Towards Nominal Computation

Mikołaj Bojańczyk [*] Laurent Braud [*†] Bartek Klin Sławomir Lasota [‡]

University of Warsaw

{bojan,klin,sl}@mimuw.edu.pl, laurent.braud@labri.fr

Abstract

Nominal sets are a different kind of set theory, with a more relaxed notion of finiteness. They offer an elegant formalism for describing λ-terms modulo α-conversion, or automata on data words.

This paper is an attempt at defining computation in nominal sets. We present a rudimentary programming language, called Nλ. The key idea is that it includes a native type for finite sets in the nominal sense. To illustrate the power of our language, we write short programs that process automata on data words.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Logic and Meanings of Programs*]: Semantics of Programming Languages

General Terms Theory, Semantics, Automata

Keywords λ-calculus, nominal sets, automata on data words

1. Introduction

The theory of nominal sets originates from the work of Fraenkel in 1922, further developed by Mostowski in the 1930s. At that time, nominal sets were used to prove independence of the axiom of choice and other axioms. In Computer Science, they have been rediscovered by Gabbay and Pitts in [11], as an elegant formalism for modeling name binding. Since then, nominal sets have become a lively topic in semantics. They were also independently rediscovered by the concurrency community, as a basis for syntax-free models of name-passing process calculi, see [14, 16]; and used in automata theory as a framework for describing automata on data words, see [3].

From the point of view of this paper, the most appealing feature of nominal sets is that their natural notion of finiteness, called *orbit-finiteness*, is more relaxed than in classical set theory. For instance, in one variant of nominal sets (the notion of variant is formalized later in the paper as a *data symmetry*), the set of rational numbers is orbit-finite.

Using orbit-finiteness, one can define more powerful variants of automata or other computing devices. As an example, take the standard definition of a nondeterministic finite automaton: it consists of finite sets Q and A for the states and alphabet, as well as three relations $\delta \subseteq Q \times A \times Q$ and $I, F \subseteq Q$. Suppose that we read this definition using orbit-finite sets instead. As proved in [3], the resulting objects have the same expressive power as well-known automata for data words, namely Finite Memory Automata of Francez and Kaminski [10]. The same idea can be applied to other computation models such as monoids [2], deterministic automata [3], two-way automata, pebble automata, or pushdown automata. Each time, the nominal model corresponds to a natural device for data words.

The examples above concerned restricted machine models, with the most expressive being pushdown automata. In this paper, we are interested in general nominal computation, and we attempt to understand it by defining a basic programming language called Nλ, designed so that it can directly process orbit-finite nominal sets. The key idea is that it includes a native type $\mathsf{F_n}\alpha$, which represents orbit-finite sets of elements of type α.

Structure of the paper. The paper has three parts. In Part I, we discuss our design objectives and related work, and show some example problems that our language can help solve. In Part II, we define the syntax and operational semantics of Nλ. The language is parametrized by a data symmetry subject to some conditions. We also show how to represent programs of Nλ so that they can be interpreted on a normal machine. Finally, in Part III, we present a substantial case study: the emptiness problem for alternating automata on data words.

Acknowledgment. We are grateful to Andrew Pitts, Dan Ghica, Pawe Urzyczyn and the anonymous referees, whose constructive comments lead to a great improvement of the presentation of this paper.

Part I: Motivation

2. Design objective for Nλ: avoid coding

Suppose, informally, that we want to design an algorithm that processes possibly infinite, but orbit-finite data structures. As a typical example, let I be the set of nondeterministic finite automata in the nominal sense. In such an automaton all components (states, alphabet, transitions, initial and final states) are orbit-finite, but perhaps infinite in the classical sense. Then consider the function $f : I \to \{\top, \bot\}$ that checks if an automaton accepts some word.

[*] Partially supported by ERC Starting Grant Sosna.

[†] Partially supported by the ESF *Games for Design and Verification* exchange grant nr 3070.

[‡] Partially supported by the Polish Ministry of Science grant nr N N206 567840.

One way to compute such a function is to encode orbit-finite sets as finite data structures (and, implicitly, bit strings). In [3, 16], one can essentially find such representations. Applying such a coding to the problem domain, one can reduce a nominal computational problem to a classical one. The resulting notion of computability depends on the coding scheme, but the schemes implicit in the literature are essentially the same, and the small differences can be corrected by Turing machines.

This approach has been implicitly used in the literature when giving decidability results for decision problems in nominal sets. One example is an algorithm [7] for deciding bisimulation equivalence of orbit-finite transitions systems. Some algorithms for problems from automata theory can be found in [3], including the above example of nondeterministic automata. Another example is the algorithm from [2] for deciding if the language recognized by a given data monoid is first-order definable.

An advantage of this approach is that it does not introduce any new concepts; it just treats nominal sets as syntactic sugar for normal sets. In particular, there is no new “Nominal Church-Turing Thesis”. The very same thing can be seen as a disadvantage: it is a clumsy way of doing computation, and it defeats the purpose of using nominal sets as an attractive syntax.

Nλ is a rudimentary functional programming language that abstracts away from the clumsy details. The design goal for Nλ can be summed up in two words: *avoid coding*. The abstract semantics of the language is defined in terms of nominal sets, and does not refer to any coding scheme. In particular, this means that correctness proofs for programs in Nλ are simpler and more convincing. The language comes with a more concrete semantics which implements a coding, therefore any function expressed in Nλ is effectively computable. However, the coding has to be done only once, as it is done in this paper. Once it has been implemented and proved correct, the programmer can simply use the language without thinking about how nominal objects are represented.

Related work. A closely related language is Fresh O’Caml [17, 18], a functional programming language based on nominal sets, aimed at defining and manipulating data structures with binding, such as λ-terms up to α-conversion. It is similar to Nλ in that both can be seen as typed λ-calculi with some additional type and term constructors. However, these extensions are substantially different.

A crucial ingredient of Fresh O’Caml is a binary type constructor <<α>>β, whose values, intuitively, are values of type β with values of type α (typically basic data values, called names in this context) “abstracted”, or bound. These abstraction types, together with a few other primitives, allow a particularly elegant treatment of α-conversion and capture-avoiding substitution in data structures.

Nλ does not have abstraction types; instead, it has a type constructor F_n to represent finitary collections. In Section 15, we show that Nλ also offers a way to treat binding in data structures, albeit in a slightly less direct and appealing way than Fresh O’Caml. However, binding in data structures is just one of the many things that we can do using our language. Also, our language works not just in standard nominal sets, but also in the more general setting of Fraïssé nominal sets [3].

Another difference with Fresh O’Caml is that the latter is a fully grown programming language, with a working compiler and extensive documentation. On the other hand, Nλ is a rudimentary core language, and a work in progress. We have tried to keep the primitives as simple as possible, while still covering a wide range of examples. We make no claim on universality of the language, and it is likely to evolve. However, it seems that our current version of Nλ is a rather expressive formalism. In Section 4, we survey some functions that can be expressed in it. First, however, we formally define nominal sets and related notions.

3. Nominal sets

We now recall the basics of nominal sets as studied by Gabbay and Pitts [11]. Then, in Section 3.1, we generalize the definitions along the lines of [3].

Data values. Fix a countably infinite set $\mathbb{D}$ of *data values*. In the examples below, we assume $\mathbb{D} = \mathbb{N}$, and write $1, 2, 3$ for elements of $\mathbb{D}$. However, no structure of the data values is used except for equality[1]. A *permutation* of $\mathbb{D}$ is any bijection $\mathbb{D} \to \mathbb{D}$. The group of all permutations will be denoted by G.

Nominal sets. A *right action* of G on a set X is a function $\cdot : X \times G \to X$ subject to the following associativity axioms (we use infix notation, writing $x \cdot \pi$ instead of $\cdot(x, \pi)$):

$$x \cdot (\pi\sigma) = (x \cdot \pi) \cdot \sigma \qquad x \cdot 1 = x,$$

where $\pi\sigma$ refers to multiplication in G and 1 is the unit of G, which is the identity function on $\mathbb{D}$.

Every subset of data values $C \subseteq \mathbb{D}$ defines a subgroup of G,

$$G_C = \{\pi \in G : \pi|_C = id_C\}.$$

A *nominal set* is a set X equipped with a right action of G, such that for every $x \in X$, there is a finite set $C \subseteq \mathbb{D}$ such that

$$x = x \cdot \pi \qquad \text{for every } \pi \in G_C.$$

The set C is then called a *support* of x. In words, every element of a nominal set has a finite support.

For example, consider the set $\mathbb{D}^\omega$ of infinite words over the alphabet of data values. Under the coordinatewise action

$$(d_1, d_2, \ldots) \cdot \pi = (\pi(d_1), \pi(d_2), \ldots).$$

every word w is supported by the set C of all letters that appear in w. The set $\mathbb{D}^\omega$ is not nominal, but the subset of words that contain only finitely many different data values is. Also the smaller set $\mathbb{D}^*$ of finite words is nominal.

Orbit-finite sets. The *orbit* of an element x is the set $x \cdot G \overset{\text{def}}{=} \{x \cdot \pi : \pi \in G\}$. A nominal set is called *orbit-finite* if it is a finite union of orbits, i.e. it has a decomposition

$$X = \bigcup_{i=1}^n x_i \cdot G \qquad \text{for some } x_1, \ldots, x_n \in X.$$

For example, the set $\mathbb{D}$ is orbit-finite (with one orbit). The set $\mathbb{D}^2$ has two orbits, namely the diagonal $\{(d, d) : d \in \mathbb{D}\}$ and the rest.

Nominal subsets. A subset Y of a nominal set X is called a *nominal subset of* X if it has a finite support as an element of the powerset of X. Formally, a finite set $C \subseteq \mathbb{D}$ is a support of Y if

$$Y = Y \cdot \pi \overset{\text{def}}{=} \{y \cdot \pi : y \in Y\} \qquad \text{for every } \pi \in G_C.$$

For example, nominal subsets of $\mathbb{D}$ are its finite or cofinite subsets.

Example 1 For a fixed $d \in \mathbb{D}$, the set $\{(d, e) : e \in \mathbb{D}\}$ is a nominal subset of $\mathbb{D}^2$.

Finitary subsets. A nominal subset Y of X is called *finitary* if it intersects finitely many orbits of X. In particular, if X is orbit-finite (for instance $\mathbb{D}$ or $\mathbb{D}^2$) then all its nominal subsets are finitary. As another example, the set of words of length at most 7 where the data value 6 appears at least twice is a finitary subset of $\mathbb{D}^*$.

Finitary subsets are the foundation of our programming language. We write $\mathsf{F}_n X$ for the family of finitary subsets of a nominal set X. In a nutshell, Nλ is simply typed λ-calculus with a type constructor for finitary subsets.

[1] We will add structure to data values in Section 3.1.

3.1 Generalized nominal sets

The centerpiece example of our paper is an algorithm for deciding emptiness of alternating automata in the nominal world. In its full generality, this example may exploit some nontrivial structure (e.g. an order relation) on the set $\mathbb{D}$. This motivates us to work in the framework of *generalized nominal sets*, introduced in [3]. The idea there was to add more structure to the data values $\mathbb{D}$, such as a total or partial order. For automata, this corresponds to more expressive tests on input letters.

To define generalized nominal sets, one begins with a *data symmetry* (called nominal signature in [3]), i.e., a set $\mathbb{D}$ of data values together with a group G of permutations of $\mathbb{D}$. It is important that G need not contain all permutations. Some examples of data symmetries include:

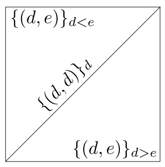
- The *classical symmetry*, where $\mathbb{D}$ is empty, and G contains only the empty bijection. This symmetry yields classical sets.
- The *equality symmetry*, where $\mathbb{D}$ is a countably infinite set, and G contains all bijections. This symmetry yields nominal sets as studied by Gabbay and Pitts.
- The *total order symmetry*, where $\mathbb{D}$ is the set of rational numbers, and G contains all order-preserving bijections of rational numbers.

Later on we will define further data symmetries. Importantly, all examples in this paper are so-called Fraïssé symmetries; this notion follows [3] and will be explained in Section 7.

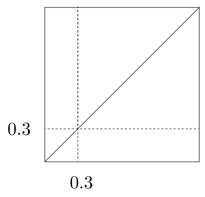
Given a data symmetry $(\mathbb{D}, G)$, one defines $(\mathbb{D}, G)$-*nominal sets* exactly as in Section 3, except with the group G and the data values $\mathbb{D}$ substituted in place of the equality symmetry. Similarly one defines the related notions, like nominal and finitary subsets. From now on, the notion of a nominal set will always be parametrized by a data symmetry.

3.2 Example: subsets of $\mathbb{D}^2$

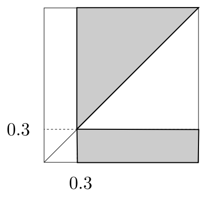
We now show examples of finitary subsets in the total order symmetry. All examples given here will be subsets of $\mathbb{D}^2$, which is shown in the following picture, with the first coordinate on the horizontal axis and the second coordinate on the vertical axis[2].



The picture also shows the three orbits of $\mathbb{D}^2$ under the action of G: above the diagonal, the diagonal, and below the diagonal. Because $\mathbb{D}^2$ is orbit-finite, any nominal subset will be finitary. Consider a support $C = \{0.3\}$. Under the action of G_C, the set $\mathbb{D}^2$ breaks up into thirteen orbits. A subset $X \subseteq \mathbb{D}^2$ has support C if and only if it is a union of some of these orbits. The partition of $\mathbb{D}^2$ into orbits under G_C and a subset X are illustrated below.

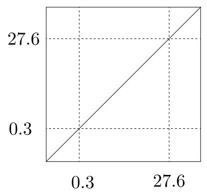


$X =$

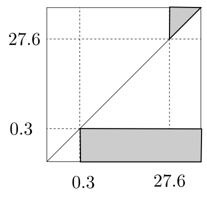


[2] The reader might recognize some similarity to the region construction from timed automata.

Consider now a bigger support than C, say $D = \{0.3, 27.6\}$. The partition of $\mathbb{D}^2$ into orbits under G_D is more refined (there are thirty one orbits), as G_D is a subgroup of G_C. In particular, there are more subsets with support D. The partition and an example subset Y are illustrated below.



$Y =$



The reader can easily see that as the support grows, so does the number of subsets with the support. Also observe that it does not make sense to count the number of "elements" in a finitary set, because the number of elements depends on the choice of support. Even if we choose the least support when counting the number of elements, there is still some room for confusion. Observe that the example set Y has least support D, but it can be decomposed into one orbit under G_C and one orbit under G_D.

4. Nλ as an algorithmic toolkit

Equipped with the basic notions of nominal sets, we discuss some functions that can be programmed in Nλ. In all examples except Section 4.3, we work with the equality symmetry only.

4.1 Transitive closure of a binary relation.

Consider the following binary relation on $\mathbb{D}$

$$R = \{(5,2)\} \cup \{(2,d) : d \neq 5\} \subseteq \mathbb{D}^2.$$

The set R has support $\{2,5\}$. The set $\mathbb{D}^2$ has two orbits, namely the diagonal and the rest. It follows that R, like any nominal subset of $\mathbb{D}^2$, is a finitary subset of $\mathbb{D}^2$, and therefore can be input and processed by programs of Nλ.

A typical thing one might want to do with a binary relation is compute its transitive closure. In this particular example, the transitive closure is

$$R^* = \{(c,d) : c \in \{2,5\},\ d \neq 5\}.$$

Note that R^* is a finitary subset of $\mathbb{D}^2$; one can prove that this holds whenever R is finitary. Also, we may write a recursive program in Nλ that inputs a finitary binary relation R over some type α and outputs its transitive closure. In our example the type α is $\mathbb{D}$, but the program works also for types that are not orbit-finite, such as lists of $\mathbb{D}$. The only condition is that α is an equality type, admitting an equality predicate $\mathtt{eq}_\alpha$.

4.2 Data monoids.

A *data monoid*, as defined in [2], is a monoid where the carrier is a nominal set, and the monoid operation is equivariant. The main result of [2] was a theorem that related first-order logic with aperiodic data monoids. However, nowhere in the paper was it said how one can represent a data monoid in a computer, or test if it is aperiodic. Nλ can help with this.

For example, consider the data monoid M where the carrier is $\{\epsilon\} \cup \mathbb{D}^2$, the identity is ϵ, and the monoid operation is defined by

$$\begin{aligned} \epsilon \cdot m = m \cdot \epsilon = m \qquad & \text{for all } m \in M \text{ and} \\ (d,e)\cdot(d',e') = (d,e') \qquad & \text{for all } d,e,d',e' \in \mathbb{D} \end{aligned}$$

This is the syntactic monoid of the language

$$\{d_1 \cdots d_n \in \mathbb{D}^+ : d_1 = d_n\}.$$

The carrier of this monoid can be seen as the finitary subset of $\mathbb{D}$ `list` that contains lists of sizes zero or two. This set can be represented in Nλ by a short piece of code. The identity of the monoid `M` is the empty list, while the monoid operation can be easily implemented, using pattern matching, as a function

$$\mathtt{f} : \mathbb{D}\,\mathtt{list} \times \mathbb{D}\,\mathtt{list} \to \mathbb{D}\,\mathtt{list}.$$

(As far as the example monoid is concerned, it is not important what `f` does for lists of lengths other than zero or two.)

Using Nλ we can test some properties of monoids represented in this fashion. For instance, we can write a polymorphic function that inputs a finitary set M of values of type α and a binary operation `f` on α, and checks whether `f` is idempotent when restricted to M (in our example, $\alpha = \mathbb{D}\,\mathtt{list}$). In the same spirit, we can write functions that test if a data monoid is aperiodic, commutative, etc.

4.3 Alternating automata with one register

In the literature on data words, there are many automata models, often of incomparable expressive power. One of the maximally expressive automaton models that still has decidable emptiness is an alternating automaton with one register. Roughly speaking, this is an alternating automaton which can store a single data value in a register. Emptiness is decidable for this automaton model, with a proof based on well quasi-orders [6].

In Part III of this paper, we encode the emptiness test for alternating automata with one register into Nλ. To do this, we provide an abstract definition of alternating automata with one register, a definition that only uses the concepts of nominal sets. Then we write a program in Nλ which tests a given automaton for emptiness.

An important advantage of our abstract definition of automata is that it is meaningful for any data symmetry. If the definition is interpreted in the equality symmetry (the usual symmetry for nominal sets), the resulting model is equivalent to the automata from [6, 10]. However, the definition can also be interpreted in the classical symmetry, in which case we get standard alternating automata without data values [4, 5]. Finally, the definition can also be interpreted in the total order symmetry, in which case we get alternating automata on ordered data values, a model that has been studied in [8] and closely related to one-clock timed automata [12, 13, 15]. In all these cases (equality, classical, total order), *the same program* in Nλ decides emptiness for alternating automata with one register; the program only needs to be fed into different interpreters (each symmetry defines a different interpreter).

The proof that the emptiness-testing program terminates depends on the chosen data symmetry. We give an example where the termination proof fails, and indeed the emptiness problem is undecidable.

4.4 Terms of λ-calculus modulo α-conversion.

Nominal sets were originally motivated [11] as an elegant algebraic approach to name binding in syntax, and in particular as a way to represent λ-terms up to α-conversion in a way that admits natural inductive reasoning, using the so-called abstraction types that have α-equivalence classes of terms as values. This idea has been implemented in the nominal programming language Fresh O'Caml [17, 18].

Although Nλ does not have abstraction types, one may represent in it α-equivalence classes of λ-terms directly, as finitary sets of terms. Such a representation is less appealing than the one used in Fresh O'Caml, as it contains many values that do not correspond to α-equivalence classes. However, it still lets us write functions that manipulate terms (such as capture-avoiding substitution). We can also write a Nλ program that tests whether a set of terms is an α-equivalence class.

We elaborate a little on this example in Section 15.

Part II: Nλ

In this part of the paper we introduce Nλ, a rudimentary language for programming in nominal sets over an arbitrary data symmetry of a certain form. The essential idea is to extend simply typed λ-calculus with a collection type that represents finitary sets of values.

We present Nλ in three stages. First, in Section 5, we present a version for the classical symmetry, where no data values or group actions are involved. This is to accustom the reader to our syntax in a familiar setting. Notably, to store intermediate values during a computation according to our operational semantics, we introduce a syntactic construct that collects a finite set of terms. In a practical implementation, these sets would be represented as lists of terms.

Then, after some semantic considerations in Sections 6-7, we generalize the language to arbitrary data symmetries in Sections 8-9. The main conceptual difference, apart from a few new primitives, is that now *finitary*, but potentially infinite, families of terms are considered in intermediate values. This may be seen as problematic, and indeed one may wonder if our semantics is really "operational", given that it directly manipulates infinite structures.

Therefore, in Section 10, we replace infinite collections of terms with a finitary syntactic construct, in a step analogous to replacing finite sets of terms with lists. The result is an entirely finitely presentable language, with a clearly computable reduction semantics, amenable to direct implementation.

The semantics of Sections 8 and 10 are equivalent (formally, bisimilar). The latter may be seen as a reference definition for implementation purposes. The former is more abstract, better suited for reasoning, and more closely related to the simple language of Section 5.

5. A simply typed calculus with finite subsets

In this section, we show how Nλ works in the classical symmetry, where nominal sets are simply sets, and where orbit-finite sets are simply finite sets. The core language is presented in Fig. 1.

The types are as in a simply typed λ-calculus (where b comes from some set of base types), extended with a type constructor F to represent finite collections. The idea is that $\mathbf{1}$ represents a singleton set, and $\mathsf{F}\alpha$ represents finite sets of elements from α. In terms, c comes from some set of term constants with fixed types. In particular, we assume (polymorphic families of) special constants; note that $\mathsf{F}\mathbf{1}$ takes the role of a boolean type. The typing rules of basic terms are standard and omitted from Fig. 1.

To simplify the operational semantics, we extend the language with an additional construct $\mathtt{set}(X)$ of a more semantic flavor, with an intuitive typing rule. With these as intermediate values, the reduction relation $\to_\beta$ on closed terms is defined as expected. We do not commit to any particular evaluation strategy, admitting reduction of arbitrary subterms.

We refrain from showing a formal denotational semantics for the language, but our intention should be clear: types are interpreted as sets, with $\mathbf{1}$ as a singleton type, $\to$ as function space, and F as the finite powerset monad, with the obvious interpretation of constants from Fig 1.

We may now define other typical constants (operations), such as the Haskell bind (written infix as usual):

$$\mathtt{>>=} : \mathsf{F}\alpha \to (\alpha \to \mathsf{F}\beta) \to \mathsf{F}\beta \quad \mathtt{a} \mathbin{\mathtt{>>=}} \mathtt{f} = \mathtt{sum}(\mathtt{map}\ \mathtt{f}\ \mathtt{a}),$$

singleton and set union:

$$\begin{array}{ll} \mathtt{just} : \alpha \to \mathsf{F}\alpha & \mathtt{just}\ \mathtt{a} = \mathtt{add}\ \varnothing\ \mathtt{a} \\ \mathtt{union} : \mathsf{F}\alpha \to \mathsf{F}\alpha \to \mathsf{F}\alpha & \mathtt{union}\ \mathtt{a}\ \mathtt{b} = \mathtt{a} \mathbin{\mathtt{>>=}} (\mathtt{add}\ \mathtt{b}). \end{array}$$

Types:

$$\alpha ::= b \mid \mathbf{1} \mid \alpha \to \alpha \mid \mathsf{F}\alpha$$

Basic terms:

$$M ::= c \mid x \mid \lambda x.M \mid MM$$

Intermediate terms: (X ranges over finite sets of terms)

$$M ::= \cdots \mid \mathtt{set}(X) \qquad \frac{\Gamma \vdash M : \alpha \quad \text{for all } M \in X}{\Gamma \vdash \mathtt{set}(X) : \mathsf{F}\alpha}$$

Special constants:

$$* : \mathbf{1} \qquad \varnothing : \mathsf{F}\alpha \qquad \mathtt{add} : \mathsf{F}\alpha \to \alpha \to \mathsf{F}\alpha$$

$$\mathtt{map} : (\alpha \to \beta) \to \mathsf{F}\alpha \to \mathsf{F}\beta \qquad \mathtt{sum} : \mathsf{FF}\alpha \to \mathsf{F}\alpha$$

$$\mathtt{if} : \mathsf{F}\mathbf{1} \to \alpha \to \alpha \to \alpha$$

Redexes:

$$(\lambda x.M)N \to_\beta M[N/x] \qquad \varnothing \to_\beta \mathtt{set}(\emptyset)$$

$$\mathtt{add}\ \mathtt{set}(X)\ M \to_\beta \mathtt{set}(X \cup \{M\})$$

$$\mathtt{map}\ M\ \mathtt{set}(X) \to_\beta \mathtt{set}(\{MN \mid N \in X\})$$

$$\mathtt{sum}\ \mathtt{set}(\{\mathtt{set}(X_i) \mid i \in I\}) \to_\beta \mathtt{set}(\textstyle\bigcup_{i \in I} X_i)$$

$$\mathtt{if}\ \mathtt{set}(X)\ M\ N \to_\beta M \qquad \text{if } X \neq \emptyset$$

$$\mathtt{if}\ \mathtt{set}(\emptyset)\ M\ N \to_\beta N$$

Set reduction:

$$\frac{M \in X \quad M \to_\beta N}{\mathtt{set}(X) \to_\beta \mathtt{set}(X \setminus \{M\} \cup \{N\})}$$

Figure 1. Nλ without data

Writing `bool` instead of $\mathsf{F}\mathbf{1}$ we define boolean constants and operations:

$$\begin{array}{rlrl}
\mathtt{true} : & \mathtt{bool} & \mathtt{true} = & \mathtt{just}\ * \\
\mathtt{false} : & \mathtt{bool} & \mathtt{false} = & \varnothing \\
\mathtt{or} : & \mathtt{bool} \to \mathtt{bool} \to \mathtt{bool} & \mathtt{or} = & \mathtt{union} \\
\mathtt{not} : & \mathtt{bool} \to \mathtt{bool} & \mathtt{not\ a} = & \mathtt{if\ a\ false\ true},
\end{array}$$

quantifiers and predicate filtering:

$$\begin{aligned}
\mathtt{exists},\ \mathtt{forall} &: \mathsf{F}\alpha \to (\alpha \to \mathtt{bool}) \to \mathtt{bool} \\
\mathtt{filter} &: \mathsf{F}\alpha \to (\alpha \to \mathtt{bool}) \to \mathsf{F}\alpha \\
\mathtt{exists} &= \mathtt{>>=} \\
\mathtt{forall\ X\ f} &= \mathtt{not}(\mathtt{exists\ X}\ \lambda\mathtt{x}.\mathtt{not}(\mathtt{f\ x})) \\
\mathtt{filter\ X\ f} &= \mathtt{X\ >>=}\ \lambda\mathtt{x}.\mathtt{if}\ (\mathtt{f\ x})(\mathtt{just\ x})\varnothing.
\end{aligned}$$

If a type α comes equipped with an equality predicate:

$$\mathtt{eq}_\alpha : \alpha \to \alpha \to \mathtt{bool}$$

we may lift it to an equality predicate for $\mathsf{F}\alpha$ by defining membership and subset predicates:

$$\begin{aligned}
\mathtt{member} &: \mathsf{F}\alpha \to \alpha \to \mathtt{bool} \\
\mathtt{subset} &: \mathsf{F}\alpha \to \mathsf{F}\alpha \to \mathtt{bool} \\
\mathtt{member\ X\ x} &= \mathtt{exists\ X}\ (\mathtt{eq}_\alpha\ \mathtt{x}) \\
\mathtt{subset\ X\ Y} &= \mathtt{forall\ X}\ (\mathtt{member\ Y})
\end{aligned}$$

Finally, equality $\mathtt{eq}_{\mathsf{F}\alpha}$ is defined as subset in both directions.

This core language can be routinely extended with further standard features, e.g. product types:

$$\alpha ::= \cdots \mid \alpha \times \alpha \qquad M ::= \cdots \mid (M, M) \mid \pi_1 M \mid \pi_2 M$$

$$\pi_1(M, N) \to_\beta M \qquad \pi_2(M, N) \to_\beta N$$

Then one can write programs such as binary relation composition `comp`, using an auxiliary function `comp0`:

$$\begin{aligned}
\mathtt{comp0} &: (\alpha \times \beta) \to (\beta \times \gamma) \to \mathsf{F}(\alpha \times \gamma) \\
\mathtt{comp} &: \mathsf{F}(\alpha \times \beta) \to \mathsf{F}(\beta \times \gamma) \to \mathsf{F}(\alpha \times \gamma) \\
\mathtt{comp0\ p\ q} &= \mathtt{if}\ (\mathtt{eq}_\beta(\pi_2\mathtt{p})(\pi_1\mathtt{q}))\ \mathtt{just}(\pi_1\mathtt{p},\pi_2\mathtt{q})\ \varnothing \\
\mathtt{comp\ R\ S} &= \mathtt{R\ >>=}\ \lambda\mathtt{p}.(\mathtt{S\ >>=\ comp0\ p})
\end{aligned}$$

where β is assumed to be an equality type. Similarly we could add recursive types such as lists (that we used in Section 4.2), etc.

Note that in a typical programming language finite collections are modeled by lists. In presence of recursive types, one could view the constants $\varnothing$ and `add` as constructors and define other operations by pattern matching and recursion; with this in mind, our `set()` construct might look like an unnecessary complication. However, in the world of nominal sets the simple list representation fails, and finding a good representation of finitary subsets of nominal sets is one of the technical challenges we shall face. Our "semantic" term construct `set()` is introduced in anticipation of a world where it is easier to say what a finitary collection is than how to write it down.

6. Nominal types: semantic considerations

We now wish to extend the calculus of Section 5 to deal with nominal sets for a data symmetry $(\mathbb{D}, G)$. The general idea is clear: given $(\mathbb{D}, G)$, we interpret types as nominal sets.

Function type. Categories of nominal sets are Cartesian closed, and it is natural to interpret function types as nominal function spaces. There, for nominal sets X and Y, the action of G on a function $f : X \to Y$ is defined by

$$(f \cdot \pi)(x) = (f(x \cdot \pi^{-1})) \cdot \pi.$$

The *nominal function space* $X \to_{\text{fs}} Y$ is the set of functions from X to Y, equipped with the action above, restricted to those functions that have finite support; we call such functions *nominal*. Note that not all nominal functions are equivariant [11]; in fact, a function is equivariant if and only if it has empty support. Write $f : X \to_{\text{fs}} Y$ to say that $f : X \to Y$ is nominal.

Example 2 Consider the equality symmetry. A function $f : \mathbb{D} \to \mathbb{D}$ is nominal if and only if there is a cofinite set $Z \subseteq \mathbb{D}$ such that $f|_Z$ is either the identity or a constant function.

Collection type. It remains to interpret the collection type constructor in the nominal world. This will be of use already in the definition of syntax: we intend to view finitary nominal collections of terms as terms, so we need to define what they are.

The *powerset* of a nominal set X is easy to define:

$$\mathsf{P}(X) = X \to_{\text{fs}} 2$$

where 2 is a two-element set with a trivial G-action. Equivalently, $\mathsf{P}(X)$ is the set of subsets $Y \subseteq X$ with an action defined by

$$Y \cdot \pi = \{y \cdot \pi \mid y \in Y\}$$

restricted to those subsets that have finite support; these are the *nominal subsets* as defined in Section 3.

Example 3 The powerset of an orbit-finite set need not be orbit-finite. For instance, for every data symmetry $(\mathbb{D}, G)$, the powerset of $\mathbb{D}$ contains all finite subsets of $\mathbb{D}$, because every finite subset has a finite support (itself). Two finite subsets of $\mathbb{D}$ cannot be in the same orbit if they have different cardinalities. Therefore, there are infinitely many orbits in $\mathsf{P}(\mathbb{D})$ if $\mathbb{D}$ is infinite.

Observe that a nominal subset Y of a nominal set X is not a $(\mathbb{D}, G)$-nominal set in general, as Y need not be preserved by all permutations from G. However, if $C \subseteq \mathbb{D}$ supports Y then Y is a $(\mathbb{D}, G_C)$-nominal set.

We now introduce our intended nominal interpretation of the collection type: the *finitary powerset*. Recall from Section 3 that a nominal subset $Y \in \mathsf{P}(X)$ is called *finitary* if Y intersects finitely many orbits of X. If C supports Y then Y is finitary if and only if Y is orbit-finite as a $(\mathbb{D}, G_C)$-nominal set. By $\mathsf{F_n}X$ we denote the set of all finitary subsets of X. If X is orbit-finite then all nominal subsets are finitary, i.e. $\mathsf{P}(X) = \mathsf{F_n}X$. It follows that $\mathsf{F_n}X$ need not be orbit-finite even if X is. Actually, the only case when $\mathsf{F_n}X$ is orbit-finite is when X is finite.

Note that our $\mathsf{F_n}X$ is not the free semi-lattice over X; the latter contains only finite subsets of X. Although we do not yet have a clear categorical understanding of $\mathsf{F_n}X$, we choose it for pragmatic reasons: it is a rich collection of subsets of X that we are able to represent in a finite way (see Section 10).

7. Fraïssé symmetries

From now on we restrict attention to a special case of data symmetries, called *Fraïssé symmetries*, defined in [3] (and called Fraïssé signature there). A Fraïssé symmetry is induced by a class $\mathcal{K}$ of finite relational structures, which needs to be closed under isomorphism, substructures and amalgamation. Given such a class, the construction from [3] uses the Fraïssé limit [9] to produce a data symmetry $(\mathbb{D}, G)$, which is well behaved. All three examples listed in Section 3.1 are Fraïssé symmetries:

- The classical symmetry arises from the empty class $\mathcal{K}$.
- The equality symmetry arises from the class $\mathcal{K}$ of finite sets, i.e. finite structures over the empty vocabulary.
- The total order symmetry arises from the class $\mathcal{K}$ of all finite total orders, over a vocabulary with a binary relation.

Other examples include the class $\mathcal{K}$ of finite partial orders or finite graphs. We will see other Fraïssé symmetries in Section 14.

From now on we consider a fixed data symmetry $(\mathbb{D}, G)$ and assume that it satisfies the following conditions:

- *Fraïssé symmetry*: the symmetry is induced by a class $\mathcal{K}$.
- *Least supports*: any element x of a nominal set has the least support with respect to inclusion, which we will denote $supp(x)$.

Under these assumptions, the following basic results hold that will be extensively used in the following:

Lemma 1 $\mathbb{D}$ is orbit-finite under the action of G.

Lemma 2 Orbit-finite sets are closed under Cartesian products.

Lemma 3 (orbit refinement) Let $x \in X$ for some nominal set X. For any finite sets $C \subseteq D$ of data values, there exists a finite subset $\{z_1, \ldots, z_n\} \subseteq X$ such that

$$x \cdot G_C = \bigcup_{i=1}^{n} z_i \cdot G_D.$$

8. A calculus with finitary nominal subsets

An extension of the language of Section 5 to nominal sets over a Fraïssé symmetry $(\mathbb{D}, G)$, is summarized in Fig. 2. We write $supp(X)$ for the least support of X.

Types:

$$\alpha ::= b \mid \mathbf{1} \mid \alpha \to \alpha \mid \mathbb{D} \mid \mathsf{F}\alpha \mid \mathsf{F_n}\alpha$$

Basic terms: as in Fig. 1,
Intermediate terms: as in Fig. 1, but X ranges over finitary sets in

$$\mathtt{set}(X) : \mathsf{F_n}\alpha$$

Special constants: to Fig. 1, add:

$$d : \mathbb{D} \qquad \text{(for each } d \in \mathbb{D}\text{)}$$

$$\mathtt{eq}_{\mathbb{D}} : \mathbb{D} \to \mathbb{D} \to \mathtt{bool} \qquad \mathtt{hull} : \mathsf{F}\mathbb{D} \to \mathsf{F_n}\alpha \to \mathsf{F_n}\alpha$$

Redexes: to Fig. 1, add:

$$\mathtt{eq}_{\mathbb{D}}\ d\ d \to_\beta \mathtt{set}(\{*\})$$

$$\mathtt{eq}_{\mathbb{D}}\ d\ e \to_\beta \mathtt{set}(\emptyset) \qquad \text{(if } d \neq e\text{)}$$

$$\mathtt{hull}\ \mathtt{set}(C)\ \mathtt{set}(X) \to_\beta \mathtt{set}(\textstyle\bigcup_{x \in X} x \cdot G_C) \qquad (C \subseteq_{\mathsf{fin}} \mathbb{D})$$

Set reduction:

$$\frac{M \in X \quad M \to_\beta N \quad supp(X) = C}{\mathtt{set}(X) \to_\beta \mathtt{set}(X \setminus (M \cdot G_C) \cup (N \cdot G_C))}$$

Figure 2. Nλ with data: abstract terms

First, a new type $\mathbb{D}$ of data values is introduced, together with a constant for each data value and an equality predicate on $\mathbb{D}$.

Second, instead of one type constructor F we now have two, written F and $\mathsf{F_n}$. The type $\mathsf{F}\alpha$ represents all *finite subsets* of α, as in Section 5, while $\mathsf{F_n}\alpha$ represents all *finitary nominal subsets* of α. Both type constructors are equipped with constants listed in Fig. 1, and further operations for $\mathsf{F_n}$ (such as `forall`, $\mathtt{eq}_{\mathsf{F_n}\alpha}$ etc.) are defined as in Section 5 for F. The name clash between constants should not lead to confusion and will always be resolved by context. In particular, the are two constructs $\mathtt{set}(X) : \mathsf{F}\alpha$ and $\mathtt{set}(X) : \mathsf{F_n}\alpha$, where in the latter case X ranges over finitary and not finite sets of terms (we explain below how to regard the set of all terms as a nominal set). To avoid confusion, the latter `set()` construct we call *nominal*.

Finally, the new constant `hull` can be used to construct finitary nominal sets, just as `add` could be used to construct finite sets in Section 5. It inputs a finite set C of data values and a finitary set X, and closes X under the action of G_C. For instance in the equality symmetry, the following piece of code

```
hull ∅ {2}
```

represents the whole set $\mathbb{D}$, and the following two calls:

```
hull ∅ {(2,2)}     hull ∅ {(2,3)}
```

create the two orbits of $\mathbb{D}^2$. The relation R from Section 4.1 can be generated by

```
R = hull {2,5} {(5,2), (2,2), (2,3)}.
```

(We use some syntactic sugar here; for example, `{2,5}` would have to be written as '`add 2 (add 5 ∅)`' in our core language.)

Terms of Nλ remain as in Fig. 1, but in the nominal $\mathtt{set}(X)$ construct, X ranges over *finitary*, rather than finite sets of terms. Here we must proceed with care to avoid a circular definition where the existence of a term $\mathtt{set}(X)$ depends on all terms understood as a nominal set. Formally, this can be done by introducing an explicit rank to terms, where the rank of a term is the maximal degree of nesting of `set()` in it. Then the group action of G on terms of rank n is defined by induction, and every term of rank n has a finite support. In other words, the set of terms of each rank n is a nominal set and we may meaningfully speak of its finitary subsets used in the `set()` construct.

Example 4 In the total order symmetry, we have a term that represents the set of all data values bigger than 7:

$$\texttt{set}(\{d \mid d > 7\}).$$

This term has rank 1 and type $\mathsf{F_n}\mathbb{D}$. We also have a term that represents a set of constant functions:

$$\texttt{set}(\{\lambda x.d \mid d > 7\}).$$

We could also write a term for the union of the two sets above, but this would not have a type.We cannot, however, write a term for all data values that are integers, as $\mathbb{Z}$ is not a nominal subset of $\mathbb{D}$:

$$\texttt{set}(\{i : i \in \mathbb{Z}\}) \text{ is not a valid term.}$$

The β-reduction relation $\to_\beta$ is defined as in Section 5, with new redexes for $\texttt{eq}_\mathbb{D}$ and `hull`. Note that for the reduction of `hull` to happen, C must be fully evaluated to a finite set of data value constants.

Finally, the set reduction rule given in Fig. 1 is unsatisfactory for the nominal $\texttt{set}(X)$ construct, as X may have infinitely many elements, causing potentially infinite reductions. We fix this problem by insisting that all elements of X that are in the same orbit of G_C (where C is the least support of X) reduce in parallel and in a uniform way. In other words, the last rule in Fig. 1 is replaced with the last rule of Fig. 2.

Our semantics satisfies the basic properties expected from a typed λ-calculus:

Proposition 4 The β-reduction is well-defined, i.e., if $M \to_\beta N$ then N is a valid term. In addition, types are preserved and so are supports: if C supports M then it also supports N.

Proof
For the first part, we must check that the reduct N only contains finitary sets of terms under the $\texttt{set}()$ construct. The proof goes by induction on the structure of terms. In some cases we make use of the fact that in Fraïssé symmetries, Cartesian products of orbit-finite sets are orbit-finite. □

Proofs of other desirable properties are routine using standard methods:

Proposition 5 The reduction relation $\to_\beta$ has the Church-Rosser property and is weakly normalising.

9. Recursion and examples

The treatment of recursion is standard: we extend the language with a new constant `fix` for any type α, with the type

$$\texttt{fix} : (\alpha \to \alpha) \to \alpha$$

and the reduction rule

$$\texttt{fix}\, M \to_\beta M\, (\texttt{fix}\, M).$$

Using this we may define recursive functions, via the following syntactic sugar:

$$\texttt{F x = M} \quad \overset{\text{def}}{=} \quad \texttt{fix}\,\lambda\texttt{F.}(\lambda\texttt{x.M})$$

where `F` may appear in M.

As usual with recursion, one can write nonterminating programs, so weak normalisation fails. However, the Church-Rosser property holds, and types and supports are still preserved by $\to_\beta$.

We now show some simple examples of recursive programs.

Example 5 Recall Section 4.1 and the problem of computing the transitive closure of a finitary relation. W may write a function

$$\texttt{trans} : \mathsf{F_n}(\alpha \times \alpha) \to \mathsf{F_n}(\alpha \times \alpha)$$

that inputs a finitary binary relation R over some equality type α and outputs its transitive closure:

$$\texttt{step} : \mathsf{F_n}(\alpha \times \alpha) \to \mathsf{F_n}(\alpha \times \alpha)$$
$$\texttt{step R} = \texttt{union R (comp R R)}$$
$$\texttt{trans R} = \texttt{if (eq}_{\mathsf{F_n}(\alpha\times\alpha)}\texttt{(step R) R) R (trans(step R))},$$

where `union` and `comp` calculate the union and composition of given relations, as defined in Section 5.

Example 6 Recall Section 4.2 and the problem of checking properties of monoids. Carriers of data monoids can be represented in Nλ as finitary subsets of some equality type α, with the monoid structure given by an element of α (the unit) and a function of the type $\alpha \times \alpha \to \alpha$ (the multiplication).

Assuming the presence of list types, the carrier of the monoid of Section 4.2 can be represented by taking $\alpha = \mathbb{D}\ \texttt{list}$ and (again, with some syntactic sugar):

```
M = hull ∅ {[],[1,1],[1,2]},
```

and the monoid operation

$$\texttt{f} : \mathbb{D}\,\texttt{list} \times \mathbb{D}\,\texttt{list} \to \mathbb{D}\,\texttt{list}.$$

is easily defined by pattern matching.

Suppose that we want to know if a monoid is idempotent. This can be accomplished by a polymorphic function

$$\texttt{idempotent} : \mathsf{F_n}\alpha \to (\alpha \times \alpha \to \alpha) \to \texttt{bool}$$

which inputs a set of arguments and a binary operation on any equality type α. The code of the function is:

$$\texttt{idempotent M f = forall M (}\lambda\texttt{x.eq}_\alpha\texttt{ (f x x) x).}$$

If we execute the `idempotent` function on our example monoid, the expression:

```
idempotent M f
```

will evaluate to `true` in finite time.

The above two examples do not involve data values directly, and the polymorphic programs `trans` and `idempotent` could be just as well interpreted in the simple language of Section 5. The next example uses data explicitly.

Example 7 One could wonder why there is no primitive in the language that returns a support of a given argument. Somewhat surprisingly, with primitives listed so far we can define a function that computes the least support for equality types. The function will simply exhaustively enumerate all candidates (infinitely many of them!). We stress, however, that this search can be implemented by a finite computation, as we explain in Section 10.

We start by writing a function

$$\texttt{supports} : \alpha \to \mathsf{F}\mathbb{D} \to \texttt{bool}$$

that checks if an element x of an equality type α is supported by a finite set of data values C. It returns true if the orbit of x with respect to G_C is a singleton:

```
supports x C =   singleton (hull C (just x))
 singleton X =   exists X λx.(forall X (eq_α x))
```

Having the function `supports` we define

$$\texttt{supp} : \alpha \to \mathsf{F_n}\mathsf{F}\mathbb{D}$$
$$\texttt{search} : \alpha \to \mathsf{F_n}\mathsf{F}\mathbb{D} \to \mathsf{F_n}\mathsf{F}\mathbb{D}.$$

The result type of `supp` is $\mathsf{F_n}\mathsf{F}\mathbb{D}$ and not just $\mathsf{F}\mathbb{D}$, but the function will always return just a single support. It is implemented by an exhaustive search:

```
      supp x =   search x (just ∅)
    search x X =   if (exists X (supports x))
                       (filter X (supports x))
                       (search x (enlarge X))
```

The search starts with the family X containing just one set of data values: the empty one. If this set is not a support, in the first recursive call the family X contains all singletons. In general, at recursive depth n, X contains all nonempty subsets of data of cardinality at most n. An auxiliary function enlarge : $\mathsf{F_n F}D \to \mathsf{F_n F}D$ increases this cardinality bound by one:

```
  enlarge X =   X >>= λx.(𝔻 >>= λd.just(add x d))
```

Observe that the search surely terminates, as every element of type α has a finite support, and always computes the least support with respect to inclusion. Finally note that we allow ourselves to use the set of data values $\mathbb{D}$ as an object of type $\mathsf{F_n}\mathbb{D}$, since $\mathbb{D}$ can be easily represented as discussed in Section 8.

10. Term representation

The language defined so far is a bit abstract in that is uses a "semantic" construct `set()` in its terms. For practical use, we need to represent the terms in a finite way, so that they can be input by users and manipulated by algorithms, for example by an interpreter that chooses a specific evaluation strategy. Such a representation is possible under the assumption that we work in a Fraïssé symmetry.

The cornerstone of our approach is Lemma 3 from Section 7, which provides a finite representation of finitary subsets:

Lemma 6 For every finitary subset Y of X there are elements $y_1, \dots, y_k$ and a finite set $D \subseteq \mathbb{D}$ such that

$$Y = \textstyle\bigcup_{i=1}^{k} y_i \cdot G_D.$$

Proof
Take as D any support of Y, and as $x_1, \dots x_k$ any representatives of those G-orbits of X that are intersected by Y. Apply Lemma 3 to each x_i with $C = \emptyset$; take the union of the results and choose as y_i those z's that are elements of Y. □

Note that the representation is not unique. Consider, for instance, the total order symmetry, where $\mathbb{D}$ is the rational numbers. Let Y be the finitary subset of $\mathbb{D}$ that contains data values in the open interval $(1; 7)$. Here are some representations of Y:

$$2{\cdot}G_{\{1,7\}}; \quad 6{\cdot}G_{\{1,7\}}; \quad 2{\cdot}G_{\{1,3,7\}} \cup 3{\cdot}G_{\{1,3,7\}} \cup 6{\cdot}G_{\{1,3,7\}}.$$

As the last one suggests, one can make the set D of data values grow, which entails a growing set of representatives $\{y_1, \dots, y_k\}$.

Thanks to Lemma 6, for a finitely presented language we may replace the nominal `set()` construct from Fig. 2 with a construct

$$\langle M_1, \dots, M_k \rangle^{\{d_1 \dots d_n\}}$$

of finite arity (cf. Fig. 3). Intuitively, its purpose is to mimic

$$\mathtt{hull}\,\{d_1 \dots d_n\}\;\{M_1, \dots, M_k\};$$

we prefer however a separate construct for technical convenience. We shall call terms built with the new construct *concrete*, as opposed to *abstract* terms built using the nominal `set()` construct.

To define a reduction relation on concrete terms, a few notions and results are needed. First, there is an obvious function $M \mapsto [M]$ from concrete to abstract terms, defined by induction with the only nontrivial case:

$$[\langle M_1, \dots, M_k \rangle^C] = \mathtt{set}(\textstyle\bigcup_{i=1}^{k} [M_i] \cdot G_C).$$

Types: as in Fig. 2.
Basic terms: as in Fig. 1-2.
Intermediate terms: (C ranges over finite subsets of $\mathbb{D}$)

$$M ::= \cdots \mid \langle M_1, \dots, M_k \rangle^C \qquad \frac{\Gamma \vdash M_i : \alpha \quad \text{for } i = 1..k}{\Gamma \vdash \langle M_1, \dots, M_k \rangle^C : \mathsf{F_n}\alpha}$$

Special constants: as in Fig. 2.
Redexes:

$$(\lambda x.M)N \to_{\hat\beta} M[N/x] \quad \varnothing \to_{\hat\beta} \langle\rangle^{\emptyset} \quad \mathtt{fix}\,M \to_{\hat\beta} M\,(\mathtt{fix}\,M)$$

$$\frac{D = \mathit{supp}([M]) \quad \langle K_1, \dots, K_n \rangle = \mathit{ref}^{C \cup D}(\langle M_1, \dots, M_k \rangle^C)}{\mathtt{add}\,\langle M_1, \dots, M_k \rangle^C\, M \to_{\hat\beta} \langle M, K_1, \dots, K_n \rangle^{C \cup D}}$$

$$\frac{D = \mathit{supp}([M]) \quad \langle K_1, \dots, K_n \rangle = \mathit{ref}^{C \cup D}(\langle M_1, \dots, M_k \rangle^C)}{\mathtt{map}\,M\,\langle M_1, \dots, M_k \rangle^C \to_{\hat\beta} \langle MK_1, \dots, MK_n \rangle^{C \cup D}}$$

$$\frac{\begin{array}{c} M_i = \langle M_{i1}, \dots, M_{in_i} \rangle^{C_i} \text{ for } i = 1 \dots k \\ E = C_1 \cup \cdots \cup C_k \\ \langle K_{i1}, \dots, K_{il_i} \rangle = \mathit{ref}^E(\langle M_{i1}, \dots, M_{in_i} \rangle^{C_i}) \text{ for } i = 1 \dots k \end{array}}{\mathtt{sum}\,\langle M_1, \dots, M_k \rangle^C \to_{\hat\beta} \langle K_{11}, \dots, K_{1l_1}, \dots, K_{k1}, \dots, K_{kl_k} \rangle^E}$$

$$\mathtt{if}\,\langle M_1, \dots, M_k \rangle^C\, M\, N \to_{\hat\beta} M \qquad \mathtt{if}\,\langle\rangle^C\, M\, N \to_{\hat\beta} N$$

$$\frac{\langle K_1, \dots, K_n \rangle = \mathit{ref}^{C \cup D}(\langle M_1, \dots, M_k \rangle^C)}{\mathtt{hull}\,\mathtt{set}(D)\,\langle M_1, \dots, M_k \rangle^C \to_{\hat\beta} \langle K_1, \dots, K_n \rangle^{C \cup D}}$$

Set reduction:

$$\frac{\begin{array}{c} \langle N_1, \dots, N_n \rangle^D \text{ is a short form of } \langle M_1, \dots, M_k \rangle^C \\ N_i \to_{\hat\beta} K \end{array}}{\langle M_1, \dots, M_k \rangle^C \to_{\hat\beta} \langle N_1, \dots, N_{i-1}, K, N_{i+1}, \dots, N_n \rangle^D,}$$

Figure 3. Nλ with data: concrete terms

Lemma 6 implies that the mapping $M \mapsto [M]$ is surjective (although, as we explained above, not injective). It also obviously preserves types.

Lemma 7 The following operations can be computed:

1. Given concrete terms M, N, decide if $[M] = [N]$;
2. Given a concrete term M, find $\mathit{supp}([M])$;
3. Given concrete terms M, N, and a finite set C of data values, decide if $[M] \cdot G_C = [N] \cdot G_C$.

Proof
By simultaneous induction on the size of terms. □

The following is a computational version of the orbit refinement lemma (Lemma 3):

Lemma 8 Given concrete terms $M_1, \dots, M_k$ and $C \subseteq E$, one can compute concrete terms $K_1, \dots, K_n$ such that

$$[\langle M_1, \dots, M_k \rangle^C] = [\langle K_1, \dots, K_n \rangle^E].$$

We then denote

$$\mathit{ref}^E(\langle M_1, \dots, M_k \rangle^C) = \langle K_1, \dots, K_n \rangle.$$

We may now formulate reduction rules for basic redexes of the language as in Fig. 3. For instance, to reduce a concrete term

$$\mathtt{add}\,\langle M_1, \dots, M_k \rangle^C\, M,$$

one first computes the least support of M using Lemma 7(2), then uses Lemma 8 to refine $\langle M_1, \dots, M_k \rangle^C$, and finally adds M to the resulting list of terms. Other redexes are similar.

For the set reduction rule, it is tempting to write simply:

$$\frac{M_i \to_{\hat\beta} N}{\langle M_1, \ldots, M_n\rangle^C \to_{\hat\beta} \langle M_1, \ldots, M_{i-1}, N, M_{i+1}, \ldots, M_n\rangle^C}.$$

However, this leads into problems: note that if $[M_1] \cdot G_C = [M_2] \cdot G_C$ then $[\langle M_1, M_2\rangle^C] = [\langle M_1\rangle^C]$, but the above rule may let $\langle M_1, M_2\rangle^C$ and $\langle M_1\rangle^C$ reduce differently. To avoid this, we convert concrete terms to a canonical form before they can reduce.

Specifically, we say that a concrete term $M = \langle M_1, \ldots, M_k\rangle^C$ is in *short form* if:

- $C = \mathit{supp}([M])$, and
- $[M_i] \cdot G_C \neq [M_j] \cdot G_C$ for $i \neq j$.

An arbitrary concrete term is in short form if all its subterms $\langle M_1, \ldots, M_k\rangle^C$ are in short form.

Lemma 9 For each concrete term M, a concrete term N in short form such that $[M] = [N]$ exists and can be computed. (N is then called a short form of M).

Proof
By induction on M, using Lemma 7. □

Using short forms, the set reduction rule is defined as in Fig. 3.

All these complications are rewarded by a close correspondence of reduction semantics of concrete and abstract terms:

Proposition 10 The reduction $\to_{\hat\beta}$ is bisimilar to $\to_\beta$, i.e.:

- if $M \to_{\hat\beta} N$ then $[M] \to_\beta [N]$, and
- if $[M] \to_\beta K$ then there exists a concrete term N such that $M \to_{\hat\beta} N$ and $K = [N]$.

Proof
By structural induction on concrete terms. □

Part III: Case studies

In this part, we demonstrate the potential of Nλ as a basis of an expressive programming language.

So far we have ignored many useful features typically found in functional languages, such as recursive datatypes (e.g. lists) and pattern matching, the `let` construct etc. We believe that adding this kind of features to Nλ is an issue orthogonal to our concerns and may be done along standard lines. In the examples to follow we feel free to use recursive types and `let` as if they were in the language.

11. Alternating automata

As a non-trivial example of nominal programming, in the remaining sections we consider the emptiness problem for alternating automata. We deliberately choose a borderline problem: it is decidable only under some restrictions on the state space of the automaton, and only for certain data symmetries. Moreover, even in known decidable cases the problem is extremely complex (non-primitive recursive, except for the classical symmetry, where it is PSPACE-complete).

Definition of alternating automaton Given a data symmetry $(\mathbb{D}, G)$, an alternating automaton is given by:

1. An input alphabet A, an orbit-finite nominal set.
2. A set of states Q, an orbit-finite nominal set.
3. A partition of states $Q = Q_\exists \cup Q_\forall$ into two nominal subsets.
4. A transition function, that is a nominal function
$$\delta : Q \times A \to_{\text{fs}} \mathsf{F}Q.$$
5. An initial state $q_I \in Q$.
6. The final states F, a nominal subset of Q.

An alternating automaton is used to accept or reject a word $w \in A^*$. The semantics is defined as follows. A *configuration* of the automaton is a finite set of states. We write X, Y, Z for configurations. The initial configuration is $\{q_I\}$. A configuration is called *final* if it is a subset of F.

We now define a one step transition relation which says how to go from one configuration to another by reading an input letter. Suppose that X, Y are configurations and $a \in A$ is an input letter. Then we write $X \overset{a}{\rightsquigarrow} Y$ if the following hold for every $q \in X$:

- If $q \in Q_\exists$, then Y contains some state from $\delta(q, a)$.
- If $q \in Q_\forall$, then Y contains all states from $\delta(q, a)$.

We say the automaton *accepts* an input word $a_1 \cdots a_n \in A^*$ if there are configurations $X_0, \ldots, X_n$ such that X_0 is the initial configuration, X_n is final, and

$$X_0 \overset{a_1}{\rightsquigarrow} X_1 \overset{a_2}{\rightsquigarrow} X_2 \overset{a_3}{\rightsquigarrow} \cdots \overset{a_n}{\rightsquigarrow} X_n. \qquad (1)$$

We say X can reach Y in one step if $X \overset{a}{\rightsquigarrow} Y$ holds for some $a \in A$, and we write this $X \rightsquigarrow Y$. We denote the reflexive transitive closure of $\rightsquigarrow$ by $\rightsquigarrow_*$. When $X \rightsquigarrow_* Y$ holds, we say that X can reach Y. The automaton accepts some word if and only if the initial configuration can reach some final configuration.

From now on, we are interested in the *emptiness problem*: given an alternating automaton, decide if it accepts some word.

Example 8 Consider the equality symmetry. We construct an automaton that recognizes the language

$$\{d_1 \cdots d_n : n \in \mathbb{N} \text{ and } d_i \neq d_j \text{ for all } i \neq j\}.$$

The state space is $Q = \mathbb{D} \cup \{\top, \bot\}$, where the states $\top$ and $\bot$ are in their own orbits. The initial state is $\top$, and all states except $\bot$ are final. All states belong to $Q_\forall$ and $Q_\exists$ is empty.

The automaton scans the word in state $\top$, and every time it sees a data value d, it spawns a new thread with state d. This corresponds to the transition

$$\delta(\top, d) = \{\top, d\} \qquad \text{for } d \in \mathbb{D}.$$

When a thread with state d sees a letter e, then it ignores it and keeps on scanning the word if $e \neq d$, otherwise it enters the error state, because d has appeared twice:

$$\delta(d, e) = \begin{cases} \{d\} & \text{if } d \neq e \\ \{\bot\} & \text{otherwise} \end{cases} \qquad \text{for } d, e \in \mathbb{D}$$

Finally, it is impossible to recover from the error:

$$\delta(\bot, d) = \{\bot\} \qquad \text{for } d \in \mathbb{D}.$$

One-dimensional alternating automata Let $k \in \mathbb{N}$. A nominal set is called *k-dimensional* if every element is supported by a set $C \subseteq \mathbb{D}$ of cardinality k. Clearly, an orbit-finite set is k-dimensional iff each of its orbits is so.

Example 9 Up to isomorphism, there is only one zero-dimensional one-orbit set. Examples of one-dimensional sets are $\mathbb{D}$ and the set $\{d\} \times \mathbb{D}$ for any $d \in \mathbb{D}$ (with the action of G not changing the first coordinate). Examples of two-dimensional sets are $\mathbb{D}^2$ or the set of all two-element subsets of $\mathbb{D}$. Every k-dimensional set is also l-dimensional for $l > k$.

In most data symmetries, with the exception of the classical symmetry, the set $\mathbb{D}^2$ is not one-dimensional.

An alternating automaton is k-dimensional if its state space is (the alphabet is not taken into account). For instance, the automaton in Example 8 is one-dimensional. From now on we consider only one-dimensional automata.

Fact 11 A single-orbit set is k-dimensional if and only if it is an image, under some equivariant function, of an orbit of the set $\mathbb{D}^k$.

Our definition of alternating automaton can be instantiated to various data symmetries.

- In the classical symmetry, the notion of dimension is irrelevant, since every nominal set is already zero-dimensional. When instantiated to the classical symmetry, our definition is equivalent to ordinary alternating automata.
- When instantiated to the equality symmetry, our k-dimensional alternating automata are equivalent to alternating k-register automata of Demri and Lazić [6]. Emptiness is known to be undecidable for $k \geq 2$, and decidable for $k = 1$. In the latter case, the complexity of the problem is very high: it is not bounded by any primitive recursive function [1, 12].
- When instantiated to the total order symmetry, our k-dimensional alternating automata are equivalent to the model of k-register automata studied by Figueira, Hofman and Lasota in [8]. These automata are very closely related (some details have to be adjusted) to alternating k-clock timed automata. Emptiness of those is known to be undecidable for $k \geq 2$, and decidable for $k = 1$ (with a similar lower complexity bound as above). The decidability result was obtained independently by Lasota and Walukiewicz [12, 13], as well as Ouaknine and Worrell [15].

In the remaining sections we will prove the following result:

Theorem 12 *There exists a single program in Nλ, which decides emptiness of alternating one-dimensional automata for:*

- *the classical symmetry,*
- *the equality symmetry, and*
- *the total order symmetry.*

12. Well-quasi order

Well-quasi orders (WQOs) are the key technical tool in the algorithm and its proof of correctness. According to a classical definition, a quasi order $(X, \leq)$ is a WQO, if for every infinite sequence

$$x_1, x_2, \ldots \in X$$

there exist indexes $i < j$ such that $x_i \leq x_j$. It is well known that $\leq$ is a WQO iff it is well-founded and every antichain is finite.

We extend this definition to the nominal setting. A *nominal quasi order* is a nominal set X together with a quasi order $\leq$ which is a nominal subset of $X \times X$. Let C be the least support of $\leq$. A nominal quasi order is called a *nominal WQO* if for every infinite sequence

$$x_1, x_2, \ldots \in X$$

there exist indexes $i < j$ and a permutation $\pi \in G_C$ such that $x_i \cdot \pi \leq x_j$. (As C supports the order $\leq$ relation, one could equivalently require $x_i \leq x_j \cdot \pi$ for some $\pi \in G_C$.) The following lemma gives an equivalent and maybe cleaner definition.

Lemma 13 A nominal quasi order X is a nominal WQO iff it is well founded and every antichain is a finitary subset of X.

For example, any orbit-finite set X, with the discrete partial order (all different elements are incomparable) is a nominal WQO.

Theorem 14 *For all data symmetries mentioned in Theorem 12, if Q is an orbit-finite one-dimensional nominal set then $\mathsf{F}Q$, ordered by inclusion, is a nominal WQO.*

Proof
The proof depends heavily on the data symmetry involved and used deep combinatoric results such as Dickson's Lemma (for the equality symmetry) or Higman's Lemma (for the total order symmetry). $\square$

Example 10 The assumption on Q being one-dimensional is important. As an illustrating example consider the equality symmetry and the two-dimensional set $Q = \mathbb{D}^2$. The family of finite sets, for all $n \in \mathbb{N}$, of the following form

$$X_n = \{\langle d_1, d_2\rangle, \langle d_2, d_3\rangle, \ldots, \langle d_n, d_1\rangle\},$$

where $d_1, \ldots, d_n$ are distinct data values, forms an orbit-infinite antichain in $\mathsf{F}Q$. Indeed, if $n \neq m$ then there is no bijection π such that $X_n \cdot \pi \subseteq X_m$. Thus Q is not a nominal WQO.

13. Decision procedure for emptiness

We now use the results of the previous section to give an emptiness algorithm for one-dimensional alternating automata. The algorithm works in the total order symmetry and in the equality symmetry, thus reproving the results of [6, 8]. The algorithm also works in the classical symmetry, but in this case it has suboptimal complexity. One of our contributions is that we make the similarities between [6, 8] explicit: we produce one piece of code in Nλ, which solves the emptiness problem for one-dimensional alternating automata for all these data symmetries[3]. Also, because the code is generic, it only focusses on the essence of the problem, and needs not to talk about technical details.

In addition, the same code solves the emptiness problem for any data symmetry which satisfies the condition of Theorem 14:

$$\text{If } Q \text{ is an orbit-finite one-dimensional nominal set then } \mathsf{F}Q\text{, ordered by inclusion, is a nominal WQO.} \tag{2}$$

As an illustrating example, in Section 14 we describe a new data symmetry, called the forest symmetry, that has property (2). We also show that the partial order symmetry does not have property (2), so the code does not work (i.e. may not terminate) for that symmetry. In fact, the emptiness of one-dimensional alternating automata is undecidable there.

In short, although our program works in many Fraïssé symmetries, the decidability (and also complexity) landscape depends on the choice of symmetry.

13.1 High-level overview of the algorithm

The algorithm for checking emptiness of alternating automata runs two semidecision procedures in parallel. The first procedure terminates if and only if the automaton is nonempty, the second procedure terminates if and only if the automaton is empty[4].

[3] Our code makes syntactic sense for two- and higher dimensional automata as well. When executed for such inputs, the program may not terminate.

[4] Formally speaking, our language does not allow running two procedures in parallel. However, it is not difficult to combine the two procedures into a single one.

The nonemptiness semidecision procedure. This procedure simply does a breadth-first search through all reachable configurations. Define $\mathcal{X}_n$ to be the set of configurations that can be reached from the initial configuration in n steps,

$$\mathcal{X}_0 \stackrel{\text{def}}{=} \{\{q_I\}\} \qquad \mathcal{X}_{n+1} \stackrel{\text{def}}{=} \{X' : X \rightsquigarrow X' \text{ for some } X \in \mathcal{X}_n\}.$$

The algorithm calculates the sets $\mathcal{X}_0, \mathcal{X}_1, \mathcal{X}_2, \ldots$ and searches each one for a final configuration. The procedure terminates if and only if the automaton is non-empty. The only question is: how can we represent $\mathcal{X}_n$? The following fact implies that each $\mathcal{X}_n$ can be stored by a program of Nλ.

Fact 15 If $\mathcal{X}_n$ is finitary, then so is $\mathcal{X}_{n+1}$.

The nonemptiness semidecision procedure is very straightforward, and it requires almost no assumptions to work. In particular, it works for all Fraïssé symmetries, not just those satisfying (2), and it works also for automata of arbitrary dimension, not just one.

The emptiness semidecision procedure. The whole weight of the algorithm is in the emptiness semidecision procedure, which searches for a finitary witness of emptiness. Unlike the nonemptiness semidecision procedure, its proof of termination requires the automaton to be one-dimensional and the data symmetry to satisfy (2).

For a set $\mathcal{X}$ of configurations, define

$$\mathcal{X}\uparrow \;=\; \{X' : X' \text{ is a configuration that includes some } X \in \mathcal{X}\}.$$

Proposition 16 The automaton is empty if and only if there is a finitary subset $\mathcal{X}$ of configurations such that:

- $\mathcal{X}\uparrow$ contains the initial configuration.
- $\mathcal{X}\uparrow$ contains no final configurations.
- Whenever $X \in \mathcal{X}$ and $X \rightsquigarrow X'$ then $X' \in \mathcal{X}\uparrow$.

The emptiness semidecision procedure searches through all finitary subsets of configurations, and terminates if it finds one that satisfies the conditions in Proposition 16.

13.2 Implementing the algorithm in Nλ

We begin by typing the program. The input to the program is an alternating automaton. Its alphabet is thus a parameter of type $\mathsf{F}_{\mathsf{n}}\alpha$ for some type α. Similarly, the set of states of the automaton is of type $\mathsf{F}_{\mathsf{n}}\beta$, for some type β. We assume that α and β are equality types. An automaton is thus given by a tuple of type:

$$\mathsf{F}_{\mathsf{n}}\alpha \times \mathsf{F}_{\mathsf{n}}\beta \times \mathsf{F}_{\mathsf{n}}\beta \times (\beta \times \alpha \to \mathsf{F}\beta) \times \beta \times \mathsf{F}_{\mathsf{n}}\beta,$$

where the six coordinates correspond to A, $Q_\exists$, $Q_\forall$, δ, q_I and F, respectively. Call the type above $\mathtt{aut}(\alpha, \beta)$. Therefore, our emptiness algorithm will have the following type

$$\mathtt{aut}(\alpha, \beta) \to \mathtt{bool}. \tag{3}$$

We now code the algorithm described in Section 13.1. The program consists of two semidecision procedures, `nonempty` and `empty`, both of type (3). The simpler nonemptiness procedure is described below; the more complex emptiness procedure is omitted due to lack of space.

We start with a wrong solution to `nonempty`, to motivate the correct one. At first sight, the problem could be solved by computing the transitive closure of the $\rightsquigarrow$ relation, for instance using the program `trans` defined in Example 5. However, we cannot apply `trans` to $\rightsquigarrow$, as $\rightsquigarrow$ is not finitary!

Instead, we systematically compute configurations reachable from $\{q_I\}$ in k steps, for $k = 0, 1, \ldots$. We focus only on the nominal aspects of the program and skip those fragments that do not process infinite nominal sets. For instance, we assume for simplicity that we have a function

$$\mathtt{conf\text{-}input\text{-}succ} : \mathsf{F}\beta \to \alpha \to \mathsf{F}_{\mathsf{n}}(\mathsf{F}\beta)$$

that computes, for a given configuration X and an input letter a, the finite set of all successor configurations of X via a. This function uses functions `add` and δ. Then we may easily define `conf-succ` : $\mathsf{F}\beta \to \mathsf{F}_{\mathsf{n}}(\mathsf{F}\beta)$ that computes, for a given configuration X, the finitary set $\{X' : X \rightsquigarrow X'\}$:

```
conf-succ x = A >>= (conf-input-succ x)
```

Similarly, `conf-succ` may be lifted to finitary sets of configurations:

```
X >>= conf-succ
```

The above term inputs a finitary set `X` of configurations, and outputs the set of its successors. Therefore, this term realizes the mapping $\mathcal{X}_n \mapsto \mathcal{X}_{n+1}$ discussed in Section 13.1. The last two auxiliary functions needed to define `nonempty` are:

$$\mathtt{final} : \mathsf{F}\beta \to \mathtt{bool}$$
$$\mathtt{final\text{-}reach} : \mathsf{F}_{\mathsf{n}}(\mathsf{F}\beta) \to \mathtt{bool}$$

```
      final x =  forall x (member F)
final-reach X =  if (exists X final) true
                 (final-reach (X >>= conf-succ))
```

For a set `X` of configurations, the function `final-reach` checks if a configuration containing only final states is reachable from any configuration in `X`, by recursively computing successors. Finally:

```
nonempty A Q∃ Q∀ δ qI F = final-reach (just [qI])
```

Note that all auxiliary functions above must be defined in an environment that contains all ingredients of an automaton, that is A, $Q_\exists$, $Q_\forall$, δ, q_I and F.

14. Decidability border

In this section we discuss what happens to our program for data symmetries other than those mentioned in Theorem 12. We provide a negative example and a positive one. This demonstrates that decidability of the emptiness problem in dimension 1 is a delicate issue and it strongly depends on a data symmetry.

Positive example: forest orders. Consider a *forest symmetry*, which corresponds to the Fraïssé class of forests. A partial order is called a forest if for every element x, the elements smaller than x are linearly ordered. The class of finite forests has amalgamation, so there is a universal structure, and we can study Nλ in the resulting data symmetry.

One can show that the forest symmetry satisfies condition (2) from Section 13. The proof is the same as in Theorem 14, except that we use Kruskal's Tree Theorem instead of Dickson's or Higman's Lemma. Because condition (2) is satisfied, our emptiness algorithm works, and therefore emptiness is decidable for one-dimensional alternating automata for the forest symmetry. This is a new decidability result, unknown in the automata literature.

Negative example: partial orders. Consider the partial order symmetry, arising from the Fraïssé class of finite partial orders.

Fact 17 Condition (2) fails in the partial order symmetry.

In fact, emptiness is undecidable for one-dimensional alternating automata in the partial order symmetry.

15. Name binding

As another case study, we use Nλ to capture name binding on the example of untyped lambda calculus, up to α-conversion. Our objective is to define a datatype to implement lambda terms, such that two α-equivalent terms are represented by the same object of that datatype. In this section we work with the equality symmetry.

Our basic idea is that a term, in particular a lambda abstraction modulo α-conversion, is represented as a finitary set of terms. For instance, the following two terms:

$$m_1 = \lambda d.d\,d \qquad m_2 = \lambda d.\lambda e.d,$$

will be represented, intuitively, by the following two sets:

$$\{\lambda d.d\,d\}_{d\in\mathbb{D}} \qquad \{\lambda d.\{\lambda e.d\}_{e\in\mathbb{D}\setminus\{d\}}\}_{d\in\mathbb{D}}.$$

The lambda terms that we want to implement are built out of variables using application and lambda abstraction. To store them, we use a recursive datatype `term`:

$$\texttt{term} = \text{VAR}(\mathbb{D}) \mid \text{APP}(\texttt{term} \times \texttt{term}) \mid \text{ABS}(\mathsf{F_n}(\mathbb{D} \times \texttt{term}))$$

In particular, we use data values to stand for variables. We will explain the idea using m_1 and m_2. The term m_1 is represented by

```
t₁ = ABS(hull ∅ (just (d, APP(VAR(d), VAR(d)))))
```

Thus binding of a variable, say d above, in a term t, is obtained by considering the orbit of (d, t). The term $m_2' = \lambda e.d$ will be represented by

```
t₂' = ABS(hull (just d) (just (e, VAR(d))))
```

that is by the orbit of `(e, VAR(d))` with respect to $G_{\{d\}}$. Intuitively, a difference between free and bound variables is that a free variable of a term is an element of its least support while the bound variable is not. Finally m_2 is represented by:

```
t₂ = ABS(hull ∅ (just (d, t₂')))
```

The set of free variables of a term can be computed as the least support of that term, cf. Section 9.

We have decided to model lambda abstraction as a suitable one-orbit set of type $\mathsf{F_n}(\mathbb{D}\times\texttt{term})$. Note that this type contains elements that do not correspond to well-formed terms. However, one can define a function `abstr` : $\mathbb{D} \times$ `term` $\to$ `term`, that constructs a well-formed lambda abstraction up to α-conversion:

```
abstr d t =   ABS( (supp t) >>= λ fvars.
                let C = minus fvars (just d)
                in hull C (just (d, t)) )
```

where `supp` : `term` $\to \mathsf{F_n}(\mathsf{F}\mathbb{D})$ is defined as in Section 9. For instance, the terms t_1 and t_2 could be defined as:

```
t₁ =   abstr d APP(VAR(d), VAR(d))
t₂ =   abstr d (abstr e VAR(d))
```

for arbitrary `d` $\neq$ `e`. For simplicity we assume from now on that all terms are well-formed, i.e., defined using only VAR(), APP() and `abstr` as exemplified above (in fact, a well-formedness check may be easily programmed).

Now we will write a capture-avoiding substitution function:

$$\texttt{subst} : \texttt{term} \to \mathbb{D} \to \texttt{term} \to \texttt{term}$$

with the meaning that `subst t d u` substitutes the term `t` for every free occurrence of `d` in term `u`. It can be defined recursively:

```
subst t d VAR(e) = if (eq𝔻 e d) t VAR(e)
subst t d APP(t₁, t₂) =
    APP((subst t d t₁), (subst t d t₂))
subst t d ABS(X) =
    let C = fvars t in
    let X' = filter (λ(e, u).not (member C e)) X in
    ABS(map (λ(e, u).(e, subst t d u)) X')
```

Before term `t` is substituted inside a lambda abstraction `ABS(X)`, we filter out from `X` all pairs `(e, u)` such that `e` occurs free in `t`. Thus the free variables of `t` are not captured and remain free.

We should note that the code of the basic programs above is rather unpleasant, especially when compared to the neat treatment of α-conversion in Fresh O'Caml [17, 18]. Also the fact that the property of being a well-formed lambda term is not captured by the type system of Nλ, is a deficiency. We see this case study as evidence of the limitations of Nλ, just as the automata-theoretic examples of the preceding sections are evidence of its strength. A detailed study of relations between Nλ and Fresh O'Caml, as well as an attempt to combine their strong points, is left for future work.

References

[1] P. Aziz Abdulla, J. Deneux, J. Ouaknine, and J. Worrell. Decidability and complexity results for timed automata via channel machines. In *ICALP*, pages 1089–1101, 2005.

[2] M. Bojańczyk. Data monoids. In *STACS*, 2011.

[3] M. Bojańczyk, B. Klin, and S. Lasota. Languages with group actions. In *LICS*, 2011.

[4] J. A. Brzozowski and E. L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980.

[5] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[6] S. Demri and R. Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3), 2009.

[7] G. L. Ferrari, U. Montanari, and M. Pistore. Minimizing transition systems for name passing calculi: A co-algebraic formulation. In *FoSSaCS*, volume 2303 of *LNCS*, pages 129–158, 2002.

[8] D. Figueira, P. Hofman, and S. Lasota. Relating timed and register automata. In *Proc. EXPRESS'10*, volume 41 of *EPTCS*, pages 61–75, 2010.

[9] R. Fraïssé. *Theory of relations*. North-Holland, 1953.

[10] N. Francez and M. Kaminski. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

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

[12] S. Lasota and I. Walukiewicz. Alternating timed automata. In *FoSSaCS*, pages 250–265, 2005.

[13] S. Lasota and I. Walukiewicz. Alternating timed automata. *ACM Trans. Comput. Log.*, 9(2), 2008.

[14] U. Montanari and M. Pistore. History-dependent automata: An introduction. In *SFM*, pages 1–28, 2005.

[15] J. Ouaknine and J. Worrell. On the decidability of metric temporal logic. In *LICS*, pages 188–197, 2005.

[16] M. Pistore. *History Dependent Automata*. PhD thesis, University of Pisa, 1999.

[17] M. R. Shinwell. The Fresh Approach: functional programming with names and binders. Technical Report UCAM-CL-TR-618, University of Cambridge, Computer Laboratory, February 2005.

[18] M. R. Shinwell and A. M. Pitts. Fresh Objective Caml user manual. Technical Report UCAM-CL-TR-621, University of Cambridge Computer Laboratory, February 2005.