From urelements to Computation

Vincenzo Ciancia

From *urelements* to computation

A journey through applications of Fraenkel's permutation model in Computer Science

Vincenzo Ciancia*

Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche - Pisa, Italy
vincenzoml@gmail.com

Abstract. Around 1922-1938, a new *permutation model* of set theory was defined. The permutation model served as a counterexample in the first proof of independence of the Axiom of Choice from the other axioms of Zermelo-Fraenkel set theory. Almost a century later, a model introduced as part of a proof in abstract mathematics fostered a plethora of research results, ranging from the area of syntax and semantics of programming languages to minimization algorithms and automated verification of systems. Among these results, we find Lawvere-style algebraic syntax with binders, final-coalgebra semantics with resource allocation, and minimization algorithms for mobile systems. These results are also obtained in various different ways, by describing, in terms of category theory, a number of models equivalent to the permutation model.
We aim at providing both a brief history of some of these developments, and a mild introduction to the recent research line of "nominal computation theory", where the essential notion of *name* is declined in several different ways.

1 Introduction

In 1922, Fraenkel [16] provided the first proof of independence of the Axiom of Choice. Such proof was based on constructing a model of set theory respecting all axioms except the Axiom of Choice. The used model was called *permutation model*. The permutation model diverts from classical set theory in the fact that it fixes an infinite set of *urelements*, or *atoms*, that are not sets, since they do not not have elements of their own. In all respects, this proof is very abstract, and people who are not acquainted with mathematics may even think that such developments are not terribly useful by themselves. One may wonder why is it important to prove *independence* of an axiom, among many others, of an abstract theory, dealing with possibly *infinite* sets, whose existence in nature is even questionable. And if in order to do so, we need to resort to a model that

* Research partially funded by the European Commission FP7 ASCENS (nr. 257414), and EU QUANTICOL (nr. 600708).

in some sense violates the original intuition, is it really worth the effort to study these results?

Indeed, there are plenty of good arguments in favour of learning and teaching Fraenkel's proof, the permutation model, any important or beautiful mathematical proof, and in general, abstract mathematics. But there is more. It is frequent in science that a mathematical result, designed to play a well-defined role in an abstract proof, finds important applications of its own way after publication. This is also the case for the permutation model. After almost 80 years from its original formulation, the model was used to provide an elegant solution to yet another theoretical problem, that of incorporating variable binding in abstract syntax. This seminal result fostered more research, leading to consistent adoption of the permutation model as a richer set theory than classical Zermelo-Fraenkel sets, providing solid foundations to the study of a number of phenomena in computer science. Nowadays, almost a century after the publication of the original proof by Fraenkel, the theory of computation in the permutation model, called *nominal computation*, has turned into a respectful subject, spanning over different areas of theoretical computer science.

In this paper, we aim at providing a mild introduction to nominal computation theory, and some evidence of its relevance in computer science. For this purpose, we propose a brief historical perspective, and some technical details, on three research lines, making use of the permutation model in the areas of *abstract syntax*, *programming language semantics*, and *automata theory*. We deliberately omit other important subjects (notably, among others, *nominal logics* and *nominal unification*). For a more complete overview of the field, we invite the interested reader to look at the book by Pitts [39], that constitutes a complete, detailed reference for the theory of nominal computation.

The structure of the paper is as follows. In Section 2 we set the scene, by providing a brief account of Fraenkel's permutation model, and a mild introduction to the proof of independence of the Axiom of Choice. In Section 3 we describe the work of Gabbay and Pitts, studying *abstract syntax with variable binding* in the setting of the permutation model. In Section 4, we discuss how the very same permutation model is also used to provide an uniform account of the operational semantics of programming languages where new resources can be generated at runtime (e.g., dynamic channels in a network topology, or fresh object creation in object-oriented programming languages). In Section 5 we deal with some recent developments in automata theory, introducing regular languages with infinite alphabets and memory registers, that have been also recognised as automata in the permutation model. In Section 6 we see how several different theories of names are actually the same theory, by an equivalence formulated in the language of category theory. In Section 7 we draw conclusions and sketch some directions for future work.

2 Urelements and the Axiom of Choice

The so-called Zermelo-Fraenkel axioms (ZF) are a mathematical theory describing the usual notion of a *set*, that is, a uniquely determined collection of *elements*. The theory itself does not make a distinction between a set and an element, simplifying the definition of sets whose elements are, in turn, sets (e.g., the set of all *subsets* of a set).

The *axiom of choice* (AC), was first implicitly formulated by Zermelo in 1904 (see [33] for a detailed account of the history of AC). Zermelo subsequently stated this principle in 1908, calling it the "postulate of choice":

> *A set S that can be decomposed into a set of disjoint parts A, B, C, ..., each containing at least one element, possess at least one subset S' having exactly one element in common with each of the parts A, B, C, ... considered.* (PC)

Nowadays, AC is usually formulated in terms of "choice functions", as follows.

> *For each set S whose elements are pairwise disjoint sets, there is at least one* choice function *f, that is, a function f from S to the union of all the elements of S, such that, for each set A in S, we have $f(A) \in A$* (AC)

Equivalently, by noting that, in ZF, one can also take the *image* S' of a choice function f, AC is also stated as described below:

> *For each set S of pairwise disjoint sets, one can form a set S' containing exactly one element for each set in S.* (AC')

Even though the first formalisation of AC is from 1904, it was only in 1922 that *independence* of AC from ZF was proved by Fraenkel [16]. Such proof was developed by introducing a mathematical model of the axioms of ZF, where AC does not hold. The model of Fraenkel, which is a form of *permutation model*, was later perfected by Mostowski [34], and called "Fraenkel-Mostowski sets" (FM-sets). The permutation model is based on a rather different model than ZF set theory. It introduces an infinite set of *urelements* or *atoms*, that are elements different from the empty set, and having no proper element in turn. Urelements, therefore, are not sets at all. However, urelements are used to form sets, much in the same way as the empty set. Furthermore, urelements are equipped with a group of permutations, whose action is extended to arbitrary sets.

It turns out that AC does not hold in FM. In the remainder of this section, we will attempt to provide a mild technical introduction to the matter. The reader who is not interested in the technical matters of our presentation may skip this part, and continue reading from Section 3, where the developments in abstract syntax based on FM-sets are presented.

In modern terminology, a concise description of FM-sets is given by *finitely supported permutation actions*:

FM-sets = finitely-supported actions of a group of permutations of a countable set. (*FM*)

The above definition (we will discuss finite support in a moment) expands to the following idea: after fixing a countable set $\mathcal{A}$ of atoms, and a set of bijections of the form $\pi : \mathcal{A} \to \mathcal{A}$, objects of the model are pairs $(X, \hat{-})$ where X is a ZF set, and $\hat{-}$ is an *interpretation* of permutations as functions from X to X, so that, for each π, $\hat{\pi} : X \to X$ is the interpretation, or *action*, of the permutation π; such interpretation is required to preserve the identity permutation, and composition of permutations. Furthermore, such *permutation actions* are required to have *finite support* for each element, meaning that, for each element x, there is a minimal, finite set of atoms s, such that only for permutations π that affect s it may happen that $\hat{\pi}(x)$ is different from x.

Example 1. A canonical way to instantiate the model is to consider the natural numbers $\mathbb{N}$ as $\mathcal{A}$. Let us introduce also two particular sets: the set $\mathcal{P}(\mathbb{N})$ of all *finite subsets* of natural numbers, and the set $\mathbb{N}^*$ of all *finite sequences* of natural numbers. So, for example, we have $\{1, 2\} \in \mathcal{P}(\mathbb{N})$ and $[1, 2, 2] \in \mathbb{N}^*$. We will define an interpretation of permutations of $\mathbb{N}$ on these two sets, turning them into two FM-sets. A natural choice on both $\mathcal{P}(\mathbb{N})$ and $\mathbb{N}^*$, is to simply apply π element-wise. Consider the permutations $\pi_{(1,2)}$, swapping 1 with 2 (and acting as the identity on each natural number other than 1 or 2), and $\pi_{(2,3)}$, swapping 2 with 3. We have $\hat{\pi}_{(1,2)}(\{1, 2\}) = \{1, 2\}$, whereas $\hat{\pi}_{(1,2)}([1, 2, 2]) = [2, 1, 1]$. Note that also the set of atoms $\mathbb{N}$ has a canonical permutation action, such that $\hat{\pi}(x \in \mathbb{N})$ is just $\pi(x)$ itself. Therefore, also the set of atoms is a (canonical) FM-set.

To complete the definition of a model of ZF, one also needs to define functions. In FM-sets, so-called *equivariant functions* are considered. When X and Y are sets equipped with permutation actions, a function from X to Y is called *equivariant* whenever for all x in X, we have $f(\hat{\pi}(x))) = \hat{\pi}(f(x))$.

Example 2. The function $head : \mathbb{N}^* \to \mathbb{N}$ mapping a sequence in $\mathbb{N}^*$ to its first element is equivariant, as permuting a sequence also permutes its first element, whereas the function $sum : \mathbb{N}^* \to \mathbb{N}$ mapping a sequence to the sum of its elements is not equivariant. To see this, consider the permutation π swapping 3 with 5. We have $\hat{\pi}([1, 2]) = [1, 2]$, and $\hat{\pi}(3) = 5$. Thus we have $sum(\hat{\pi}([1, 2])) = sum([1, 2]) = 3 \neq \hat{\pi}(sum([1, 2]) = \hat{\pi}(3) = 5$.

Indeed, it is possible to prove that all the axioms of ZF hold in FM with equivariant functions; but the most interesting bit for us is to see a counterexample to the axiom of choice in FM, proving that FM is not a model of AC. Consider the set of sets $\{\mathcal{A}\}$ containing just one element, the set of atoms $\mathcal{A}$ itself. The permutation action associated to sets of sets is element-wise, just as in the case of finite sets. A choice function f for $\{\mathcal{A}\}$ has type $f : \{\mathcal{A}\} \to \mathcal{A}$, and in order to fully specify f, it is sufficient to give the value of $f(\mathcal{A})$, since $\mathcal{A}$ is the only element of $\{\mathcal{A}\}$. However, no matter what value x we fix in $f(\mathcal{A}) = x$, the resulting function is not equivariant. To see this, first note that, for every permutation π, we have $\hat{\pi}(\mathcal{A}) = \mathcal{A}$. Now consider a permutation π such that

$\pi(x) \neq x$. Then we have $f(\hat{\pi}(\mathcal{A})) = f(\mathcal{A}) = x \neq \hat{\pi}(x) = \hat{\pi}(f(\mathcal{A}))$. Since we did not make any assumption on f, except equivariance, this is an example (out of infinitely many) of an FM-set where AC can not be used.

3 Abstract syntax in *FM*-sets

Abstract syntax is among the most important subjects in theoretical computer science. Applications span across a wide number of topics, such as parsing and compiler construction, abstract data types, linguistics, graph theory and so on. The syntactic form of an *abstract syntax tree* (AST) is especially aimed at the application of definitions and proofs, by the principle of *induction on the structure* of terms. ASTs cater for a simple and intuitive definition of terms and data structures, witnessed by their widespread application for data representation, especially in the style of *functional programming*.

However, the simplicity of the AST approach to representation of terms is lost when dealing with *variable binding*. Variable binding happens when a syntactic construct introduces a variable (which is *bound* in the scope of the construct), aimed at denoting a yet-unknown, arbitrary entity within a given scope. For example, the "for all" and "exists" constructs of first-order logic are binding constructs (or *binders*). The bound variable x in "for all x" denotes an arbitrary entity satisfying a certain property, and whose meaning is well-established only in the scope of the "for all" construct. In computer science, the prototypical example of a language with binders is the *lambda calculus* [5], where binding is used to introduce variables denoting arguments of functions. The most prominent feature of binders is the *alpha-equivalence* relation: any bound variable may be substituted with another *fresh* variable, without changing the intended meaning of a term. Here, "fresh" means that the variable chosen to replace the bound one must not appear as a free variable in the scope of the binder. Ideally, two terms that are alpha-equivalent should be identified, so that any function defined (e.g., by induction) on terms can not distinguish alpha-equivalent terms. However, in order to do so, the simple representation of AST becomes more technically involved, and the elegance of inductive definitions is hindered by freshness constraints on variables. In fact, mainstream functional languages do not possess forms of binding in their abstact data type definitions[1].

In order to recover a purely inductive style in definitions and proofs about terms with binders, Gabbay and Pitts, in their seminal paper [17], used induction principles on FM-sets to describe abstract syntax with binders. The richer setting of FM-sets allowed the authors to define a novel set constructor (complementing traditional product, union, complementation, etc.) called *abstraction*, whose equational laws mimic *alpha-conversion*. When proofs are done in FM-sets, there is no need to choose fresh names and substitute variables, as *binding* is internal to the model.

[1] It is worth noting that after the adoption of the *nominal* methods that we are discussing in this section, several extensions of traditional functional languages appeared, where binding and alpha-equivalence are part of the language; see e.g. [41].

One way to make sense of this idea is to think at *FM*-sets (equipped with *equivariant functions* as explained in Section 2) as an enhancement of classical set theory, where each element of a set has a *finite* set of urelements *attached*, subject to the action of permutations[2]. Such finite set coincides with the *finite support* that we mentioned in Section 2. In classical set theory, elements can only be compared for equality; in *FM*-sets, on the other hand, urelements serve as a basic "labelling" mechanism that relates different elements to each other in new and interesting ways.

A set such that every element has an empty support is a classical set; any function between such sets is trivially equivariant. By this observation, classical set theory is embedded in *FM*-sets. On the other hand, elements with non-empty support are similar to terms with free variables, in that they have an open "interface" (the finite set of urelements in the support). For example, in abstract syntax within *FM*-sets, terms of a language with variables and binding form an *FM*-set. Urelements in the support of each term represent its free variables. The abstraction operator defined by Gabbay and Pitts is able to *hide* a free variable, making it bound. The inductive construction of terms with binding is simply done by adding abstraction as one of the allowed set constructors, in the traditional inductive definition of terms.

In this context, it is also worth recalling the so-called *De Bruijn indices* notation, representing terms of the lambda calculus using natural numbers instead of variables, and choosing a fresh number, namely the least natural number that has not yet been used, when introducing a bound variable. It is not difficult to prove [7] that the De Bruijn notation can be also obtained by algebraic constructions in *FM*-sets, using natural numbers as the set of atoms. This also requires a definition of abstraction which is different from that of Gabbay and Pitts, taking care of the choice of a least natural number that we mentioned above. However, the two notions of abstraction turn out to be isomorphic (thus, these are essentially the same notion, with a different concrete representation).

We can summarise the point of view introduced by [17] with the slogan

urelements are names (slogan)

Since [17], a *FM*-set obeying to the finite support condition is called a *nominal set*, and the adjective "nominal" has been used to qualify any mathematical construction that existed in classical set theory, when lifted to nominal sets in order to cater for binding. Some prominent examples, besides *nominal abstract syntax*, are "nominal algebra" [25], "nominal logic" [38], "nominal unification" [43], "nominal automata theory" [10,2], and "nominal Kleene algebra" [18,24].

[2] Additionally, in *FM*-sets it may happen that the action of some (but not all) permutations affecting urelements in the support of an element leave such element unchanged. By this, *FM*-sets also exhibit *symmetry* (see e.g. [39,8] for more details).

4 Program semantics in *FM*-sets

In [29], Milner explained the relevance in computer science of what Needham [36] had called *pure names*. A pure name is an entity that has no defined operation, and can only be compared for identity. That essay is part of an extensive research programme, following the introduction of the Pi-calculus [30] to assign semantics to networks of systems with dynamic communication topology.

In the tradition of programming language semantics, the semantic interpretation function, or simply *semantics*, typically accepts as input a program, its input data, and some context (such as the definition of global functions and variables), and returns a *value* computed by the program upon termination, so that a program denotes a context-dependent function from input to output. In this view, the semantics is very often a *partial* function. The fact that a program may not terminate is considered the same as the semantics being not defined on that particular configuration of program, input, and context. However, this view is limiting when considering interactive systems, where the meaning of a program is more often defined by the interaction possibilities with the external world, rather than by a function computed by the program itself. The prototypical example is that of *services* (e.g., in an operating system, or in a client/server application scenario). A service is typically not intended to terminate at all. Rather, the functionality that it offers is well-defined by the way client requests are processed, results are computed and sent back to clients. Interactive semantics of systems that communicate has been successfully tackled by associating to each program a *transition system*. The semantics of a program is given by *transitions*, defining interaction with the outside world, and a semantic equivalence induced by these interactions, often called *behavioural equivalence*. This was done for the *Calculus of Communicating Systems* (*CCS*) [28], a prototypical parallel programming language. The *CCS* features constructs for: *synchronous communication* on named channels, *interleaving parallel* execution, *non-deterministic choice*, and *hiding* of channels to make these private to a sub-process. The behavioural equivalence of choice in the *CCS* is called *bisimilarity*. Based on this type of semantics, so-called *finite state methods* can be used for full automatic verification of properties of programs. Procedures for *minimization* up-to bisimilarity, mostly based on *partition refinement*, permit one to decide semantic equivalence of finite-state programs. Formulas expressed in the language of *modal logics* can be machine-checked using classical results in the area of *model checking*.

The Pi-calculus is an extension of the *CCS* that permits dynamic reconfiguration of the communication topology of a program, by unifying communication channels and data. Processes are allowed to transmit channels, that can subsequently be used for communication by the receiving process. The semantics of a process must therefore cater for the fact that the received channel may have been private to another process, before communication. To all practical matters, this is the same as considering the transmitted channel *fresh* in the receiving process. If there is a clash between the name of a private channel which is going to be transmitted, and an existing channel in the receiving process, the transmitted

channel can be renamed in the receiving process, as soon as a fresh name is chosen. This machinery is strongly reminiscent of alpha-equivalence (see Section 3), except that the binding operation happens along a transition, rather than in the syntax of a process. Furthermore, channels of the Pi-calculus can only be tested for equality, making a channel nothing more than a pure name. By this, names, and *fresh name* generation, are the crucial aspects of the semantics of the Pi-calculus. Side-conditions related to freshness of channels are part of the definition of the transition system and behavioural equivalence of the language, resulting in a non-standard theory, where classical finite-state methods such as minimization and model checking cannot be applied.

In the nineties, a plethora of influential papers appeared, in a research line aimed at a mathematical theory of the semantics of languages with fresh names, in such a way that classical results could be easily re-formulated. In this paper, will not attempt to mention all of them, but rather focus on just two results that are very relevant for our discussion.

In [15], *presheaf categories* are used to provide a fully abstract account of the semantics of the Pi-calculus. We will relate this result to applications of nominal sets in Section 6. In [31], Montanari and Pistore introduced a semantics for the Pi-calculus, based on *history-dependent automata* (*HDA*) [37]. This work, strongly oriented to practical applications such as the definition of a minimization procedure [12], also led to the implementation of a model checking tool [11]. *HDA* are a kind of transition systems featuring a finite set of local registers[3] for each state. Along transitions, names (including those that are freshly generated) can be stored into registers, and later retrieved. Bisimilarity of *HDA* only refers to registers, rather than the names they contain, which is the crucial step towards a decision procedure. The most interesting bit for our discussion, relating *AC* with nominal sets, nominal abstract syntax, and the Pi-calculus, is that *FM*-sets were used in [31], under the name of *permutation algebras*, as an intermediate representation, when mapping the Pi-calculus into *HDA*. More precisely, the semantics of the Pi-calculus is defined in the setting of *FM*-sets, by the means of mathematical structures called *coalgebras* [40], that generalise transition systems. Coalgebras are equipped with a standard notion of bisimilarity, which in this case encompasses and internalises the side-conditions related to freshness that Milner had introduced. Recalling our slogan from Section 3, namely that *urelements are names*, this is probably not surprising. The mentioned coalgebras in the permutation model could as well have been called just *nominal transition systems*, and the related behavioural equivalence could have taken the name of *nominal bisimilarity*.

[3] And a group of permutations on the registers, which is required for minimization purposes; we are deliberately hiding this aspect under the carpet as it is too technical for the scope of this paper, but we leave this remark as a pointer for the reader interested in symmetry in computation (see [8] for more details).

5 Automata theory in *FM*-sets

Finite-state automata on finite words (just called *automata* for brevity, from now on) are a classical type of finite-state machine in computer science. By the means of a simple *acceptance process*, any automaton *accepts* a *regular language*, that is, an automaton denotes a possibly infinite set of finite sequences, or *words*, drawn from a finite set of symbols, called *alphabet*. Even though these structures describe infinite sets, (languages of) automata are closed under Boolean operations, and such operations can be performed automatically, and efficiently, on the corresponding automata. By this, automata are ubiquitous in computer science, and play an essential role in compilers and interpreters, databases, dictionaries, text analysis, control systems, network routing, and several other widespread applications. The theory of automata is complemented by the language of *regular expressions*, which provides an intuitive way to specify automata. For each regular expression there is an automaton implementing the specification, and for each automaton there is a regular expression specifying it.

A limitation to the expressiveness of automata is finiteness of the alphabet. This is not a problem in the (classical) applications we mentioned so far, where the alphabet is intrinsically finite. However, sometimes it would be desirable to apply the methods of automata and regular expressions to domains where the alphabet is infinite. One example is attempting to characterise sequences of operations of a system, where each operation depends on the user performing it, and the set of users may change over time. Another example is trying to denote the set of possible traces of an entity moving in an unknown (unbounded, possibly infinite) space, and communicating its location every now and then. A third example is the idea, typical of networking security, of defining regular expressions that discard or accept messages in a node of a network, based on a finite portion of the previous history of received messages, and on comparing the content of messages, which is dynamic by its own nature, with previously stored data. Yet another example is provided by the specification of admissible executions of security protocols, where the possible observations on the system depend upon freshly generated, unique *nonces*, drawn from a theoretically infinite domain. Finally, consider protocols for network communication based on packets, where freshly generated *session identifiers* are used to separate the (interleaved) communication streams of independent pairs of actors communicating on the same physical channel.

In [21], Francez and Kaminski defined *finite-memory* automata (*FMA*). These automata accept words drawn from an infinite alphabet, using a global, finite set of memory registers apt to store symbols of the alphabet. Along the acceptance process, a word is read symbol by symbol; each symbol is compared to those that are in the registers; if a new symbol is found, the automaton may store it in a register, replacing another symbol. One thing that should be noted is that these machines do not possess *finite states* in the classical sense, as the possible register assignments along the acceptance process are unknown and not bounded *a priori*. Nevertheless, *FMA* retain good decidability properties. In particular, for the sub-class of *deterministic FMA*, decidability of all boolean operations

is retained, in machines that accept and use previously *unknown* data. We can make sense of decidability in this context by observing that an *FMA* is a machine with two different kinds of memory: one stores the *state* of the automaton, belonging to a finite set; the other assigns *identity* to data that the machine *knows* at runtime. In order to intersect or join two *FMA*, it is only necessary to reason about states, as memory assignments can be dealt with in a symbolic way using registers.

Since [21], research on finite memory automata and their variants, belonging to the family of *register automata*, is still ongoing. However, from what we said about registers, and decidability, it is clear that there is some resemblance, that we are going to discuss below, between *FMA* and *HDA*. Furthermore, symbols of the alphabet in *FMA* can only be compared for equality, and the acceptance process can not make a distinction between different fresh symbols, since it can only recognise that a symbol is unknown, and then store it in a specific register. In other words, symbols behave like pure names, and fresh symbols are subject to a form of alpha-equivalence. Again, recalling that *urelements are names*, it does not come as a surprise that, as shown in [10,2], finite-memory automata are equivalent to automata defined in *FM*-sets, having the set of atoms as alphabet. Referring to our previous examples, the symbols of such an alphabet may denote *names* of users, *names* of locations, *data pointers*, *nonces*, *session identifiers*, etc.

However, it is worth noting that automata defined in *FM*-sets are not finite, as the action of permutations forces any non-trivial *FM*-set to be infinite; but it was also proved in [10] that one may equip *HDA* with an acceptance process, and with decision procedures for boolean operations, so that the obtained class of automata is equivalent to *FMA*. Some results on regular expressions already appeared (see e.g., [18], [24], [26]), even though quite a number of classical results in automata theory still have to be explored in the new setting.

6 Category theory as a unifying framework

We mentioned concepts like "abstract syntax in *FM*-sets", "program semantics in *FM*-sets" and "automata theory in *FM*-sets". But what does it mean, that some mathematical concept is "in *FM*-sets"? To a first approximation, it means that "the same" constructions are done in the new model, but how is this formalised? One could go ahead and say that "the same" means that, in the mathematical definition of every concept of a given theory, the words "set" and "function" should be replaced by "named set" and "equivariant function". This is still unsatisfactory, for example because the already mentioned name abstraction operation is not available in set theory. Thus nominal abstract syntax as defined by Gabbay and Pitts has *not* been obtained by merely replacing "set" with "nominal set". There is a gap between the two models and still there is some similarity. The mathematical language of *category theory* comes to help, by providing general machinery to describe concepts, which can be instantiated in different *categories*. Examples of categories in our context are: *ZF*-sets and ordinary functions, nominal sets and equivariant functions, named sets, presheaf

categories. In category theory, for example, one describes abstract syntax using *algebras* [27], or program semantics using *coalgebras* [40]. Algebras and coalgebras instantiated in classical sets correspond to classical abstract syntax and to labelled transition systems, respectively. Algebras and coalgebras in nominal sets, on the other hand, provide abstract syntax with binding, and program semantics with generation of fresh names. Coalgebras in named sets correspond to *history-dependent automata*.

Let us zoom out a little bit and look at some results in this area. The work by Fiore, Plotkin and Turi [13] uses so-called *presheaf categories* to provide abstract syntax with binders. Roughly, a presheaf category is a typed set theory, often with infinite types. To each type is associated a set of elements of that type[4]. For the case of binding, finite sets (of variables) are used. For each finite set s of variables, elements of type s are terms having their free variabiles included in s. Presheaf categories where also used for the semantics of the Pi-calculus in [15]. We already mentioned the results of [31,12], about *partition refinement* algorithms to decide semantic equivalence. In these papers, *HDA* are formulated as coalgebras in the category of *named sets*, faithfully representing semantics of the Pi-calculus.

These developments can be all considered part of the *nominal computation* framework. For this, one first needs to keep in mind that a *complete* description of Fraenkel's permutation model is not just (FM) as given in Section 2. As typical in category theory, *morphisms* must be considered, leading to a more thorough formulation:

> *FM-sets = finitely-supported actions of a group of automorphisms of a countable set, with permutation-preserving-and-reflecting functions as morphisms.* (FM')

After this, several conclusions follow. First of all, *nominal sets*, *named sets*, and the sheaves in one of the categories used in [15] are *the same* model, by a category-theoretical *equivalence* (see [19,14] for details). A categorical equivalence between two categories of models C and D establishes that, for every model M in C, there is a model $M^{(D)}$ in D, and that for each model N in D there is a model $N^{(C)}$ in C, so that M is isomorphic to $(M^{(D)})^{(C)}$ and N is isomorphic to $(N^{(C)})^{(D)}$. In other words, the two models can be translated to each other back and forth without loss of information.

The significance of such result is two-fold. On the one hand, choosing one of many equivalent models does not change the features of the models that can be expressed. On the other hand, different specification formalisms may have different applications. In fact, nominal sets may be very convenient for specification purposes, given their resemblance to ordinary sets. A nominal set is a collection of elements, with the added notion of permutation action, and an equivariant function is just a function obeying to specific constraints. However, it is only

[4] There also are operations mapping a type to another, playing the same role as permutations in FM; in our mild introduction, we omit the technical details, referring the interested reader to the references for more details.

when resorting to named sets that finite representations, suitable for algorithms, are obtained. Neither presheaf categories nor nominal sets are well-suited for the purpose. A thorough discussion of finite representability of algebraic and coalgebraic constructions in nominal sets, by the means of named sets, can be found in [8]. Presheaf categories, on the other hand, have the advantage of generality. As we said, a presheaf model is a typed set theory, and nominal sets are equivalent to a specific kind of presheaf model, typed by finite sets. But the types can be richer, yielding theories with more features than just pure names. For example, in [3], finite graphs, that is, relations, are used as types to represent syntax and semantics in the presence of so-called *explicit fusions*, that is, explicit constraints in a language that make two names denote the same entity. Names are no longer *pure*, as they may be related to each other by this basic form of *aliasing*. One may wonder if finite representations similar to named sets and *HDA* are available also for these richer theories. This is not possible for all sorts of presheaf categories, but in some interesting cases, that include finite graphs and relations, it is done, by resorting to basic building blocks known as *representable* objects, and to permutation groups over them (see [6]). These finite representations are used in [32,4] to provide formal semantics and finite representations of explicit, dynamic network topologies in a novel variant of the Pi-calcululs, and causal relationships between events in Petri nets. Similarly, in [2], in the context of automata, generalised forms of nominal sets are defined, where different kinds of permutation models give rise to different notions of name, equipped e.g., with partial orders, or relations, and the same finite representations of [6] are used to cater for automata-theoretic decision procedures.

Concluding, no model can be "better" than the others, as they are all equivalent, and each model has specific contexts of applications. However, the categorical formulation, and the equivalence results, shed light on what is essential, and what is accessory, in each model, and pave the way to more general results and novel applications.

7 Beyond pure names

As we saw, a novel theory of computation has been derived from the permutation model, way after its publication as a counterexample in an abstract mathematical proof. Nowadays, these developments are described as "nominal computation", and there is active research on its applications. Classical research questions, such as computability via *Turing machines*, are finding their way in this new area of computation theory [1,22], as well as algebraic description of terms with binding over richer algebraic structures than just pure names [20], coalgebraic interpretations of nominal regular expressions [23], alternative, more powerful interpretations of freshness [42,35], theories of languages of infinite words aimed at automata-theoretic model checking [9], and several other developments scattered along many research groups.

This is just the surface of a much richer theory. In nominal sets, names are solely characterised by *identity*. As we said, bending this assumption, variants of

the theory are obtained, that have, as a primitive concept, e.g., a universal network connecting elements (think of *social networks*), or a universal partial order (such as, *events* and *causal dependencies*). This observation, and the categorical understanding of the constructions that have been done in nominal sets, pave the way to very general frameworks for the definition and finite representation of the syntax, semantics, logics, automata theory, and automated verification of dynamic structures. Much of the theory is yet to be developed. We expect that this will be done soon by the enlarged research community, advancing the state of the art of theoretical computer science in surprising and useful ways.

It is worth recalling once again that all these developments took a century to stem, from the permutation model that Fraenkel developed, in search of a proof of independence of the axiom of choice. We consider this a relevant lesson, among many similar ones, in the history of Science.

Acknowledgments. The author wishes to thank Matteo Sammartino for several interesting discussions on history-dependent automata, and for helping with proofreading this paper.

References

1. Bojanczyk, M., Klin, B., Lasota, S., Torunczyk, S.: Turing machines with atoms. In: Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on. pp. 183–192. IEEE Computer Society (2013)
2. Bojanczyk, M., Klin, B., Lasota, S.: Automata with group actions. In: Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science. pp. 355–364. LICS '11, IEEE Computer Society (2011)
3. Bonchi, F., Buscemi, M.G., Ciancia, V., Gadducci, F.: A presheaf environment for the explicit fusion calculus. Journal of Automated Reasoning 49(2), 161–183 (2012)
4. Bruni, R., Montanari, U., Sammartino, M.: Revisiting causality, coalgebraically. Acta Informatica 52(1), 5–33 (2015)
5. Church, A.: A set of postulates for the foundation of logic. Annals of Mathematics 33(2), 346–366 (1932)
6. Ciancia, V., Kurz, A., Montanari, U.: Families of symmetries as efficient models of resource binding. Electronic Notes in Theoretical Computer Science 264(2), 63 – 81 (2010)
7. Ciancia, V., Montanari, U.: A name abstraction functor for named sets. Electronic Notes in Theoretical Computer Science 203(5), 49 – 70 (2008)
8. Ciancia, V., Montanari, U.: Symmetries, local names and dynamic (de)-allocation of names. Information and Computation 208(12), 1349 – 1367 (2010)
9. Ciancia, V., Sammartino, M.: A class of automata for the verification of infinite, resource-allocating behaviours. Lecture Notes in Computer Science, vol. 8902, pp. 97–111. Springer Berlin Heidelberg (2014)
10. Ciancia, V., Tuosto, E.: A novel class of automata for languages on infinite alphabets. Tech. rep., Technical Report CS-09-003, Leicester (2009)
11. Ferrari, G.L., Gnesi, S., Montanari, U., Pistore, M.: A model-checking verification environment for mobile processes. ACM Trans. Softw. Eng. Methodol. 12(4), 440–473 (2003)

12. Ferrari, G., Montanari, U., Tuosto, E.: Coalgebraic minimization of hd-automata for the -calculus using polymorphic types. Theoretical Computer Science 331(23), 325 – 365 (2005)
13. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Logic in Computer Science, 1999. Proceedings. 14th Symposium on. pp. 193–202. IEEE Computer Society (1999)
14. Fiore, M., Staton, S.: Comparing operational models of name-passing process calculi. Information and Computation 204(4), 524 – 560 (2006)
15. Fiore, M., Moggi, E., Sangiorgi, D.: A fully-abstract model for the pi;-calculus. In: Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on. pp. 43–54. IEEE Computer Society (1996)
16. Fraenkel, A.: Der Begriff "definit" und die Unabhängigkeit des Auswahlaxioms. Berl. Ber. 1922, 253–257 (1922)
17. Gabbay, M., Pitts, A.: A new approach to abstract syntax involving binders. In: Logic in Computer Science, 1999. Proceedings. 14th Symposium on. pp. 214–224. IEEE Computer Society (1999)
18. Gabbay, M., Ciancia, V.: Freshness and name-restriction in sets of traces with names. Lecture Notes in Computer Science, vol. 6604, pp. 365–380. Springer Berlin Heidelberg (2011)
19. Gadducci, F., Miculan, M., Montanari, U.: About permutation algebras, (pre)sheaves and named sets. Higher-Order and Symbolic Computation 19(2-3), 283–304 (2006)
20. Jacobs, B., Silva, A.: Initial algebras of terms with binding and algebraic structure. Lecture Notes in Computer Science, vol. 8222, pp. 211–234. Springer Berlin Heidelberg (2014)
21. Kaminski, M., Francez, N.: Finite-memory automata (extended abstract). In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II. pp. 683–688. IEEE Computer Society (1990)
22. Klin, B., Lasota, S., Ochremiak, J., Toruńczyk, S.: Turing machines with atoms, constraint satisfaction problems, and descriptive complexity. pp. 58:1–58:10. CSL-LICS '14, ACM (2014)
23. Kozen, D., Mamouras, K., Petrisan, D., Silva, A.: Nominal kleene coalgebra. Lecture Notes in Computer Science, vol. 9135, pp. 286–298. Springer Berlin Heidelberg (2015)
24. Kozen, D., Mamouras, K., Silva, A.: Completeness and incompleteness in nominal kleene algebra. Lecture Notes in Computer Science, vol. 9348, pp. 51–66. Springer International Publishing (2015)
25. Kurz, A., Petrisan, D.: On universal algebra over nominal sets. Mathematical Structures in Computer Science 20, 285–318 (2010)
26. Kurz, A., Suzuki, T., Tuosto, E.: On nominal regular languages with binders. Lecture Notes in Computer Science, vol. 7213, pp. 255–269. Springer Berlin Heidelberg (2012)
27. Lawvere, F.W.: Functorial semantics of algebraic theories. Proceedings of the National Academy of Sciences of the United States of America 50(1), 869–872 (1963)
28. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc. (1982)
29. Milner, R.: Whats in a name? In: Herbert, A., Jones, K. (eds.) Computer Systems, pp. 205–209. Monographs in Computer Science, Springer New York (2004)
30. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i. Information and Computation 100(1), 1 – 40 (1992)

31. Montanari, U., Pistore, M.: Pi-Calculus, Structured Coalgebras, and Minimal HD-Automata. Lecture Notes in Computer Science, vol. 1893, pp. 569–578. Springer Berlin Heidelberg (2000)
32. Montanari, U., Sammartino, M.: A network-conscious -calculus and its coalgebraic semantics. Theoretical Computer Science 546, 188 – 224 (2014)
33. Moore, G.: Zermelo's axiom of choice: its origins, development, and influence. Studies in the history of mathematics and physical sciences, Springer-Verlag (1982)
34. Mostowski, A.: Uber den Begriff einer Endlichen Menge. Comptes rendus des sances de la Socit des Sciences et des Lettres de Varsovie, Classe III 31(8), 1320 (1938)
35. Murawski, A., Ramsay, S., Tzevelekos, N.: Bisimilarity in fresh-register automata. In: Logic in Computer Science (LICS), 2015 30th Annual ACM/IEEE Symposium on. pp. 156–167. IEEE Computer Society (2015)
36. Needham, R.M.: Distributed systems. chap. Names, pp. 89–101. ACM (1989)
37. Pistore, M.: History Dependent Automata. Ph.D. thesis, Università di Pisa, Dipartimento di Informatica (1999)
38. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation 186(2), 165 – 193 (2003)
39. Pitts, A.M.: Nominal Sets: Names and Symmetry in Computer Science. Cambridge University Press (2013)
40. Rutten, J.: Universal coalgebra: a theory of systems. Theoretical Computer Science 249(1), 3 – 80 (2000)
41. Shinwell, M.R., Pitts, A.M., Gabbay, M.J.: Freshml: Programming with binders made simple. pp. 263–274. ICFP '03, ACM (2003)
42. Tzevelekos, N.: Fresh-register automata. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 295–306. POPL '11, ACM (2011)
43. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theoretical Computer Science 323(13), 473 – 497 (2004)