$$
\begin{aligned}
& \pi \cdot (g \circ f)x \\
= & \, g(\pi \cdot (fx)) \quad [\because g \text{ is equivariant}] \\
= & \, g(f(\pi \cdot x)) \quad [\because f \text{ is equivariant}]
\end{aligned}
$$

Once we have shown that **Nom** and $\mathbf{Nom_{fs}}$ are categories, we now show that both of them have products, coproducts and exponentials. But before that, let us look at the initial and terminal objects in **Nom** and $\mathbf{Nom_{fs}}$. The empty set ($\bot$) is an initial object while a singleton set ($\top$) is a terminal object both in **Nom** and $\mathbf{Nom_{fs}}$. The unique morphisms from the initial object and to the terminal object are as in **Set**. Such morphisms are equivariant functions and hence are finitely supported, making $\bot$ the initial and $\top$ the terminal object in both **Nom** and $\mathbf{Nom_{fs}}$.

3.5 Products in Nom and $\mathbf{Nom_{fs}}$

For a category $\mathbb{C}$, the product of $X, Y \in \text{Obj } \mathbb{C}$ is given by $X \times Y \in \text{Obj } \mathbb{C}$ along with the projection morphisms $\pi_1 \in \mathbb{C}(X \times Y, X)$ and $\pi_2 \in \mathbb{C}(X \times Y, Y)$ such that for any $Z \in \text{Obj } \mathbb{C}$ and $f \in \mathbb{C}(Z, X), g \in \mathbb{C}(Z, Y)$, $\exists! \langle f, g \rangle \in \mathbb{C}(Z, X \times Y)$ such that $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$.

$$\begin{array}{ccccc} & & Z & & \\ & \swarrow^{f} & \downarrow{\scriptstyle \langle f,g \rangle} & \searrow^{g} & \\ X & \xleftarrow[\pi_1]{} & X \times Y & \xrightarrow[\pi_2]{} & Y \end{array}$$

For nominal sets X and Y, $X \times Y$ is the cartesian product of X and Y, with π_1 and π_2 the first and second projection morphisms respectively. Since X, Y are nominal sets, the cartesian product of X and Y is also a nominal set. That π_1 and π_2 are equivariant functions follow from definition. Now, for any nominal set Z and equivariant functions $f \in \mathbf{Nom}\ (Z, X)$ and $g \in \mathbf{Nom}\ (Z, Y)$, $\langle f, g \rangle \in \mathbf{Nom}\ (Z, X \times Y) = \lambda z \to (fz, gz)$. This is an equivariant function since f and g are equivariant functions. It is easy to check that the existence and uniqueness conditions are satisfied. If instead of equivariant functions, we have finitely supported functions $f \in \mathbf{Nom_{fs}}\ (Z, X), g \in \mathbf{Nom_{fs}}\ (Z, Y)$, then $\langle f, g \rangle \in \mathbf{Nom}\ (Z, X \times Y) = \lambda z \to (fz, gz)$ is a finitely supported function with $S_{Z \to X \times Y} \langle f, g \rangle = S_{Z \to X}(f) \cup S_{Z \to Y}(g)$. That $S(\langle f, g \rangle)$ supports $\langle f, g \rangle$ can be seen from the fact that for any $b, c \in \mathbb{A}$, if $b, c \notin S(\langle f, g \rangle)$, then $b, c \notin S(f)$ and $b, c \notin S(g)$ and as such for any $z \in Z$, $(b\ \ c) \cdot \langle f, g \rangle((b\ \ c) \cdot z) = \langle f, g \rangle(z)$. The existence and uniqueness conditions are satisfied by definition.

Thus, **Nom** and $\mathbf{Nom_{fs}}$ are cartesian categories.

3.6 Coproducts in Nom and $\mathbf{Nom_{fs}}$

For a category $\mathbb{C}$, the coproduct of $X, Y \in \text{Obj } \mathbb{C}$ is given by $X + Y \in \text{Obj } \mathbb{C}$ along with the injection morphisms $i_1 \in \mathbb{C}(X, X + Y)$ and $i_2 \in \mathbb{C}(Y, X + Y)$ such that for any $Z \in \text{Obj } \mathbb{C}$ and $f \in \mathbb{C}(X, Z), g \in \mathbb{C}(Y, Z)$, $\exists! [f, g] \in \mathbb{C}(X + Y, Z)$ such that $[f, g] \circ i_1 = f$ and $[f, g] \circ i_2 = g$.

$$\begin{array}{ccccc} & & Z & & \\ & \overset{f}{\nearrow} & \uparrow [f,g] & \overset{g}{\nwarrow} & \\ X & \xrightarrow{i_1} & X+Y & \xleftarrow{i_2} & Y \end{array}$$

For nominal sets X and Y, $X+Y$ is given by the disjoint union of X and Y defined as:

Definition 3.10

```
data _⊎_ (A : Set)(B : Set) : Set where
  inl : (x : A) → A ⊎ B
  inr : (y : B) → A ⊎ B
```

Since X and Y are nominal sets, the disjoint union $X \uplus Y$ is also a nominal set. The injection morphisms are given as $i_1(x) = \text{ inl } x, i_2(y) = \text{ inr } y$. That these functions are equivariant follow by definition. Now for any nominal set Z and equivariant functions $f \in \textbf{Nom } (X,Z), g \in \textbf{Nom } (Y,Z)$,

$$\begin{aligned} [f,g]w = \text{match } w \text{ with} \\ \mid \text{inl } x \to fx \\ \mid \text{inr } y \to gy \end{aligned}$$

That $[f,g]$ is equivariant follows from the equivariance of f and g. The existence and uniqueness properties are then satisfied definitionally. If f, g are finitely supported functions, then a support for $[f,g]$ can be given by the union of the supports of f and g. By reasoning as in the previous section, we can show that this indeed is a support for $[f,g]$. The existence and uniqueness properties can then be easily verified.

Thus, **Nom** and $\textbf{Nom}_{\textbf{fs}}$ are cocartesian categories.

3.7 Exponentials in Nom and $\textbf{Nom}_{\textbf{fs}}$

For a category $\mathbb{C}$, the exponential of $X, Y \in \text{ Obj } \mathbb{C}$ is given by $Y^X \in \text{ Obj } \mathbb{C}$ along with the application morphism app $\in \mathbb{C}(Y^X \times X, Y)$ such that for any $Z \in \text{ Obj } \mathbb{C}$ and $f \in (Z \times X, Y)$, $\exists! \hat{f} \in \mathbb{C}(Z, Y^X)$ such that app $\circ (\hat{f} \times \text{id}_X) = f$.

$$
\begin{array}{ccc}
Y^X \times X & \xrightarrow{\text{app}} & Y \\
\uparrow {\scriptstyle \hat{f}\times \mathrm{id}_X} & \nearrow {\scriptstyle f} & \\
Z \times X & &
\end{array}
$$

For nominal sets X and Y, the exponential object Y^X is given by $\mathrm{Hom}_{\mathbf{Nom_{fs}}}(X, Y)$, i.e. the set of all finitely supported functions from X to Y. First, we need to check that $\mathrm{Hom}_{\mathbf{Nom_{fs}}}(X, Y)$ is indeed a nominal set.

3.7.1 Exponential object

The action of a permutation $\pi \in \mathrm{Perm}$ on a finitely supported function $f : X \to Y$ is defined as:

$$\pi \cdot f = \lambda x \in X \to \pi \cdot f(\pi^{-1} \cdot x) \tag{AFS}$$

Now we check that $\pi \cdot f$ is indeed a finitely supported function with $S_{X\to Y}(\pi \cdot f) = S_{X\to Y}(f) \cup (\text{flatten } \pi)$. For any $b, c \in \mathbb{A}$, if $b \notin S(\pi \cdot f)$, $c \notin S(\pi \cdot f)$, then we have, for all $x \in X$,

$$
\begin{aligned}
& (b\ \ c) \cdot (\pi \cdot f)((b\ \ c) \cdot x) \\
= {} & (b\ \ c) \cdot \pi \cdot f(\pi^{-1} \cdot (b\ \ c) \cdot x) \quad \text{[By definition]} \\
= {} & \pi \cdot (b\ \ c) \cdot f(\pi^{-1} \cdot (b\ \ c) \cdot x) \quad \text{[By corollary (2.26)]} \\
= {} & \pi \cdot (b\ \ c) \cdot f((b\ \ c) \cdot \pi^{-1} \cdot x) \quad \text{[By corollary (2.26)]} \\
= {} & \pi \cdot f(\pi^{-1} \cdot x) \quad [\because b, c \notin S(f)] \\
= {} & (\pi \cdot f)x
\end{aligned}
$$

We now check that the group axioms are satisfied. It is easy to check that axiom (GA2) is satisfied. We just show that axioms (GA1) and (GA3) are satisfied. For $\pi_1, \pi_2 \in \mathrm{Perm}$, we have for all $x \in X$,

$$
\begin{aligned}
& (\pi_1 \cdot \pi_2 \cdot f)x \\
= {} & \pi_1 \cdot \pi_2 \cdot f(\pi_2^{-1} \cdot \pi_1^{-1} \cdot x) \quad \text{[By definition]} \\
= {} & (\pi_2 + \pi_1) \cdot f((\pi_1^{-1} + \pi_2^{-1}) \cdot x) \quad \text{[By axiom (GA1)]} \\
= {} & (\pi_2 + \pi_1) \cdot f((\pi_2 + \pi_1)^{-1} \cdot x) \quad \text{[By lemma (2.16)]} \\
= {} & ((\pi_2 + \pi_1) \cdot f)x \quad \text{[By definition]}
\end{aligned}
$$

Now, for $\pi_1, \pi_2 \in$ Perm and $f_1, f_2 \in \mathbf{Nom_{fs}}\ (X,Y)$, if $\pi_1 \approx \pi_2$ and $f_1 \approx f_2$, then we have, for all $x \in X$,

$$
\begin{aligned}
& (\pi_1 \cdot f_1)x \\
&= \pi_1 \cdot f_1(\pi_1^{-1} \cdot x) \quad \text{[By definition]} \\
&= \pi_1 \cdot f_1(\pi_2^{-1} \cdot x) \quad \text{[By lemma (2.12), } \because \pi_1 \approx \pi_2\text{]} \\
&= \pi_2 \cdot f_2(\pi_2^{-1} \cdot x) \quad [\because \pi_1 \approx \pi_2, f_1 \approx f_2] \\
&= (\pi_2 \cdot f_2)x \quad \text{[By definition]}
\end{aligned}
$$

This shows that (AFS) defines a group action. Now, to show that $\mathrm{Hom}_{\mathbf{Nom_{fs}}}\ (X,Y)$ is a nominal set, we just need to show that every $f \in \mathbf{Nom_{fs}}\ (X,Y)$ is finitely supported. But this is true since f is a finitely supported function, and as such axiom (N) is satisfied due to axiom (FS1).

3.7.2 Application morphism

The application morphism app $\in \mathbf{Nom}\ (Y^X \times X, Y)$ is given by app $(f,x) = fx$. We now check that this definition satisfies the equivariance property. For any $\pi \in$ Perm and any $(f,x) \in Y^X \times X$, we have,

$$
\begin{aligned}
& \text{app } (\pi \cdot f, \pi \cdot x) \\
&= (\pi \cdot f)(\pi \cdot x) \quad \text{[By definition]} \\
&= \pi \cdot f(\pi^{-1} \cdot \pi \cdot x) \quad \text{[By definition]} \\
&= \pi \cdot (fx) \\
&= \pi \cdot (\text{ app } (f,x)) \quad \text{[By definition]}
\end{aligned}
$$

Once we have got the exponential object and the application morphism, we can now look for curried functions.

3.7.3 Currying in Nom

Given a nominal set Z and an equivariant function $f : \mathbf{Nom}\ (Z \times X, Y)$, we define the curried equivariant function $\hat{f} \in \mathbf{Nom}\ (Z, Y^X)$ as:

$$
\hat{f}z = f_z, \text{ where } f_z \in \mathbf{Nom_{fs}}\ (X,Y) = \lambda x \to f(z,x) \tag{CURRN}
$$

We can show that f_z is a finitely supported function with $S_{X\to Y}(f_z) = S_Z(z)$. For any $b, c \in \mathbb{A}$, if $b, c \notin S(f_z)$, then we have, for all $x \in X$:

$$
\begin{aligned}
&(b\ \ c)\cdot f_z((b\ \ c)\cdot x)\\
&= (b\ \ c)\cdot f(z,(b\ \ c)\cdot x) \quad \text{[By (CURRN)]}\\
&= f((b\ \ c)\cdot z,(b\ \ c)\cdot(b\ \ c)\cdot x) \quad \text{[By equivariance of } f\text{]}\\
&= f((b\ \ c)\cdot z, x) \quad \text{[By idempotence of swapping]}\\
&= f(z,x) \quad [\because b, c \notin S(z)]\\
&= f_z x \quad \text{[By (CURRN)]}
\end{aligned}
$$

Now, we show that $\hat{f}$ satisfies the equivariance property. For any $\pi \in \mathrm{Perm}$ and any $z \in Z$, we have for all $x \in X$,

$$
\begin{aligned}
&(\pi\cdot f_z)x\\
&= \pi f_z(\pi^{-1}\cdot x) \quad \text{[By (AFS)]}\\
&= \pi f(z,\pi^{-1}\cdot x) \quad \text{[By (CURRN)]}\\
&= f(\pi\cdot z,\pi\cdot\pi^{-1}\cdot x) \quad \text{[By equivariance of } f\text{]}\\
&= f(\pi\cdot z, x)\\
&= f_{\pi\cdot z}x \quad \text{[By (CURRN)]}
\end{aligned}
$$

We now just need to check the existence and uniqueness conditions. To prove the existence condition, we note that for all $(z,x) \in Z \times X$,

$$
\begin{aligned}
&\mathrm{app}\ \circ(\hat{f}\times \mathrm{id}_X)(z,x)\\
&= \mathrm{app}\ (f_z, x) \quad \text{[By (CURRN)]}\\
&= f_z x \quad \text{[By definition]}\\
&= f(z,x) \quad \text{[By (CURRN)]}
\end{aligned}
$$

The uniqueness condition follow from function extensionality. Hence **Nom** is a cartesian closed category. We can show that $\mathbf{Nom_{fs}}$ is also a cartesian category. The exponential object and the application morphism remain the same as in **Nom**. The curried function needs a little modification.

3.7.4 Currying in $\mathbf{Nom_{fs}}$

Given a nominal set Z and a finitely supported function $f : \mathbf{Nom}\ (Z \times X, Y)$, we define the curried finitely supported function $\hat{f} \in \mathbf{Nom}\ (Z, Y^X)$ as:

$$\hat{f}z = f_z, \text{ where } f_z \in \mathbf{Nom_{fs}}\ (X, Y) = \lambda x \to f(z, x) \qquad \text{(CURRF)}$$

We can show that f_z is a finitely supported function with $S_{X \to Y}(f_z) = S_{Z \times X \to Y}(f) \cup S_Z(z)$. For any $b, c \in \mathbb{A}$, if $b, c \notin S(f_z)$, then we have, for all $x \in X$:

$$\begin{aligned}
&(b\ \ c) \cdot f_z((b\ \ c) \cdot x) \\
&= (b\ \ c) \cdot f(z, (b\ \ c) \cdot x) \quad \text{[By (CURRF)]} \\
&= (b\ \ c) \cdot f((b\ \ c) \cdot z, (b\ \ c) \cdot x) \quad [\because b, c \notin S(z)] \\
&= (b\ \ c) f((b\ \ c) \cdot (z, x)) \quad \text{[By definition]} \\
&= f(z, x) \quad [\because b, c \notin S(f)] \\
&= f_z x \quad \text{[By (CURRF)]}
\end{aligned}$$

We now show that $\hat{f}$ is a finitely supported function with $S_{Z \to Y^X}(\hat{f}) = S_{Z \times X \to Y}(f)$. For any $b, c \in \mathbb{A}$, if $b, c \notin S(\hat{f})$, then given $z \in Z$, we have for all $x \in X$,

$$\begin{aligned}
&((b\ \ c) \cdot \hat{f}((b\ \ c) \cdot z))x \\
&= ((b\ \ c) \cdot f_{(b\ \ c) \cdot z})x \quad \text{[By (CURRF)]} \\
&= (b\ \ c) f((b\ \ c) \cdot z, (b\ \ c) \cdot x) \quad \text{[By (AFS)]} \\
&= (b\ \ c) f((b\ \ c) \cdot (z, x)) \quad \text{[By definition]} \\
&= f(z, x) \quad [\because b, c \notin S(f)] \\
&= \hat{f}(z)x \quad \text{[By (CURRF)]}
\end{aligned}$$

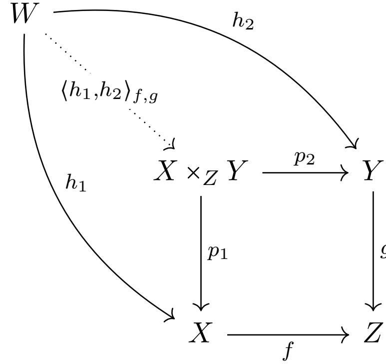
The existence and uniqueness conditions are satisfied as in the previous section. Thus, $\mathbf{Nom_{fs}}$ is also a cartesian closed category.

Now we look at pullbacks in **Nom**.

3.8 Pullbacks in Nom

For any category $\mathbb{C}$ and any $X, Y, Z \in \text{Obj}\ \mathbb{C}$, the pullback for a pair of morphisms $f \in \mathbb{C}(X, Z), g \in \mathbb{C}(Y, Z)$ is given by $X \times_Z Y \in \text{Obj}\ \mathbb{C}$ along with the morphisms $p_1 \in \mathbb{C}(X \times_Z Y, X), p_2 \in \mathbb{C}(X \times_Z Y, Y)$ such that $f \circ p_1 = g \circ p_2$ and for any $W \in \text{Obj}\ \mathbb{C}$ and

$h_1 \in \mathbb{C}(W, X), h_2 \in \mathbb{C}(W, Y)$, if $f \circ h_1 = g \circ h_2$, then $\exists! \langle h_1, h_2 \rangle_{f,g} \in \mathbb{C}(W, X \times_Z Y)$ such that $p_1 \circ \langle h_1, h_2 \rangle_{f,g} = h_1$ and $p_2 \circ \langle h_1, h_2 \rangle_{f,g} = h_2$.



Before showing that **Nom** has pullbacks, we need to show that an equivariant subset of a nominal set is again a nominal set. For any Perm-Set X, a subset $S \subset X$ is equivariant if for all $\pi \in$ Perm, we have,

$$a \in S \Rightarrow \pi \cdot a \in S \tag{EQC}$$

An equivariant subset of a Perm-set is itself a Perm-Set since it inherits the properties of the superset and is itself closed under Perm action. For any equivariant subset S of a nominal set X, if we assign $S_S(a) = S_X(a)$, then S is also a nominal set. Thus, given any condition for selecting elements from a nominal set, if we can show that the condition satisfies (EQC), we know that the subset so selected will be a nominal set.

Let us now consider nominal sets X, Y, Z and equivariant functions $f \in$ **Nom** $(X, Z), g \in$ **Nom** (Y, Z). We can then give the pullback object for f, g as:

$$X \times_Z Y = \{(x, y) \in X \times Y \mid fx = gy\} \tag{PULL}$$

That the condition mentioned in (PULL) is an equivariant one follows from the fact that f, g are equivariant functions. Thus $X \times_Z Y$ is a nominal set. Now p_1 and p_2 are given by the first and second projection morphisms. For any nominal set W and equivariant functions $h_1 \in$ **Nom** $(W, X), h_2 \in$ **Nom** (W, Y) with $f \circ h_1 = g \circ h_2$, we have $\langle h_1, h_2 \rangle_{f,g} = \lambda w \to (fw, gw)$. The function so defined is equivariant since f and g are equivariant. The existence and uniqueness conditions then follow by definition.

When $Z = \mathbb{B}$ and $Y = \top$ with $g(\mathbb{1}) =$ true , then for each equivariant function $f \in$ **Nom** $(X, \mathbb{B})$, we get an equivariant subset of X.

$$\begin{array}{ccc} X \times_{\mathbb{B}} \top & \xrightarrow{\langle\rangle} & \top \\ {\scriptstyle p_1} \big\downarrow & & \big\downarrow {\scriptstyle \text{true}} \\ X & \xrightarrow[f]{} & \mathbb{B} \end{array}$$

Here p_1 is a subobject of X and $\mathbb{B}$ acts as a subobject classifier. The subobjects of X are in bijection with the equivariant functions from X to $\mathbb{B}$, i.e. $\mathrm{Sub}(X) \cong \mathrm{Hom}_{\mathbf{Nom}}(X, \mathbb{B})$. Since **Nom** is cartesian closed and has a subobject classifier, therefore **Nom** is a topos.

It is worth noting that $\mathbf{Nom_{fs}}$ does not have pullbacks, since the condition in (PULL) is not an equivariant one if f and g are finitely supported functions instead of equivariant functions. However, as we show later, the nominal powerset of X is given by $\mathrm{Hom}_{\mathbf{Nom_{fs}}}(X, \mathbb{B})$.

3.9 Nom-sets and Perm-sets

We can easily check that Perm-Sets and equivariant functions form a category (**Perm**). **Nom** is then a subcategory of **Perm**. So, we have an inclusion functor I from **Nom** to **Perm**. We can show that I has a right adjoint. For categories $\mathbb{C}, \mathbb{D}$ and functors $F : \mathbb{C} \to \mathbb{D}$ and $G : \mathbb{D} \to \mathbb{C}$, G is a right adjoint of F (written $F \dashv G$) if:

$$\forall D \in \text{ Obj } \mathbb{D}, \quad \exists \epsilon_D \in \mathbb{D}(FGD, D) \tag{AD1}$$

$$\forall C \in \text{Obj } \mathbb{C}, D \in \text{Obj } \mathbb{D}, f \in \mathbb{D}(FC, D), \quad \exists! \bar{f} \in \mathbb{C}(C, GD) \text{ such that } \epsilon_D \circ F\bar{f} = f \tag{AD2}$$

In $\mathbb{D}$,

$$\begin{array}{ccc} FC & & \\ {\scriptstyle F\bar{f}} \big\downarrow & \searrow^{f} & \\ FGD & \xrightarrow[\epsilon_D]{} & D \end{array}$$

Now, let us define a functor PN : **Perm** $\to$ **Nom**. A functor F from category $\mathbb{C}$ to category $\mathbb{D}$ is a mapping which assigns to every $X \in \text{Obj } \mathbb{C}$, an object $FX \in \text{Obj } \mathbb{D}$ and to every morphism $f \in \mathbb{C}(X, Y)$, a morphism $Ff \in \mathbb{D}(FX, FY)$ such that the following conditions are satisfied:

$$\forall X \in \text{Obj } \mathbb{C}, \quad F(\mathrm{id}_X) = \mathrm{id}_{FX} \tag{FUN1}$$

$$\text{Given } X \xrightarrow{f} Y \xrightarrow{g} Z \text{ in } \mathbb{C}, \quad F(g \circ f) = Fg \circ Ff \tag{FUN2}$$

But since we are working with setoids, we need to add an additional axiom:

$$\text{Given } f_1, f_2 \in \mathbb{C}(X,Y), \quad f_1 \approx f_2 \Rightarrow F f_1 \approx F f_2 \tag{FUN3}$$

Now for any $P \in \text{Obj}\ \mathbf{Perm}$, we define $PN(P)$ as:

$$PN(P) \triangleq \{p \mid p \text{ is finitely supported in } P\} \tag{PNO}$$

We have the following record type for this:

Definition 3.11

```
record FinSupp (P : PermSet) : Set where
   field
    pel : AₛP P
    psupp : List Atom
    psuppAx : (b c : Atom) → (b ∉ psupp) → (c ∉ psupp) →
                    (≈ₐP P) ((ActP P) [(b , c)] pel) pel
```

Now, since all the elements of $PN(P)$ are finitely supported, if we can show that $PN(P)$ is an equivariant subset of P, then $PN(P)$ is a nominal set. For any $p \in PN(P)$ and $\pi \in \text{Perm}$, let $S(\pi \cdot p) = S(p) \cup (\text{flatten } \pi)$. So, for any $b, c \in \mathbb{A}$, if $b, c \notin S(\pi \cdot p)$, then we have,

$$\begin{aligned}
&(b\ \ c) \cdot (\pi \cdot p) \\
&= \pi \cdot (b\ \ c) \cdot p \quad [\text{By corollary (2.26)}] \\
&= \pi \cdot p \quad [\because b, c \notin S(p)]
\end{aligned}$$

This shows that (PNO) is an equivariant condition. Now, for any equivariant function $f \in \mathbf{Perm}\ (P, Q)$, we can define $PN(f) = \lambda p \to fp$. We can check that for a finitely supported $p \in P$, fp is also finitely supported with $S(fp) = S(p)$ since for $b, c \in \mathbb{A}$, if $b, c \notin S(fp)$, we have,

$$\begin{aligned}
&(b\ \ c) \cdot (fp) \\
&= f((b\ \ c) \cdot p) \quad [\text{By equivariance of } f] \\
&= fp \quad [\because b, c \notin S(p)]
\end{aligned}$$

Hence, $PN(f)$ is a well-defined function. The equivariance of $PN(f)$ follow from the equivariance of f. Axioms (FUN1), (FUN2) and (FUN3) are obeyed by definition. So PN is a functor. Now we show that $I \dashv PN$.

For any $P \in \textbf{Perm}$, ϵ_P is defined by subset inclusion. Now, for any nominal set N and Perm-Set P, given an equivariant function $f \in \textbf{Perm}\ (N, P)$, we can define $\bar{f}(N, PN(P)) = \lambda n \to fn$. We can check, as in the previous derivation, that fn is finitely supported with $S(fn) = S(n)$. The equivariance of $\hat{f}$ follow from the equivariance of f. The existence and uniqueness conditions follow by definition.

Since the inclusion functor I has a right adjoint, **Nom** is a co-reflective subcategory of **Perm**.

3.10 Powerset

For any Perm-Set P, the powerset is given by $\text{Pow}_{\textbf{Perm}} P \in \text{Obj}\ \textbf{Perm}\ = \{S \mid S \subseteq P\}$. Thus, $\text{Pow}_{\textbf{Perm}} P \cong \textbf{Set}\ (P, \mathbb{B})$. The nominal powerset of any nominal set N is given by $\text{Pow}_{\textbf{Nom}}\ N \in \textbf{Nom}\ \triangleq PN(\text{Pow}_{\textbf{Perm}} N)$. Now, $\text{Hom}_{\textbf{Nom}_{\textbf{fs}}}\ (N, \mathbb{B}) \cong PN(\textbf{Set}\ (N, \mathbb{B}))$. Hence, $\text{Hom}_{\textbf{Nom}_{\textbf{fs}}}\ (N, \mathbb{B}) \cong \text{Pow}_{\textbf{Nom}}\ N$.

This completes our category theoretic characterization of **Nom** and $\textbf{Nom}_{\textbf{fs}}$. The underlying theme of this chapter was the notion of support for Perm-Sets which make them nominal sets. The next chapter focusses on another important notion of nominal set theory, freshness.

Chapter 4

Freshness

4.1 Definition

Freshness is the complementary notion of support. It gives an idea of name independence, i.e. names on which an element of a given nominal set does not depend. So we can say that the atomic names which do not belong to the support set of an element of a nominal set are fresh for that element. In fact, in the classical theory of nominal sets, '$a \in \mathbb{A}$ is fresh for $x \in X \in \text{Obj } \mathbf{Nom}$' (written $a\#x$) is equivalent to $a \notin \text{supp}_X x$. But since here we are working with some support instead of the smallest support, we cannot use this classical definition, since we shall later need # to satisfy the equivariance property (EQC), but we cannot guarantee that, for any arbitrary $\pi \in \text{Perm}$, $a \notin S_X(x) \Rightarrow \pi \cdot a \notin S_X(\pi \cdot x)$, though it is true that $a \notin \text{supp}_X(x) \Rightarrow \pi \cdot a \notin \text{supp}_X(\pi \cdot x)$. So, we define the criterion of being a fresh atomic name as:

$$\begin{array}{l}\text{Given } X \in \text{Obj } \mathbf{Nom} \text{ and } x \in X\text{, an atomic name } a \in \mathbb{A} \text{ is fresh} \\ \text{for } x \text{ if } \exists \textit{ some } b \in \mathbb{A} \text{ such that } b \notin S_X(x) \text{ and } (a\ \ b) \cdot x = x.\end{array} \quad \text{(FRESH)}$$

We have the following record type defining #:

Definition 4.1

```
record # {X : Nominal}(a : Atom)(x : A_s X) : Set where
  constructor fresh
  field
    new : Atom
    notin : new ∉ (some_supp X) x
    fixed : (≈_a X) ((Act X) [ ( a , new ) ] x) x
```

We note that according to our above definition, given x in nominal set X, if for some $b \in \mathbb{A}$, $b \notin S_X(x)$, then $b\#x$, since $b \notin S_X(x)$ and $(b\ \ b) \cdot x = x$. But the converse may not be true. If $b\#x$, we cannot show that $b \notin S_X(x)$, because $S_X(x)$ is not unique and so given an $S_X(x)$ satisfying $b \notin S_X(x)$, we can give a new support $S_X(x) = S_X(x) \cup \{b\}$.

So, to find some $a \in \mathbb{A}$ which satisfies $a\#x$, we can simply find any $a \notin S_X(x)$. This we can do as in section (1.10). Finding fresh atomic names for elements of nominal sets is then, not difficult in general, but in case of nominal sets which are exponential objects, we need to be a bit careful, since the freshness condition of a finitely supported function cannot be reduced to 'freshness properties for the domain and codomain of the function' (Example 3.5, [6]). In section (4.3), we shall consider how the freshness of a finitely supported function is related to the freshness properties for the domain and codomain. But now let us look at an important theorem which allows us to replace *some* in (FRESH) by co-finitely many.

4.2 Some/any theorem

Theorem 4.2 For any nominal set X and any $x \in X$, given atoms $a, c \in \mathbb{A}$ such that $a\#x$ and $c \notin S_X(x)$, we have $(a\ \ c) \cdot x = x$.

Proof. Since $a\#x$, there exists some $b \in \mathbb{A}$, such that $b \notin S_X(x)$ and $(a\ \ b) \cdot x = x$. If $c = a$ or $c = b$, then the theorem follows immediately. So let us assume $c \neq a$ and $c \neq b$. Then, we have,

$$
\begin{aligned}
& (a\ \ c) \cdot x \\
&= (a\ \ b) \cdot (b\ \ c) \cdot (a\ \ b) \cdot x \quad \text{[By corollary (2.28)]} \\
&= (a\ \ b) \cdot (b\ \ c) \cdot x \quad \text{[Given]} \\
&= (a\ \ b) \cdot x \quad [\because b, c \notin S_X(x)] \\
&= x \quad \text{[Given]}
\end{aligned}
$$

□

This theorem allows us to change from some fresh name to any fresh name so that if we can prove any property for some fresh name, we know that the property holds for any fresh name. So to check that a certain property holds for co-finitely many fresh names, we just need to check that the property holds for some fresh name. This freedom to move to and fro between some and any makes working with nominal sets a lot easier

and this also reflects our informal reasoning about names. Now we look at the notion of freshness in case of finitely supported functions.

4.3 Freshness in finitely supported functions

Lemma 4.3 Suppose X and Y are nominal sets and $f \in \mathbf{Nom_{fs}}\ (X, Y)$. Then, for any $a \in \mathbb{A}$ and $x \in X$, if $a \# f$ and $a \# x$, then $a \# fx$.

Proof. Let us choose an atom $b \in \mathbb{A}$ such that $b \notin S_{X \to Y}(f) \cup S_X(x) \cup S_Y(fx)$. Then, by theorem (4.2), we have,

$$\begin{aligned} ((a\ \ b) \cdot f)x &= fx \\ \text{or, } (a\ \ b) \cdot f((a\ \ b) \cdot x) &= fx \quad \text{[By (AFS)]} \\ \text{or, } (a\ \ b) \cdot (fx) &= fx \quad [\because a \# x \wedge b \notin S(x)] \end{aligned}$$

Now, since $b \notin S(fx)$ and $(a\ \ b) \cdot (fx) = fx$, so $a \# fx$. □

Corollary 4.4 Suppose X and Y are nominal sets and $f \in \mathbf{Nom}\ (X, Y)$. Then, for any $a \in \mathbb{A}$ and $x \in X$, if $a \# x$, then $a \# fx$.

Proof. Lemma (4.3) along with the fact that equivariant functions are emptily supported functions. □

Lemma (4.3) says that if an atom is fresh for a finitely supported function and some element in the domain of the function, then that atom is also fresh for the image of that element under the given function. Now, what happens when a given finitely supported function from $\mathbb{A}$ to some nominal set X satisfies the property that if some atom is fresh for the function, then that atom is also fresh for the image of the atom under the given function. We show that for such a function, all atoms that are fresh for the function map to the same element in the codomain of the function.

Theorem 4.5 For any nominal set X and $f \in \mathbf{Nom_{fs}}\ (\mathbb{A}, X)$, if

$$\forall a \in \mathbb{A}, \quad a \# f \Rightarrow a \# fa \tag{FCON}$$

holds, then for atoms $b, c \in \mathbb{A}$, if $b, c \# f$, then $fb = fc$.

Proof. Let us choose an atom $d \in \mathbb{A}$ such that $d \notin S_{\mathbb{A}\to X}(f) \cup S_X(fb) \cup S_X(fc)$. Then, we have, by theorem (4.2):

$$\begin{aligned} &((b\ \ d) \cdot f)b = fb \\ \text{or, } &(b\ \ d) \cdot f((b\ \ d) \cdot b) = fb \quad \text{[By (AFS)]} \\ \text{or, } &(b\ \ d) \cdot fd = fb \\ \text{or, } &fd = (b\ \ d) \cdot (fb) \\ \text{or, } &fd = fb \quad [\because d \notin S(fb) \wedge b\#fb] \end{aligned}$$

Similarly, we can show that $fd = fc$. The theorem follows. □

4.4 Separated product functor

Given any nominal set X, we can form a nominal set $X * \mathbb{A} \subseteq X \times \mathbb{A}$ such that for any $(x, a) \in X * \mathbb{A}$, we have $a\#x$. This subset of the cartesian product of X and $\mathbb{A}$ is called the separated product of X and $\mathbb{A}$. We need to check that $X * \mathbb{A}$ is indeed a nominal set. For this, we just need to show that the freshness relation is equivariant.

Lemma 4.6 Given nominal set X and $x \in X$, if for any atom $a \in \mathbb{A}$, $a\#x$, then for any permutation $\pi \in \text{Perm}$, $\pi \cdot a\#\pi \cdot x$.

Proof. Let us choose an atom $d \in \mathbb{A}$ such that $d \notin (\text{flatten } \pi) \cup S_X(x) \cup S_X(\pi \cdot x)$. Then we have,

$$\begin{aligned} &(\pi \cdot a\ \ d) \cdot (\pi \cdot x) \\ &= \pi \cdot (a\ \ d) \cdot x \quad \text{[By corollary (2.25)]} \\ &= \pi \cdot x \quad [\because d \notin S(x) \wedge a\#x] \end{aligned}$$

Now, since $d \notin S(\pi \cdot x)$ and $(\pi \cdot a\ \ d) \cdot (\pi \cdot x) = \pi \cdot x$, so $\pi \cdot a\#\pi \cdot x$. □

So, for every nominal set X, $X * \mathbb{A}$ is also a nominal set. In fact, we can show that $_ * \mathbb{A}$ is a functor from **Nom** to **Nom** .

Given nominal sets X and Y and an equivariant function $f \in$ **Nom** (X, Y), we can give an equivariant function $f' \in$ **Nom** $(X * \mathbb{A}, Y * \mathbb{A})$ defined as $f' = \lambda(x, a) \to (fx, a)$. First we need to check that the function is well-defined, i.e. given $a \in \mathbb{A}$ and $x \in X$, if

$a\#x$, then $a\#fx$. To show this, let us choose some $d \in \mathbb{A}$ such that $d \notin S_X(x) \cup S_Y(fx)$. Then we have,

$$\begin{aligned} & (a\ \ d) \cdot (fx) \\ = & f((a\ \ d) \cdot x) \quad \text{[By equivariance of } f] \\ = & fx \quad [\because d \notin S(x) \wedge a\#x] \end{aligned}$$

Since, $d \notin S(fx)$ and $(a\ \ d) \cdot (fx) = fx$, so $a\#fx$. This shows that f' is a well-defined function. The equivariance of f' follow from the equivariance of f. The functor axioms ((FUN1), (FUN2) and (FUN3)) follow due to definition. Hence, $_ * \mathbb{A}$ is a functor.

In the next chapter, we show that this functor has a right adjoint, which is the name abstraction functor. Chapter 2 dealt with support, Chapter 3 with freshness. We are now left with one other key idea of nominal set theory, name abstraction.

Chapter 5

Name Abstraction

What does it mean for a name to be bound in a term? An informal answer might be, if the bound name is replaced by a name that is fresh for the term, then the new term is equivalent to the original term. Name abstraction sets formalize this notion of alpha-equivalence in terms of freshness.

5.1 Alpha-equivalence

Given any nominal set X, let us define the binary relation $\approx_\alpha$ on $\mathbb{A} \times X$ as:

$$(a_1, x_1) \approx_\alpha (a_2, x_2) \triangleq \exists a \in \mathbb{A} \text{ such that } a \# x_1 \wedge a \# x_2 \wedge (a \;\; a_1) \cdot x_1 = (a \;\; a_2) \cdot x_2 \tag{ALPHA}$$

So we have the following record type for $\approx_\alpha$:

Definition 5.1

```
record NewName {X : Nominal}(a₁x₁ : Atom × (Aₛ X))
               (a₂x₂ : Atom × (Aₛ X)) : Set where
 constructor NName
 field
  nn : Atom
  nn∉x₁ : nn ∉ (some_supp X) (proj₂ a₁x₁)
  nn∉x₂ : nn ∉ (some_supp X) (proj₂ a₂x₂)
  nna₁≈nna₂ : (≈ₐ X) ((Act X) [(nn , (proj₁ a₁x₁))] (proj₂ a₁x₁))
                    ((Act X) [(nn , (proj₁ a₂x₂))] (proj₂ a₂x₂))
```

In line with some/any theorem (4.2), we have the following for (ALPHA):

Lemma 5.2

$$(a_1, x_1) \approx_\alpha (a_2, x_2) \Rightarrow (\forall c \in \mathbb{A}, c\#x_1 \wedge c\#x_2 \Rightarrow (c\ \ a_1) \cdot x_1 = (c\ \ a_2) \cdot x_2) \qquad \text{(ALPHA')}$$

Proof. Since $(a_1, x_1) \approx_\alpha (a_2, x_2)$, by (ALPHA), there exists some $b \in \mathbb{A}$ such that:

$$b\#x_1 \wedge b\#x_2 \wedge (b\ \ a_1) \cdot x_1 = (b\ \ a_2) \cdot x_2 \qquad \text{(ThC1)}$$

Now we check the following cases:

- $b = c$
  Theorem follows immediately.
- $b = a_1 \wedge c = a_2 \wedge b \neq a_2 \wedge c \neq a_1$
  By (ThC1), $x_1 = (a_1\ \ a_2) \cdot x_2$ and hence $(a_2\ \ a_1) \cdot x_1 = x_2$.
- $b = a_2 \wedge c = a_1 \wedge b \neq a_1 \wedge c \neq a_2$
  Similar to the previous case.
- $b \neq a_1 \wedge c \neq a_1 \wedge b \neq a_2 \wedge c \neq a_2$
  We have,

$$\begin{aligned}
&(b\ \ a_1) \cdot x_1 = (b\ \ a_2) \cdot x_2 \\
\text{or, } &(c\ \ b) \cdot (b\ \ a_1) \cdot x_1 = (c\ \ b) \cdot (b\ \ a_2) \cdot x_2 \\
\text{or, } &(c\ \ b) \cdot (b\ \ a_1) \cdot (c\ \ b) \cdot x_1 = (c\ \ b) \cdot (b\ \ a_2) \cdot (c\ \ b) \cdot x_2 \quad [\because b, c\#x_1 \wedge b, c\#x_2] \\
\text{or, } &(c\ \ a_1) \cdot x_1 = (c\ \ a_2) \cdot x_2 \quad [\text{By corollary (2.28)}]
\end{aligned}$$

The lemma follows. □

Now, we prove that $\approx_\alpha$ is an equivalence relation. Reflexivity and symmetricity are easy to check. We prove the transitivity of $\approx_\alpha$.

Lemma 5.3 $\approx_\alpha$ is transitive.

Proof. Let $(a_1, x_1) \approx_\alpha (a_2, x_2)$ and $(a_2, x_2) \approx_\alpha (a_3, x_3)$. Let us choose $c \in \mathbb{A}$ such that $c \notin S_X(x_1) \cup S_X(x_2) \cup S_X(x_3)$. Therefore by lemma (5.2), $(c\ \ a_1) \cdot x_1 = (c\ \ a_2) \cdot x_2$ and $(c\ \ a_2) \cdot x_2 = (c\ \ a_3) \cdot x_3$. So $c\#x_1$ and $c\#x_3$ and $(c\ \ a_1) \cdot x_1 = (c\ \ a_3) \cdot x_3$. Hence $(a_1, x_1) \approx_\alpha (a_3, x_3)$. □

We now prove a few lemmas on $\approx_\alpha$.

Lemma 5.4 Given $(a_1, x_1) \approx_\alpha (a_2, x_2)$, if $a_1 = a_2$ then $x_1 = x_2$.

Proof. Since $(a_1, x_1) \approx_\alpha (a_2, x_2)$, so for some $b \in \mathbb{A}$, we have $(b \;\; a_1) \cdot x_1 = (b \;\; a_2) \cdot x_2$. But since $a_1 = a_2$, so $x_1 = x_2$. □

Lemma 5.5 Given $(a_1, x_1) \approx_\alpha (a_2, x_2)$, if $a_1 \# x_1$ and $a_2 \# x_2$, then $x_1 = x_2$.

Proof. Since $(a_1, x_1) \approx_\alpha (a_2, x_2)$, so for some $b \in \mathbb{A}$, we have $b \# x_1$ and $b \# x_2$ and $(b \;\; a_1) \cdot x_1 = (b \;\; a_2) \cdot x_2$. Now since $b, a_1 \# x_1$ and $b, a_2 \# x_2$, therefore $x_1 = x_2$. □

Lemma 5.6 Given any nominal set X and $x \in X$, for atoms $a, a' \in \mathbb{A}$, if $a' \# x$, then $(a, x) \approx_\alpha (a', (a' \;\; a) \cdot x)$.

Proof. Let us choose some $b \in \mathbb{A}$ such that $b \notin S_X(x) \cup S_X((a' \;\; a) \cdot x)$. Then we have,

$$\begin{aligned} & (b \;\; a') \cdot ((a' \;\; a) \cdot x) \\ = \; & (b \;\; a') \cdot (a' \;\; a) \cdot (b \;\; a') \cdot x \quad [\because b \notin S(x) \wedge a' \# x] \\ = \; & (b \;\; a) \cdot x \quad \text{[By corollary (2.28)]} \end{aligned}$$

Now, $b \# x$ and $b \# (a' \;\; a) \cdot x$ and $(b \;\; a) \cdot x = (b \;\; a') \cdot ((a' \;\; a) \cdot x)$. The lemma follows. □

Given any term, we can change to any other member of the alpha-equivalence class of the term using the above lemma. Now we can define name abstraction sets.

5.2 Name abstraction sets

Given a nominal set X, the set $\mathbb{A} \times X$ quotiented by $\approx_\alpha$ is called the set of name abstractions of elements of X. It is denoted by $[\mathbb{A}]X$. We now prove that $[\mathbb{A}]X$ is indeed a nominal set. Since we are working with setoids, we declare the underlying set of $[\mathbb{A}]X$ to be $A \times X$ and the underlying relation to be $\approx_\alpha$. The associativity and identity group action axioms ((GA1) and (GA2)) follow from their corresponding counterparts for $\mathbb{A}$ and X. Now we need to verify axiom (GA3), that equivalent permutations act on equivalent elements of $A \times X$ to give equivalent elements.

Lemma 5.7 Given $\pi_1, \pi_2 \in \text{Perm}$ and $(a_1, x_1), (a_2, x_2) \in \mathbb{A} \times X$, if $\pi_1 \approx \pi_2$ and $(a_1, x_1) \approx_\alpha (a_2, x_2)$, then $(\pi_1 \cdot a_1, \pi_1 \cdot x_1) \approx_\alpha (\pi_2 \cdot a_2, \pi_2 \cdot x_2)$.

Proof. Let us choose $b \in \mathbb{A}$ such that $b \notin (\text{flatten } \pi_1) \cup (\text{flatten } \pi_2) \cup S_X(x_1) \cup S_X(x_2) \cup S_X(\pi_1 \cdot x_1) \cup S_X(\pi_2 \cdot x_2)$. So, by lemma (5.2),

$$(b\ \ a_1) \cdot x_1 = (b\ \ a_2) \cdot x_2$$

Now, we have,

$$\begin{aligned}
&(b\ \ \pi_1 \cdot a_1) \cdot (\pi_1 \cdot x_1) \\
&= \pi_1 \cdot (b\ \ a_1) \cdot x_1 \quad \text{[By corollary (2.25)]} \\
&= \pi_1 \cdot (b\ \ a_2) \cdot x_2 \\
&= \pi_2 \cdot (b\ \ a_2) \cdot x_2 \quad [\because \pi_1 \approx \pi_2] \\
&= (b\ \ \pi_2 \cdot a_2) \cdot (\pi_2 \cdot x_2) \quad \text{[By corollary (2.25)]}
\end{aligned}$$

This along with the fact that $b \# \pi_1 \cdot x_1$ and $b \# \pi_2 \cdot x_2$ shows that $(\pi_1 \cdot a_1, \pi_1 \cdot x_1) \approx_\alpha (\pi_2 \cdot a_2, \pi_2 \cdot x_2)$. □

Once (GA1), (GA2) and (GA3) are satisfied, we know that $[\mathbb{A}]X$ is a PermSet. Now we give the support for any $(a, x) \in [\mathbb{A}]X$ as $S_{[\mathbb{A}]X}(a, x) = S_X(x)$. Next we show that this indeed is a support.

Lemma 5.8 Given a nominal set X and $x \in X$, for atoms $a, b, c \in \mathbb{A}$, if $b, c \notin S_{[\mathbb{A}]X}(a, x)$, then $(b\ \ c) \cdot (a, x) \approx_\alpha (a, x)$.

Proof. If $b = a$ or $c = a$, then $a \# x$ and $(b\ \ c) \cdot a \# x$. The lemma then follows by lemma (5.5). So, let us assume that $b \neq a$ and $c \neq a$. Then, since $b, c \notin S_X(x)$, so $(b\ \ c) \cdot (a, x) = ((b\ \ c) \cdot a, (b\ \ c) \cdot x) = (a, x)$. The lemma follows. □

This shows that $[\mathbb{A}]X$ is a nominal set. In fact we can define a functor $[\mathbb{A}]_ : \mathbf{Nom} \to \mathbf{Nom}$, the name abstraction functor.

5.3 Name abstraction functor

Given nominal sets X and Y and an equivariant function $f \in \mathbf{Nom}\ (X, Y)$, we have $f' \in \mathbf{Nom}\ ([\mathbb{A}]X, [\mathbb{A}]Y)$ defined as $f' = \lambda(a, x) \to (a, fx)$. First we need to show that this respects $\approx_\alpha$ (EqP2).

Lemma 5.9 Given $f \in \mathbf{Nom}\ (X, Y)$ and $(a_1, x_1) \approx_\alpha (a_2, x_2) \in [\mathbb{A}]X$, $(a_1, fx_1) \approx_\alpha (a_2, fx_2)$.

Proof. Let us choose $b \notin S_X(x_1) \cup S_X(x_2) \cup S_Y(fx_1) \cup S_Y(fx_2)$. Then, by lemma (5.2),

$$(b\ \ a_1) \cdot x_1 = (b\ \ a_2) \cdot x_2$$

Now, we have,

$$\begin{aligned} & (b\ \ a_1)\cdot(fx_1) \\ &= f((b\ \ a_1)\cdot x_1) \quad \text{[By equivariance of } f\text{]} \\ &= f((b\ \ a_2)\cdot x_2) \\ &= (b\ \ a_2)\cdot(fx_2) \quad \text{[By equivariance of } f\text{]} \end{aligned}$$

This along with the fact that $b\#fx_1$ and $b\#fx_2$ shows that $(a_1, fx_1) \approx_\alpha (a_2, fx_2)$. □

Once we know that f' respects $\approx_\alpha$, we just need to check that it's an equivariant function. But this follows from the equivariance of f. This defines the functor $[\mathbb{A}]_$ action on morphisms. The functor axioms ((FUN1), (FUN2) and (FUN3)) follow by definition. Hence $[\mathbb{A}]_$ is a functor.

5.4 Properties of name abstraction functor

The name abstraction functor is very well-behaved, it preserves limits and colimits in **Nom**. Below we check that $[\mathbb{A}]_$ preserves products and coproducts.

Lemma 5.10 For nominal sets X_1 and X_2, $[\mathbb{A}](X_1 \times X_2) \cong ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$.

Proof. We need to give equivariant functions f and g respectively from $[\mathbb{A}](X_1 \times X_2)$ to $([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$ and vice-versa such that $g \circ f$ and $f \circ g$ are identity morphisms on $[\mathbb{A}](X_1 \times X_2)$ and $([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$ respectively.

- $f : [\mathbb{A}](X_1 \times X_2) \to ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$
  We define f as $f = \lambda(a, (x_1, x_2)) \to ((a, x_1), (a, x_2))$. We need to check that this definition respects $\approx_\alpha$ (EqP2). For this, let us consider $(a, (x_1, x_2)), (a', (x_1', x_2')) \in [\mathbb{A}](X_1 \times X_2)$ such that $(a, (x_1, x_2)) \approx_\alpha (a', (x_1', x_2'))$. Now we need to show that $((a, x_1), (a, x_2)) \approx_\alpha ((a', x_1'), (a', x_2'))$.

  Let us choose $b \in \mathbb{A}$ such that $b \notin S_{X_1}(x_1) \cup S_{X_2}(x_2) \cup S_{X_1}(x_1') \cup S_{X_2}(x_2')$. Then, we have,

$$\begin{aligned} & (b\ \ a)\cdot(x_1, x_2) = (b\ \ a')\cdot(x_1', x_2') \\ \text{or, } & (b\ \ a)\cdot x_1 = (b\ \ a')\cdot x_1' \wedge (b\ \ a)\cdot x_2 = (b\ \ a')\cdot x_2' \\ \text{or, } & (a, x_1) \approx_\alpha (a', x_1') \wedge (a, x_2) \approx_\alpha (a', x_2') \\ \text{or, } & ((a, x_1), (a, x_2)) \approx_\alpha ((a', x_1'), (a', x_2')) \end{aligned}$$

Once we have shown that f respects $\approx_\alpha$, we need to check it is an equivariant function. But that follows from the definition of f.

- $g : ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2) \to [\mathbb{A}](X_1 \times X_2)$
Given $((a_1, x_1), (a_2, x_2)) \in ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$, we define $g((a_1, x_1), (a_2, x_2))$ in the following way:

Let us choose some $b \in \mathbb{A}$ such that $b \notin \{a_1\} \cup \{a_2\} \cup S_{X_1}(x_1) \cup S_{X_2}(x_2)$. Then
$$g((a_1, x_1), (a_2, x_2)) = (b, ((b\ \ a_1) \cdot x_1, (b\ \ a_2) \cdot x_2)).$$

We need to check that this definition of g respects $\approx_\alpha$ (EqP2). For this, let us consider $((a_1, x_1), (a_2, x_2)), ((a_1', x_1'), (a_2', x_2')) \in ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$ such that $((a_1, x_1), (a_2, x_2)) \approx_\alpha ((a_1', x_1'), (a_2', x_2'))$. We need to show that,

$$(b, ((b\ \ a_1) \cdot x_1, (b\ \ a_2) \cdot x_2)) \approx_\alpha (b', ((b'\ \ a_1') \cdot x_1', (b'\ \ a_2') \cdot x_2'))$$

for some $b \notin \{a_1\} \cup \{a_2\} \cup S_{X_1}(x_1) \cup S_{X_2}(x_2)$ and $b' \notin \{a_1'\} \cup \{a_2'\} \cup S_{X_1}(x_1') \cup S_{X_2}(x_2')$. For this, let us choose $c \in \mathbb{A}$ such that $c \notin \{a_1, a_2, a_1', a_2'\} \cup S_{X_1}(x_1) \cup S_{X_2}(x_2) \cup S_{X_1}(x_1') \cup S_{X_2}(x_2') \cup S_{X_1}((b\ \ a_1) \cdot x_1) \cup S_{X_2}((b\ \ a_2) \cdot x_2) \cup S_{X_1}((b'\ \ a_1') \cdot x_1') \cup S_{X_2}((b'\ \ a_2') \cdot x_2')$. Then we have,

$$\begin{aligned}
&(c\ \ b) \cdot ((b\ \ a_1) \cdot x_1) \\
&= (c\ \ b) \cdot (b\ \ a_1) \cdot (c\ \ b) \cdot x_1 \quad [\because c, b \notin S(x_1)] \\
&= (c\ \ a_1) \cdot x_1 \quad [\text{By corollary (2.28)}] \\
&= (c\ \ a_1') \cdot x_1' \quad [\because (a_1, x_1) \approx_\alpha (a_1', x_1')] \\
&= (c\ \ b') \cdot (b'\ \ a_1') \cdot (c\ \ b') \cdot x_1' \quad [\text{By corollary (2.28)}] \\
&= (c\ \ b') \cdot ((b'\ \ a_1') \cdot x_1') \quad [\because c, b' \notin S(x_1')]
\end{aligned}$$

Similarly, we can show that $(c\ \ b) \cdot ((b\ \ a_2) \cdot x_2) = (c\ \ b') \cdot ((b'\ \ a_2') \cdot x_2')$. So, $(c\ \ b) \cdot ((b\ \ a_1) \cdot x_1, (b\ \ a_2) \cdot x_2) = (c\ \ b') \cdot ((b'\ \ a_1') \cdot x_1', (b'\ \ a_2') \cdot x_2'))$. Hence, g respects $\approx_\alpha$. Now, we check that g is an equivariant function. For this, let us suppose $((a_1, x_1), (a_2, x_2)) \in ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$ and $\pi \in \text{Perm}$. Then, we need to show that,

$$(\pi \cdot b, (\pi \cdot (b\ \ a_1) \cdot x_1, \pi \cdot (b\ \ a_2) \cdot x_2)) \approx_\alpha (b_\pi, ((b_\pi\ \ \pi \cdot a_1) \cdot \pi \cdot x_1, (b_\pi\ \ \pi \cdot a_2) \cdot \pi \cdot x_2))$$

for some $b \notin \{a_1\} \cup \{a_2\} \cup S_{X_1}(x_1) \cup S_{X_2}(x_2)$ and $b_\pi \notin \{\pi \cdot a_1\} \cup \{\pi \cdot a_2\} \cup S_{X_1}(\pi \cdot x_1) \cup S_{X_2}(\pi \cdot x_2)$. To this end, let us choose $c \in \mathbb{A}$ such that $c \notin \{a_1, a_2, \pi \cdot a_1, \pi \cdot a_2\} \cup (\text{flatten } \pi) \cup S_{X_1}(x_1) \cup S_{X_2}(x_2) \cup S_{X_1}(\pi \cdot x_1) \cup S_{X_2}(\pi \cdot x_2) \cup S_{X_1}(\pi \cdot (b\ \ a_1) \cdot$

$x_1) \cup S_{X_2}(\pi\cdot(b\ \ a_2)\cdot x_2) \cup S_{X_1}((b_\pi\ \ \pi\cdot a_1)\cdot\pi\cdot x_1) \cup S_{X_2}((b_\pi\ \ \pi\cdot a_2)\cdot\pi\cdot x_2)$. Then, we have,

$$\begin{aligned}
&(c\ \ \pi\cdot b)\cdot\pi\cdot(b\ \ a_1)\cdot x_1\\
&= \pi\cdot(c\ \ b)\cdot(b\ \ a_1)\cdot x_1 \quad \text{[By corollary(2.25)]}\\
&= \pi\cdot(c\ \ b)\cdot(b\ \ a_1)\cdot(c\ \ b)\cdot x_1 \quad [\because c, b \notin S(x_1)]\\
&= \pi\cdot(c\ \ a_1)\cdot x_1 \quad \text{[By corollary (2.28)]}\\
&= (c\ \ \pi\cdot a_1)\cdot\pi x_1 \quad \text{[By corollary (2.25)]}\\
&= (c\ \ b_\pi)\cdot(b_\pi\ \ \pi\cdot a_1)\cdot(c\ \ b_\pi)\cdot\pi\cdot x_1 \quad \text{[By corollary (2.28)]}\\
&= (c\ \ b_\pi)\cdot(b_\pi\ \ \pi\cdot a_1)\cdot\pi\cdot x_1 \quad [\because c, b_\pi \notin S(\pi\cdot x_1)]
\end{aligned}$$

Similarly, we can show that $(c\ \ \pi\cdot b)\cdot\pi\cdot(b\ \ a_2)\cdot x_2 = (c\ \ b_\pi)\cdot(b_\pi\ \ \pi\cdot a_2)\cdot\pi\cdot x_2$. Hence, $(c\ \ \pi\cdot b)\cdot(\pi\cdot(b\ \ a_1)\cdot x_1, \pi\cdot(b\ \ a_2)\cdot x_2) = (c\ \ b_\pi)\cdot((b_\pi\ \ \pi\cdot a_1)\cdot\pi\cdot x_1, (b_\pi\ \ \pi\cdot a_2)\cdot\pi\cdot x_2)$. This shows that g is an equivariant function.

- $g \circ f = \text{id}$
  Suppose $(a, (x_1, x_2)) \in [\mathbb{A}](X_1 \times X_2)$. We have $(g \circ f)(a, (x_1, x_2)) = (b, ((b\ \ a_1)\cdot x_1, (b\ \ a_2)\cdot x_2))$ for some $b \in \mathbb{A}$ such that $b \notin \{a\} \cup S_{X_1}(x_1) \cup S_{X_2}(x_2)$. By lemma (5.6), $(a, (x_1, x_2)) \approx_\alpha (b, ((b\ \ a_1)\cdot x_1, (b\ \ a_2)\cdot x_2))$.

- $f \circ g = \text{id}$
  Suppose $((a_1, x_1), (a_2, x_2)) \in ([\mathbb{A}]X_1)\times([\mathbb{A}]X_2)$. We have $(f\circ g)((a_1, x_1), (a_2, x_2)) = ((b, (b\ \ a_1)\cdot x_1), (b, (b\ \ a_2)\cdot x_2))$ for some $b \in \mathbb{A}$ such that $b \notin \{a_1\}\cup\{a_2\}\cup S_{X_1}(x_1)\cup S_{X_2}(x_2)$. By lemma (5.6), $((a_1, x_1), (a_2, x_2)) \approx_\alpha ((b, (b\ \ a_1)\cdot x_1), (b, (b\ \ a_2)\cdot x_2))$.

The lemma follows. □

Lemma 5.11 For nominal sets X_1 and X_2, $[\mathbb{A}](X_1 + X_2) \cong ([\mathbb{A}]X_1) + ([\mathbb{A}]X_2)$.

Proof. We can give equivariant functions f and g respectively from $[\mathbb{A}](X_1 + X_2)$ to $([\mathbb{A}]X_1) + ([\mathbb{A}]X_2)$ and vice-versa as:

$$\begin{aligned}
f(a, w) &= \text{match } w \text{ with}\\
&\quad \mid \text{inl } x_1 \to \text{ inl } (a, x_1)\\
&\quad \mid \text{inr } x_2 \to \text{ inr } (a, x_2)\\
gv &= \text{match } v \text{ with}\\
&\quad \mid \text{inl } (a, x_1) \to (a, \text{ inl } x_1)\\
&\quad \mid \text{inr } (a, x_2) \to (a, \text{ inr } x_2)
\end{aligned}$$

We can easily check that both f and g satisfy axioms (EqP1) and (EqP2) and that $g \circ f =$ id and $f \circ g =$ id. □

The separated product functor encapsulates the notion of freshness whereas the name abstraction functor the notion of alpha-equivalence. Applying these functors in succession to any nominal set gives us the following lemma.

Lemma 5.12 For any nominal set X, $([\mathbb{A}]X) * \mathbb{A} \cong \mathbb{A} \times X$.

Proof. We can give equivariant functions f and g from $([\mathbb{A}]X)*\mathbb{A}$ to $\mathbb{A}\times X$ and vice-versa respectively as:

$$f((a,x),a') = (a',(a' \ \ a)\cdot x)$$
$$g(a,x) = ((a,x),a)$$

We need to check that f satisfies axiom (EqP2), i.e. given $((a_1,x_1),a'),((a_2,x_2),a') \in ([\mathbb{A}]X) * \mathbb{A}$, if $(a_1,x_1) \approx_\alpha (a_2,x_2)$, then $(a' \ \ a_1)\cdot x_1 = (a' \ \ a_2)\cdot x_2$. But this is true, by lemma (5.2), since $a' \# x_1, x_2$. We now check that f satisfies the equivariance property (EqP1). For this, let $((a,x),a') \in ([\mathbb{A}]X) * \mathbb{A}$ and $\pi \in$ Perm. We need to show that $\pi \cdot ((a' \ \ a)\cdot x) = (\pi\cdot a' \ \ \pi\cdot a)\cdot(\pi\cdot x)$. But this follows from lemma (2.24). Hence f is an equivariant function.

Now, we need to check that g is a well-defined function, i.e. $a\#(a,x) \in [\mathbb{A}]X$. But this is true since for any $b \in \mathbb{A}$, if $b \notin S_X(x)$, then we have,

$$\begin{aligned}
&(b \ \ a)\cdot(a,x)\\
\approx_\alpha\ &(b,(b \ \ a)\cdot x)\\
\approx_\alpha\ &(a,x) \quad \text{[By lemma (5.6), } \because b\#x]
\end{aligned}$$

Once we have shown that g is well-defined, we need to verify axioms (EqP1) and (EqP2). But these follow easily from definition. Hence g is an equivariant function.

Now we show that $g \circ f =$ id. Let $((a,x),a') \in ([\mathbb{A}]X) * \mathbb{A}$. Then, we have,

$$\begin{aligned}
&(g\circ f)((a,x),a')\\
=\ &g(a',(a' \ \ a)\cdot x)\\
=\ &((a',(a' \ \ a)\cdot x),a')
\end{aligned}$$

But by lemma (5.6), $(a', (a'\ \ a) \cdot x) \approx_\alpha (a, x)$ for $a' \# x$. Hence $g \circ f = \text{id}$. Next we show that $f \circ g = \text{id}$. Let $(a, x) \in \mathbb{A} \times X$. Then, we have,

$$\begin{aligned} & (f \circ g)(a, x) \\ = & f((a, x), a) \\ = & (a, (a\ \ a) \cdot x) \\ = & (a, x) \end{aligned}$$

The lemma follows. □

Now we show that the name abstraction functor has a left adjoint, the separated product functor.

Theorem 5.13 $_ * \mathbb{A} \dashv [\mathbb{A}]_$

Proof. First we give the counit of the adjunction. For any nominal set X, let $\epsilon_X \in$ **Nom** $(([\mathbb{A}]X) * \mathbb{A}, X)$ defined as $\epsilon_X = \lambda((a, x), a') \to (a'\ \ a) \cdot x$. First, we check that ϵ_X satisfies axiom (EqP2). Let $((a_1, x_1), a'), ((a_2, x_2), a') \in ([\mathbb{A}])X * \mathbb{A}$ such that $(a_1, x_1) \approx_\alpha (a_2, x_2)$. We need to show that $(a'\ \ a_1) \cdot x_1 = (a'\ \ a_2) \cdot x_2$. But this is true by (5.2), since $a' \# x_1, x_2$. Now we check for equivariance (EqP1). Let $((a, x), a') \in ([\mathbb{A}])X * \mathbb{A}$ and $\pi \in$ Perm. Then, we have,

$$\begin{aligned} & \pi \cdot \epsilon_X((a, x), a') \\ = & \pi \cdot (a'\ \ a) \cdot x \\ = & (\pi \cdot a'\ \ \pi \cdot a) \cdot \pi \cdot x \quad \text{[By lemma (2.24)]} \\ = & \epsilon_X(\pi \cdot ((a, x), a')) \end{aligned}$$

Once we have shown that ϵ_X is an equivariant function, we need to show that it has the required universal property. Let X, Y are nominal sets and $f \in$ **Nom** $(Y * \mathbb{A}, X)$. We define $\bar{f} \in$ **Nom** $(Y, [\mathbb{A}]X)$ in the following way.

For any $y \in Y$, let us choose $b \in \mathbb{A}$ such that $b \# y$. Then $(y, b) \in Y * \mathbb{A}$.
So, $\bar{f}(y) = f(y, b)$ is well-defined.

But we need to check that $\bar{f} \in$ **Nom** $(Y, [\mathbb{A}]X)$. First we verify axiom (EqP2), i.e. given $y_1, y_2 \in Y$ such that $y_1 \approx y_2$ (where $\approx$ is the underlying relation on setoid Y), we show that $(b_1, f(y_1, b_1)) \approx_\alpha (b_2, f(y_2, b_2))$, where $b_1 \# y_1$ and $b_2 \# y_2$. Let us choose $c \in \mathbb{A}$ such

that $c \notin S_Y(y_1) \cup S_Y(y_2) \cup S_X(f(y_1,b_1)) \cup S_X(f(y_2,b_2))$. Then, we have,

$$\begin{aligned}
& (c \;\; b_1) \cdot f(y_1, b_1) \\
&= f((c \;\; b_1) \cdot y_1, (c \;\; b_1) \cdot b_1) \quad \text{[By equivariance of } f] \\
&= f(y_1, c) \quad [\because b_1 \# y_1 \wedge c \notin S(y_1)] \\
&= f(y_2, c) \quad [\because f \text{ respects } \approx] \\
&= f((c \;\; b_2) \cdot y_2, (c \;\; b_2) \cdot b_2) \quad [\because b_2 \# y_2 \wedge c \notin S(y_2)] \\
&= (c \;\; b_2) \cdot f(y_2, b_2) \quad \text{[By equivariance of } f]
\end{aligned}$$

Hence, $(b_1, f(y_1,b_1)) \approx_\alpha (b_2, f(y_2,b_2))$. We now check that $\bar{f}$ satisfies equivariance property (EqP1). Let $y \in Y$ and let $\bar{f}(y) = (b, f(y,b))$ for some $b \in \mathbb{A}$ such that $b \# y$. Also, let $\pi \in \mathrm{Perm}$. Now let $\bar{f}(\pi \cdot y) = (b_\pi, f(\pi \cdot y, b_\pi))$ for some $b_\pi \in \mathbb{A}$ such that $b_\pi \# \pi \cdot y$. We need to show that $(\pi \cdot b, \pi \cdot f(y,b)) \approx_\alpha (b_\pi, f(\pi \cdot y, b_\pi))$. For this, let us choose $c \in \mathbb{A}$ such that $c \notin (\text{flatten } \pi) \cup S_Y(y) \cup S_Y(\pi \cdot y) \cup S_X(f(y,b)) \cup S_X(f(\pi \cdot y, b_\pi))$. Then, we have,

$$\begin{aligned}
& (c \;\; \pi \cdot b) \cdot (\pi \cdot f(y,b)) \\
&= \pi \cdot (c \;\; b) \cdot f(y,b) \quad \text{[By lemma (2.24)]} \\
&= \pi \cdot f((c \;\; b) \cdot y, (c \;\; b) \cdot b) \quad \text{[By equivariance of } f] \\
&= \pi \cdot f(y, c) \quad [\because b \# y \wedge c \notin S(y)] \\
&= f(\pi \cdot y, \pi \cdot c) \quad \text{[By equivariance of } f] \\
&= f(\pi \cdot y, c) \quad \text{[By lemma (2.23)]} \\
&= f((c \;\; b_\pi) \cdot \pi \cdot y, (c \;\; b_\pi) \cdot b_\pi) \quad [\because b_\pi \# \pi \cdot y \wedge c \notin S(\pi \cdot y)] \\
&= (c \;\; b_\pi) \cdot f(\pi \cdot y, b_\pi) \quad \text{[By equivariance of } f]
\end{aligned}$$

This shows that $(\pi \cdot b, \pi \cdot f(y,b)) \approx_\alpha (b_\pi, f(\pi \cdot y, b_\pi))$. Hence $\bar{f} \in \mathbf{Nom}\ (Y, [\mathbb{A}]X)$.

Now we check the existence condition. Let $(y,a) \in Y * \mathbb{A}$. Then, $(\bar{f} * \mathrm{id}_\mathbb{A})(y,a) = ((b, f(y,b)), a)$ for some $b \# y$. This gives,

$$\begin{aligned}
& \epsilon_X((b, f(y,b)), a) \\
&= (a \;\; b) \cdot f(y,b) \quad \text{[By definition]} \\
&= f((a \;\; b) \cdot y, (a \;\; b) \cdot b) \quad \text{[By equivariance of } f] \\
&= f(y, a) \quad [\because a \# y \wedge b \# y]
\end{aligned}$$

This shows that the triangle below commutes.

$$\begin{array}{ccc} Y * \mathbb{A} & & \\ \bar{f} * \mathrm{id}_{\mathbb{A}} \downarrow & \searrow^{f} & \\ ([\mathbb{A}]X) * \mathbb{A} & \xrightarrow{\epsilon_X} & X \end{array}$$

Next we check the uniqueness condition. Suppose $g \in \mathbf{Nom}\ (Y, [\mathbb{A}]X)$ such that $\epsilon_X \circ (g * \mathrm{id}_{\mathbb{A}}) = f$. We need to show that $g = \bar{f}$. Let $y \in Y$. Then, $\bar{f}(y) = (b, f(y, b))$ for some $b\#y$. Let $g(y) = (b', x)$. Now, we have,

$$\begin{aligned} \epsilon_X \circ (g * \mathrm{id}_{\mathbb{A}})(y, b) &= f(y, b) \\ \text{or, } \epsilon_X(gy, b) &= f(y, b) \\ \text{or, } \epsilon_X((b', x), b) &= f(y, b) \\ \text{or, } (b\ \ b') \cdot x &= f(y, b) \end{aligned}$$

Now, since $(gy, b) \in ([\mathbb{A}]X) * \mathbb{A}$, so $b\#gy$ and hence $b\#x$. So, by lemma (5.6), $(b', x) \approx_\alpha (b, (b\ \ b') \cdot x)$. But by the derivation above, $(b, (b\ \ b') \cdot x) \approx_\alpha (b, f(y, b))$. So by lemma (5.3), $(b', x) \approx_\alpha (b, f(y, b))$. Hence, $gy = \bar{f}y$ and thus, by function extensionality, $g = \bar{f}$. □

Now we look at what condition a function needs to satisfy in order for it to respect alpha-equivalence. Functions with domain $[\mathbb{A}]X$ (X being a nominal set), by virtue of construction, respect alpha-equivalence. But given any finitely supported function from $\mathbb{A} \times X$ to Y (where X and Y are nominal sets), we can state, the requirement for it to respect alpha-equivalence, in terms of the freshness properties of the domain and codomain of the function. This requirement is called the freshness condition for binders by Pitts (2006) [32].

5.5 Freshness condition for binders

Theorem 5.14 For nominal sets X and Y and $f \in \mathbf{Nom_{fs}}\ (\mathbb{A} \times X, Y)$ satisfying,

$$\forall a \in \mathbb{A}, \quad a \notin S_{X \to Y}(f) \Rightarrow (\forall x \in X, a \notin S_Y(f(a, x))) \tag{FFS}$$

there exists a unique function $\bar{f} \in \mathbf{Nom_{fs}}\ ([\mathbb{A}]X, Y)$ such that,

$$\forall a \in \mathbb{A}, \quad a \notin S_{X \to Y}(f) \Rightarrow (\forall x \in X, \bar{f}(a, x) = f(a, x)) \tag{EFS}$$

Proof. First, we define $\bar{f}$ in the following way.

Let $(a,x) \in [\mathbb{A}]X$. Let us choose some $b \in \mathbb{A}$ such that $b \notin \{a\} \cup S_{X\to Y}(f) \cup S_X(x)$.
Then $\bar{f}(a,x) = f(b,(b\ \ a)\cdot x)$.

We now prove that $\bar{f}$ is a finitely supported function with $S_{[\mathbb{A}]X\to Y}(\bar{f}) = S_{\mathbb{A}\times X\to Y}(f)$. But before that, we need to prove a lemma.

Lemma 5.15 Given $a_1, a_2 \in \mathbb{A}$ and $x_1, x_2 \in X$ such that $a_1, a_2 \notin S_{\mathbb{A}\times X\to Y}(f)$ and $(a_1,x_1) \approx_\alpha (a_2,x_2)$ holds, then $f(a_1,x_1) = f(a_2,x_2)$.

Proof. Let us choose $a_3 \in \mathbb{A}$ such that $a_3 \notin S_X(x_1)\cup S_X(x_2)\cup S_Y(f(a_1,x_1))\cup S_Y(f(a_2,x_2))\cup S_{\mathbb{A}\times X\to Y}(f)$. Then we have,

$$\begin{aligned}
& f(a_1,x_1)\\
&= (a_3\ \ a_1)\cdot(f(a_1,x_1)) \quad [\because a_3 \notin S(f(a_1,x_1)) \wedge a_1 \notin S(f(a_1,x_1))]\\
&= f((a_3\ \ a_1)\cdot(a_1,x_1)) \quad [\because a_3, a_1 \notin S(f)]\\
&= f(a_3,(a_3\ \ a_1)\cdot x_1)
\end{aligned}$$

Similarly, we can show that $f(a_2,x_2) = f(a_3,(a_3\ \ a_2)\cdot x_2)$. But since $(a_1,x_1) \approx_\alpha (a_2,x_2)$, so $(a_3\ \ a_1)\cdot x_1 = (a_3\ \ a_2)\cdot x_2$. The lemma follows. □

Now we shall verify axiom (FS1). Let $c,d \in \mathbb{A}$ such that $c,d \notin S(\bar{f})$. Let $(a,x) \in [\mathbb{A}]\times X$. Now $\bar{f}(a,x) = f(b,(b\ \ a)\cdot x) = y_1$ (say) for some $b \notin \{a\}\cup S(f)\cup S(x)$. And $\bar{f}((c\ \ d)\cdot a,(c\ \ d)\cdot x) = f(b',(b'\ \ (c\ \ d)\cdot a)\cdot(c\ \ d)\cdot x) = y_2$ (say) for some $b' \notin \{(c\ \ d)\cdot a\}\cup S(f)\cup S((c\ \ d)\cdot x)$. We need to show that $(c\ \ d)\cdot y_2 = y_1$. For this, let us choose $m \in \mathbb{A}$ such that $m \notin \{c,d,(c\ \ d)\cdot a\}\cup S(f)\cup S(y_2)\cup S(x)\cup S((c\ \ d)\cdot x)$. Then, we have,

$$\begin{aligned}
& (c\ \ d)\cdot y_2\\
&= (c\ \ d)\cdot(m\ \ b')\cdot y_2 \quad [\because m, b' \notin S(y_2)]\\
&= (c\ \ d)\cdot f(m\ \ b')\cdot(b',(b'\ \ (c\ \ d)\cdot a)\cdot(c\ \ d)\cdot x) \quad [\because m, b' \notin f]\\
&= (c\ \ d)\cdot f(m,(m\ \ b')\cdot(b'\ \ (c\ \ d)\cdot a)\cdot(c\ \ d)\cdot x)\\
&= (c\ \ d)\cdot f(m,(m\ \ b')\cdot(b'\ \ (c\ \ d)\cdot a)\cdot(m\ \ b')\cdot(c\ \ d)\cdot x)\\
&\qquad [\because m, b' \notin S((c\ \ d)\cdot x)]\\
&= (c\ \ d)\cdot f(m,(m\ \ (c\ \ d)\cdot a)\cdot(c\ \ d)\cdot x) \quad [\text{By lemma (2.28)}]\\
&= (c\ \ d)\cdot f((c\ \ d)\cdot m,(c\ \ d)\cdot(m\ \ a)\cdot x) \quad [\text{By corollary (2.25)}]\\
&= (c\ \ d)\cdot f(c\ \ d)\cdot(m,(m\ \ a)\cdot x)\\
&= f(m,(m\ \ a)\cdot x) \quad [\because c, d \notin S(f)]\\
&= f(b,(b\ \ a)\cdot x) \quad [\text{By lemma (5.15)}]
\end{aligned}$$

To show that $\bar{f}$ is a finitely supported function, we need to verify axiom (FS2) as well. But this follows from lemma (5.15). Now we shall prove the existence and uniqueness conditions. The existence condition follows from lemma (5.15). To prove the uniqueness condition, let us assume $g \in \mathbf{Nom_{fs}}\ ([\mathbb{A}]X, Y)$ satisfies EFS. We need to show that $g = \bar{f}$. Let $(a, x) \in [\mathbb{A}]X$. Let $\bar{f}(a, x) = f(b, (b\ \ a) \cdot x)$ for some $b \in \mathbb{A}$ such that $b \notin \{a\} \cup S(f) \cup S(x)$. Now, we have,

$$\begin{aligned} & g(a, x) \\ = & \ g(b, (b\ \ a) \cdot x) \quad [\because g \text{ respects } \approx_\alpha] \\ = & \ f(b, (b\ \ a) \cdot x) \quad [\because b \notin S(f)] \\ = & \ \bar{f}(a, x) \end{aligned}$$

The lemma follows. □

Equivariant functions are emptily supported functions. So for equivariant functions, we have the following corollary of the above theorem.

Corollary 5.16 For nominal sets X, Y, Z and $f \in \mathbf{Nom}\ (X \times \mathbb{A} \times Y, Z)$ satisfying,

$$\forall x \in X, a \in \mathbb{A}, \quad a \notin S_X(x) \Rightarrow ((\forall y \in Y), a \notin S_Z(f(x, a, y))) \tag{FEQ}$$

there exists a unique $\bar{f} \in \mathbf{Nom}\ (X \times [\mathbb{A}]Y, Z)$ such that,

$$\forall x \in X, a \in \mathbb{A}, \quad a \notin S_X(x) \Rightarrow ((\forall y \in Y), \bar{f}(x, a, y) = f(x, a, y)) \tag{EEQ}$$

Proof. First, we define $\bar{f}$ in the following way.

Let $(x, a, y) \in X \times [\mathbb{A}]Y$. Let us choose $b \in \mathbb{A}$ such that $b \notin \{a\} \cup S_X(x) \cup S_Y(y)$. Then $\bar{f}(x, a, y) = f(x, b, (b\ \ a) \cdot y)$.

In order to show that $\bar{f}$ is an equivariant function, we need to prove a lemma.

Lemma 5.17 Given $x \in X$ and $(a_1, y_1), (a_2, y_2) \in [\mathbb{A}]Y$ such that $a_1, a_2 \notin S_X(x)$ and $(a_1, y_1) \approx_\alpha (a_2, y_2)$, we have $f(x, a_1, y_1) = f(x, a_2, y_2)$.

Proof. Let us choose some $a_3 \in \mathbb{A}$ such that $a_3 \notin S_X(x) \cup S_Y(y_1) \cup S_Y(y_2) \cup S_Z(f(x,a_1,y_1)) \cup S_Z(f(x,a_2,y_2))$. Then, we have,

$$\begin{aligned}
& f(x,a_1,y_1) \\
&= (a_3\ \ a_1) \cdot f(x,a_1,y_1) \quad [\because a_1, a_3 \notin S(f(x,a_1,y_1))] \\
&= f(a_3\ \ a_1) \cdot (x,a_1,y_1) \quad [\text{By equivariance of } f] \\
&= f((a_3\ \ a_1) \cdot x, (a_3\ \ a_1) \cdot a_1, (a_3\ \ a_1) \cdot y_1) \\
&= f(x, a_3, (a_3\ \ a_1) \cdot y_1) \quad [\because a_1, a_3 \notin S(x)]
\end{aligned}$$

Similarly, we can show that $f(x,a_2,y_2) = f(x,a_3,(a_3\ \ a_2) \cdot y_2)$. But since $(a_1,x_1) \approx_\alpha (a_2,x_2)$, so $(a_3\ \ a_1) \cdot x_1 = (a_3\ \ a_2) \cdot x_2$. The lemma follows. □

Now we show that $\bar{f}$ is an equivariant function (axiom (EqP1)). Let $(x,a,y) \in X \times [\mathbb{A}]Y$ and $\pi \in$ Perm. Now, $\bar{f}(x,a,y) = f(x,b,(b\ \ a) \cdot y) = z_1$ (say) for some $b \in \mathbb{A}$ such that $b \notin \{a\} \cup S(x) \cup S(y)$. And $\bar{f}(\pi \cdot x, \pi \cdot a, \pi \cdot y) = f(\pi \cdot x, b', (b'\ \ \pi \cdot a) \cdot \pi \cdot x) = z_2$ (say). We need to show that $\pi \cdot z_1 = z_2$. For this, let us choose $c \in \mathbb{A}$ such that $c \notin S_X(x) \cup S_Y(y) \cup S_Y(\pi \cdot y) \cup (\text{flatten } \pi)$. Then, we have,

$$\begin{aligned}
& \pi \cdot z_1 \\
&= \pi \cdot f(x,c,(c\ \ a) \cdot y) \quad [\text{By lemma (5.17)}] \\
&= f(\pi \cdot x, \pi \cdot c, \pi \cdot (c\ \ a) \cdot y) \quad [\text{By equivariance of } f] \\
&= f(\pi \cdot x, c, \pi \cdot (c\ \ a) \cdot y) \quad [\text{By lemma (2.23)}] \\
&= f(\pi \cdot x, c, (c\ \ \pi \cdot a) \cdot \pi \cdot y) \quad [\text{By corollary (2.25)}] \\
&= f(\pi \cdot x, b', (b'\ \ \pi \cdot a) \cdot \pi \cdot y) \quad [\text{By lemma (5.17)}] \\
&= z_2
\end{aligned}$$

Since we are working with setoids, we need to verify axiom (EqP2). If we had worked with sets instead, lemma (5.17) would have sufficed. Let $(x_1,a_1,y_1),(x_2,a_2,y_2) \in X \times [\mathbb{A}]Y$ such that $x_1 \approx x_2$ and $(a_1,y_1) \approx_\alpha (a_2,y_2)$. Then $\bar{f}(x_1,a_1,y_1) = f(x_1,b_1,(b_1\ \ a_1) \cdot y_1) = z_3$ (say) for some $b_1 \in \mathbb{A}$ such that $b \notin \{a_1\} \cup S_X(x_1) \cup S_Y(y_1)$. And $\bar{f}(x_2,a_2,y_2) = f(x_2,b_2,(b_2\ \ a_2) \cdot y_2) = z_4$ (say) for some $b_2 \in \mathbb{A}$ such that $b_2 \notin \{a_2\} \cup S_X(x_2) \cup S_Y(y_2)$. Now we need to show that $z_3 = z_4$. For this, let us choose some $c \in \mathbb{A}$ such that

$c \notin S_X(x_1) \cup S_X(x_2) \cup S_Y(y_1) \cup S_Y(y_2)$. Then, we have,

$$\begin{aligned}
&z_3 \\
&= f(x_1, c, (c\ \ a_1) \cdot y_1) \quad \text{[By lemma (5.17)]} \\
&= f(x_2, c, (c\ \ a_1) \cdot y_1) \quad [\because x_1 \approx x_2] \\
&= f(x_2, c, (c\ \ a_2) \cdot y_2) \quad [\because (a_1, y_1) \approx_\alpha (a_2, y_2)] \\
&= z_4 \quad \text{[By lemma (5.17)]}
\end{aligned}$$

This shows that $\bar{f}$ is an equivariant function. Now we just need to check the existence and uniqueness conditions. The existence condition follows from lemma (5.17). To prove the uniqueness condition, let us assume $g \in \textbf{Nom}\ (X \times [\mathbb{A}]Y, Z)$ satisfies (EEQ). We need to show that $g = \bar{f}$. Let $(x, a, y) \in X \times [\mathbb{A}]Y$. Let $\bar{f}(x, a, y) = f(x, b, (b\ \ a) \cdot y)$ for some $b \in \mathbb{A}$ such that $b \notin \{a\} \cup S_X(x) \cup S_Y(y)$. Now, we have,

$$\begin{aligned}
&g(x, a, y) \\
&= g(x, b, (b\ \ a) \cdot y) \quad [\because g \text{ respects } \approx_\alpha] \\
&= f(x, b, (b\ \ a) \cdot y) \quad [\because b \notin S(x)] \\
&= \bar{f}(x, a, y)
\end{aligned}$$

The lemma follows. □

In this chapter, we have discussed the properties of the name abstraction functor, which models the notion of alpha equivalence for the elements of nominal sets. We also looked at the freshness condition for binders. With this, we conclude our discussion of the three key notions of nominal set theory — support, freshness and name abstraction.

Chapter 6

Conclusion

We have shown that a considerable portion of the theory of nominal sets can be developed constructively in the dependent type theoretic environment of Agda. We have formalized the category theoretic properties of nominal sets, properties of freshness and name abstraction sets as presented in [6]. This development can form the base for future constructive implementations of nominal techniques in languages like Coq and Agda. One of the key challenges in that direction will be the reduction of complexity of the proofs involved. In general, machine level formalizations are much more tedious as compared to pen and paper formalizations, since one has to work out even the smallest technical detail explicitly. In addition to that, the thorough dependence of our work on setoids made our proofs even more verbose, since the equivalence relation has to be invoked at every step when working with elements of setoids. The situation becomes unwieldy when one has to simultaneously manipulate several setoids, which are also related to one another. It would be interesting to find out how much of this setoid dependence can be done away with, without yielding to non-constructive methods. For example, if the permutations can be represented as sets instead of setoids, something which we believe might be possible, then the verbosity of the proofs can be substantially reduced. The other area of exploration might be looking for a minimal data type for Atom that would satisfy all the required properties. The other problem we faced while developing our proofs in Agda is that very often Agda fails to infer implicit arguments, thus requiring one to explicitly mention all the implicit arguments, something that makes proofs look more verbose than is necessary. If the type inference mechanism of Agda can be made stronger, proofs will become leaner. These improvements would make large-scale development of nominal techniques in Agda more feasible.

Bibliography

[1] Rod M. Burstall. Proving properties of programs by structural induction. *The computer journal*, 12, 1969.

[2] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the church-rosser theorem. *Indagationes Mathematicae (Koninglijke Nederlandse Akademie van Wetenschappen)*, 34(5):381–392, 1972.

[3] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[4] Furio Honsell, Marino Miculan, and Ivan Scagnetto. pi-calculus in (co)inductive type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

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

[6] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013. ISBN 9781107017788.

[7] A. Swan. An Algebraic Weak Factorisation System on 01-Substitution Sets: A Constructive Proof. *ArXiv e-prints*, September 2014.

[8] Mars Climate Orbiter Mishap Investigation Board. Mars Climate Orbiter Mishap Investigation Board Phase I Report, November 1999.

[9] U.S.-Canada Power System Outage Task Force. Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations, April 2004.

[10] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[11] Daniel Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. In ElsaL. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin Heidelberg, 1997.

[12] BrianE. Aydemir, Aaron Bohannon, Matthew Fairbairn, J.Nathan Foster, BenjaminC. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 2005.

[13] Mark Ross Shinwell. *The Fresh Approach: Functional Programming with Names and Binders.* PhD thesis, University of Cambridge, 2005.

[14] Christian Urban and Stefan Berghofer. A recursion combinator for nominal datatypes implemented in isabelle/hol. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 498–512. Springer, 2006.

[15] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in coq: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 174(5):69 – 77, 2007.

[16] R. Zach. Hilbert's program then and now. In *Dale Jacquette, ed., Philosophy of Logic*, volume 5 of *Handbook of the Philosophy of Science*, pages 411–417, Amsterdam, 2006. Elsevier.

[17] L. E. J. Brouwer. *Philosophy and Foundations of Mathematics.* Collected Works, Volume-1. North-Holland Pub. Co., 1975.

[18] A. Heyting. *Intuitionism: an introduction.* Studies in logic and the foundations of mathematics. North-Holland Pub. Co., 1971.

[19] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[20] Per Martin-Löf. An intuitionistic theory of types, 1972.

[21] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

[22] A. Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[23] Jean Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[24] Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.

[25] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[26] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[27] Ulf Norell. Dependently typed programming in agda. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer Berlin Heidelberg, 2009.

[28] Martin Hofmann. *Extensional Concepts in Intensional Type Theory.* PhD thesis, University of Edinburgh, Scotland , UK, July 1995.

[29] Steve Awodey. *Category theory*, volume 49 of *Oxford Logic Guides.* The Clarendon Press Oxford University Press, New York, 2006.

[30] S. Eilenberg and Saunders Mac Lane. General theory of natural equivalences. *Trans. Am. Math. Soc.*, 58:231–294, 1945.

[31] Peter Freyd. Algebraically complete categories. In Aurelio Carboni, MariaCristina Pedicchio, and Guiseppe Rosolini, editors, *Category Theory*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer Berlin Heidelberg, 1991.

[32] Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, May 2006.