Leaving the Nest: Nominal Techniques for Variables with Interleaving Scopes

Murdoch J. Gabbay[1], **Dan R. Ghica**[2], **and Daniela Petrişan**[3]

1 **Heriot-Watt University, U.K.**
2 **University of Birmingham, U.K.**
3 **Radboud University, The Netherlands**

Abstract

We examine the key syntactic and semantic aspects of a nominal framework allowing scopes of name bindings to be arbitrarily interleaved. Name binding (e.g. $\lambda x.M$) is handled by explicit name-creation and name-destruction brackets (e.g. $\langle xMx\rangle$) which admit interleaving. We define an appropriate notion of alpha-equivalence for such a language and study the syntactic structure required for alpha-equivalence to be a congruence. We develop denotational and categorical semantics for dynamic binding and provide a generalised nominal inductive reasoning principle. We give several standard synthetic examples of working with dynamic sequences (e.g. substitution) and we sketch some preliminary applications to game semantics.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory

Keywords and phrases nominal sets, scope, alpha equivalence, dynamic sequences

Digital Object Identifier 10.4230/LIPIcs.CSL.2015.374

1 Introduction and motivation

In the syntax of formal languages it is common to see names created by locally-scoped operators such as $\lambda a.(st)$ and $\forall a.(\phi{\Rightarrow}\psi)$. The binders λ and $\forall$ have scope from the left-hand (to the matching right-hand) and scope is determined at the site of binding in the structure of the term. However, dynamically-scoped binding is an often encountered phenomena occurring whenever resources are allocated and freed explicitly. The most common situation is that of memory in C-like languages, but this also applies to other resources such as opening and closing files or network sockets. In general, a physical resource will be handled using a name which can be used by the program. The choice of name, between the allocation and the release of the resource, is irrelevant, leading to notions similar to binding and α-equivalence, but more finely grained, to account for possible scope interleaving.

Nominal techniques [15, 12, 24] provide a state-of-the-art formalism for reasoning about abstract syntax with *statically-scoped* binding. However, existing techniques do not accommodate syntax with *dynamically-scoped* binding. This paper addresses this issue by introducing a syntactic notion of *dynamic sequences* and suitable denotational and categorical models.

In dynamic sequences, scope is managed by name-creation and name-destruction brackets 'create a' and 'destroy a', written as $\langle a$ and $a\rangle$, respectively. These may be interleaved and need not match up; $\langle a\langle ba\rangle b\rangle$, $\langle a\langle bb\rangle a\rangle$, and indeed just $\langle a$ and $a\rangle$ are perfectly valid sequences. In the special case of a well-matched name-creation/-destruction pair the theory specialises back to something that models nominal-style atoms-abstraction. For instance, $\langle aaa\rangle$, just as the nominal atoms-abstraction $[a]a$, models the α-equivalence behaviour of $\lambda a.a$.

To define a mathematically well-behaved notion of α-equivalence, the basic notions of *free* and *bound* names require generalisation, and a notion of *freshness arity* emerges (Sec. 2), generalising the *freshness side-conditions* of nominal terms. In Sec. 3 we provide a relational semantics for dynamic sequences and in Sec. 4 we take stock of the monoid structure on dynamic sequences. We give an equational axiomatisation for a notion of *dynamic binding monoid*, that is, a monoid equipped with compatible nominal set structure and with left and right binders. We then prove that dynamic sequences form a free such dynamic binding monoid and obtain as a corollary a structural induction principle and a simpler characterisation for the α-equivalence relation.

Dynamic sequences have a 'flat' monoid structure, as opposed to the syntax trees one encounters in nominal abstract syntax. We use this 'flatness' to our advantage, as it allows us to interpret the non-hierarchical structure of interleaved scope. Thus one interpretation of dynamic sequences is as the structures used in game semantics. As an application, Sec. 6 gives a resource-sensitive formulation of pointer sequences as used in game semantics. We conclude in Sec. 7 with an overview of related work and directions for future research.

2 Dynamic sequences

2.1 Preliminaries

Let $\mathbb{A}$ be a countably infinite set of *names* or *atoms*. Given a bijection (permutation) $\pi : \mathbb{A} \to \mathbb{A}$ define its *support* by $\mathsf{supp}(\pi) = \{a \in \mathbb{A} \mid \pi a \neq a\}$. Write $\mathsf{Perm}(\mathbb{A})$ for the set of all permutations with finite support. Write ι for the *identity permutation* and $(a\ b)$ for the *swapping* or *transposition* of a and b.

If X is a set with a $\mathsf{Perm}(\mathbb{A})$-action, write this action infix as $-\cdot-$. An element $x \in X$ is *supported* by $A \subseteq \mathbb{A}$ when for all $\pi \in \mathsf{Perm}(\mathbb{A})$, $\forall a \in A. \pi a = a$ implies $\pi \cdot x = x$. Given $\pi \in \mathsf{Perm}(\mathbb{A})$ and $x \in X$, we say that π fixes x when $\pi \cdot x = x$. A *nominal set* is a set with a $\mathsf{Perm}(\mathbb{A})$-action where every element x has finite support. It is a fact that if X is a nominal set then every $x \in X$ has a least finite support, which we write $\mathsf{supp}(x)$.[1] We are interested in elements with finite support. If $a \in \mathbb{A}$ such that $a \notin \mathsf{supp}(x)$ we write $a\#x$. If X, Y are sets with $\mathsf{Perm}(\mathbb{A})$-action, call $f : X \to Y$ *equivariant* when $f(\pi \cdot x) = \pi \cdot (fx)$ for every $\pi \in \mathsf{Perm}(\mathbb{A})$ and $x \in X$. Finally, if $\phi(c)$ is a predicate on names, write $\text{И}c.\phi$ for "$\phi(c)$ holds for all but finitely many $c \in \mathbb{A}$"; this is the *NEW-quantifier* and we may read it as "for *fresh* c, $\phi(c)$". For more on the theory above see [15, 12, 24].

▶ **Definition 1.** Fix disjoint sets $\mathbb{A}$ of *atoms* and $\mathbb{K}$ of *constants*, writing $a, b, c \in \mathbb{A}$ and $k \in \mathbb{K}$. Define sets $\mathbb{T}$ of *tokens* and RSeq of *raw sequences*, writing $m \in \mathbb{T}$ and $e \in \mathsf{RSeq}$, inductively by: $m ::=\ a \mid a\rangle \mid \langle a \mid k,\ \ e ::=\ \varepsilon \mid em.$

RSeq is equivalently the set of lists of tokens and is the free monoid on $\mathbb{T}$. The set $\mathbb{K}$ can be equipped with a trivial $\mathsf{Perm}(\mathbb{A})$-action: every permutation fixes all the elements of $\mathbb{K}$. The permutation actions on $\mathbb{A}$ and $\mathbb{K}$ can be extended pointwise on the elements of raw sequences; for instance, $(c\ a) \cdot cc\rangle b\rangle a\rangle\langle c = aa\rangle b\rangle c\rangle\langle a$. Then RSeq is a nominal set and the support of a sequence is the set of names occurring in it. The set RSeq also has a monoid structure given by concatenation, which is compatible with the permutation action (monoid multiplication is equivariant and ε has empty support and thus is fixed by all permutations.)

[1] The set $\mathsf{Perm}(\mathbb{A})$ can also be seen as a nominal set with the $\mathsf{Perm}(\mathbb{A})$-action given by conjugation. The support of a permutation π is indeed the set $\mathsf{supp}(\pi)$ as defined in the previous paragraph.

Our notation and terminology suggest we should read the raw sequence $\langle aaa\rangle$ as "create a, use a (in some manner), then destroy it"—so if we assume a constant $\lambda \in \mathbb{K}$ then $\lambda\langle aaa\rangle$ shall, informally, model the syntax $\lambda a.a$.

We can call the binding of raw sequences, which we will make formal shortly, *dynamic* in the sense that scope is not determined by a single binder but by bracket-*pairs*; $\langle a$ does not 'know' where its matching $a\rangle$ is, or vice versa, and indeed $\langle a$ on its own and unpaired with any $a\rangle$ is also a valid raw sequence, as are $a\rangle$, $\langle aa\langle aaa\rangle$, and so forth.

We endow raw sequences with a binding structure using the following ideas, which will be illustrated in Ex. 6:

- *Bound:* An atom is *bound* if it is in the scope of a creation well-paired with a destruction.
- *Created:* An atom may appear following a creation operation which is not followed by a matching destruction.
- *Destructed:* Conversely, a destruction operation may appear without a matching creation.
- *Free:* An atom may be used without being created or destructed.

An atom occurrence cannot be characterised as merely 'free' or 'bound', but we need the more refined notion of *freshness arities.* We define a freshness arity as an element of a monoid $\mathcal{B}$, which we call the *binding monoid*, and which is the free monoid over carrier $\{\mathsf{c},\mathsf{f},\mathsf{d}\}$ modulo the following equations:

$$\mathsf{f}\cdot\mathsf{f} = \mathsf{f} \qquad \text{absorption} \qquad (1)$$

$$\mathsf{c}\cdot\mathsf{f} = \mathsf{c} \qquad \text{pre-absorption} \qquad (2)$$

$$\mathsf{f}\cdot\mathsf{d} = \mathsf{d} \qquad \text{post-absorption} \qquad (3)$$

$$\mathsf{c}\cdot\mathsf{d} = \varepsilon \qquad \text{cancellation} \qquad (4)$$

The freshness arity is assigned by a monoid homomorphism $\mathcal{F}_a : \mathsf{RSeq} \to \mathcal{B}$ defined by:

$$\mathcal{F}_a = \{\langle a \mapsto \mathsf{c}, \langle b \mapsto \varepsilon, a \mapsto \mathsf{f}, b \mapsto \varepsilon, a\rangle \mapsto \mathsf{d}, b\rangle \mapsto \varepsilon, k \mapsto \varepsilon\} \text{ where } (a \neq b \in \mathbb{A},\ k \in \mathbb{K}).$$

The set of finitely supported functions denoted by $\mathcal{B}^{\mathbb{A}}$ has a nominal monoid structure, with the multiplication of functions defined pointwise. The proof of the next lemmas is immediate.

▶ **Lemma 2.** *The map $\mathcal{F} : \mathsf{RSeq} \to \mathcal{B}^{\mathbb{A}}$ defined by $e \mapsto \lambda a.\mathcal{F}_a(e)$ is an equivariant monoid morphism.*

▶ **Lemma 3.** *For any $\beta \in \mathcal{B}$ there are unique $m, p \in \mathbb{N}$ and $n \in \{0,1\}$ such that $\beta =_{\mathcal{B}} \mathsf{d}^m\cdot\mathsf{f}^n\cdot\mathsf{c}^p$.*

▶ **Definition 4.** Call the unique representation of $\beta \in \mathcal{B}$ its *normal form.*

▶ **Definition 5.** Given a sequence e and a name a it is helpful to introduce some notational shortcuts regarding the arity of a in e:

- We write $a \rhd e$ when $\mathcal{F}_a e = \mathsf{d}^m\cdot\mathsf{f}^n$, for some $m, n \in \mathbb{N}$, i.e. there are no pending unmatched name creations $\langle a$ in e.
- We write $a \lhd e$ when $\mathcal{F}_a e = \mathsf{f}^n\cdot\mathsf{c}^m$, for some $m, n \in \mathbb{N}$, i.e. there are no pending unmatched destructors $a\rangle$ in e.
- We write $a \diamond e$, and call a *balanced* in e, when $a \rhd e \wedge a \lhd e$. That is, there are no un-matched a-creations or a-destructions, so any occurrence of a is, informally, either 'bound' ($\mathcal{F}_a e = \varepsilon$) or 'free' ($\mathcal{F}_a e = \mathsf{f}$) in the conventional sense.

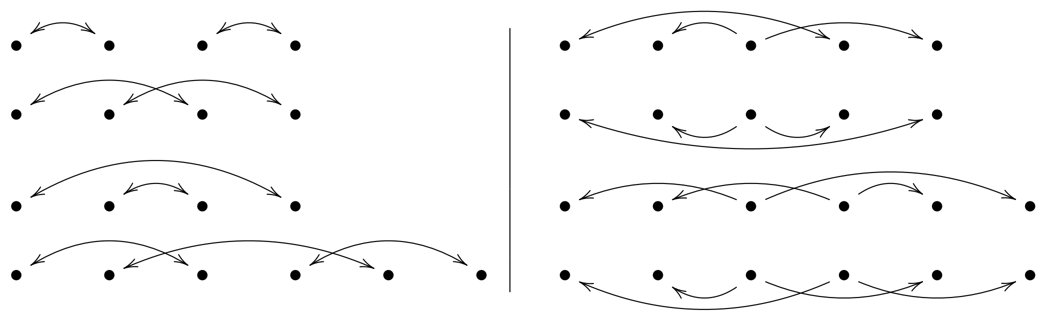


Figure 1 Pointer sequence illustration of dynamic sequences with $\mathcal{F}_a(e) = \varepsilon$.

▶ **Example 6.** Here are some 3-long sequences involving atom a, showing various arities.

$$\begin{aligned} &\mathcal{F}_a(\langle a\langle a\langle a) = \mathsf{c}^3 && \mathcal{F}_a(a\rangle\langle a\langle a) = \mathsf{dc}^2 \\ &\mathcal{F}_a(\langle aa\rangle a\rangle) = \mathsf{d} && \mathcal{F}_a(a\rangle aa\rangle) = \mathsf{d}^2 \\ &\mathcal{F}_a(a\rangle a\rangle a) = \mathsf{d}^2\mathsf{f} && \mathcal{F}_a(aaa) = \mathsf{f} \\ &\mathcal{F}_a(\langle aa\rangle a) = \mathsf{f} && \mathcal{F}_a(\langle aaa\rangle) = \varepsilon \end{aligned}$$

2.2 α-equivalence

The pairing of atom creation and atom destruction operations creates a phenomenon similar to binding. The concrete choice of a name, between its creation and its destruction, should not matter. This leads directly to a dynamic version of the α-equivalence relation, illustrated by the following examples and non-examples.

$$\langle aa\rangle\langle bb\rangle =_\alpha \langle aa\rangle\langle aa\rangle \tag{5}$$
$$\langle a\langle ba\rangle b\rangle =_\alpha \langle b\langle cb\rangle c\rangle \tag{6}$$
$$\langle a\langle aa\rangle a\rangle =_\alpha \langle b\langle cc\rangle b\rangle \tag{7}$$
$$\langle a\langle ca\rangle\langle bc\rangle b\rangle =_\alpha \langle a\langle ca\rangle\langle ac\rangle a\rangle \tag{8}$$
$$\langle a\langle bba\rangle b\rangle \neq_\alpha \langle a\langle bbb\rangle a\rangle \tag{9}$$
$$\langle a\langle babb\rangle a\rangle \neq_\alpha \langle a\langle bbab\rangle a\rangle. \tag{10}$$

These sequences, in general sequences where $\mathcal{F}_a(e) = \varepsilon$ for any atom occurring in the sequence, can be informally illustrated using "pointer sequences": a node with a left-pointing arrow corresponds to a name creation, one with a right-pointing arrow to a name destruction, and an arrow-less dot to a name mention. Thus the pairs of α-equivalent sequences in (5)–(8) can be represented as the pointer sequences on the left column in Fig. 1, while the inequivalent pairs of sequences in (9) and (10) are represented on the right column in Fig. 1.

We now give a syntax-directed definition of α-equivalence.

▶ **Definition 7.** Define *alpha-equivalence* $=_\alpha \subseteq \mathsf{RSeq}\times\mathsf{RSeq}$ inductively by:

$$\frac{}{\varepsilon =_\alpha \varepsilon}\,(\alpha\varepsilon) \qquad \frac{e_1 =_\alpha e_2 \quad m \in \mathbb{T}}{e_1 m =_\alpha e_2 m}\,(\alpha\mathbf{m})$$

$$\frac{\text{И}c.\, e_1\langle c\,(c\ a)\cdot e_2 =_\alpha e'_1\langle c\,(c\ b)\cdot e'_2 \quad a \mathrel{\Diamond} e_2, b \mathrel{\Diamond} e'_2}{e_1\langle a e_2 a\rangle =_\alpha e'_1\langle b e'_2 b\rangle}\,(\alpha\alpha)$$

(In $(\alpha\alpha)$ $(c\ a)\cdot e_2$ denotes the action of the permutation $(c\ a)$ on e_2, and similarly for $(c\ b)\cdot e'_2$.)

For other characterisations of $=_\alpha$ see Thm. 18 and Cor. 24.

The next two lemmas are instrumental in establishing that $=_\alpha$ is an equivalence relation and a congruence.

▶ **Lemma 8.** *If* $e_1 =_\alpha e_2$ *then* $\mathcal{F}(e_1) = \mathcal{F}(e_2)$.

▶ **Lemma 9.** $e_1\langle ae_2a\rangle{=}e_1'\langle ae_2'a\rangle$ *and* $a \diamond e_2, e_2'$ *imply* $e_1{=}e_1'$ *and* $e_2{=}e_2'$.

▶ **Lemma 10.** $=_\alpha$ *is an equivalence relation.*

Proof sketch. Transitivity is proved inductively using a case analysis of the possible rules ending the derivations. For example, assume that $ea\rangle =_\alpha ga'\rangle$ and $ga'\rangle =_\alpha ha''\rangle$ are both obtained using $(\alpha\alpha)$. Then we have $e = e_1\langle ae_2, g = g_1\langle a'g_2$ with $a \diamond e_2$, $a' \diamond g_2$ such that $\text{И}c.e_1\langle c\,(c\ a)\cdot e_2 =_\alpha g_1\langle c\,(c\ a')\cdot g_2$. Similarly, $g = g_1'\langle a'g_2', h = h_1\langle a''h_2$ with $a' \diamond g_2'$, $a'' \diamond h_2$ such that $\text{И}c.g_1'\langle c\,(c\ a')\cdot g_2' =_\alpha h_1\langle c\,(c\ a'')\cdot h_2$. Using Lem. 9 it follows that $g_i = g_i'$ from which we derive $\text{И}c.e_1\langle c\,(c\ a)\cdot e_2 =_\alpha h_1\langle c\,(c\ a'')\cdot h_2$ with $a \diamond e_2$, $a'' \diamond h_2$. Thus $ea\rangle =_\alpha ha''\rangle$ is obtained using $(\alpha\alpha)$. ◀

▶ **Theorem 11** ($=_\alpha$ is a congruence)**.** *If* $e_1 =_\alpha e_1'$ *and* $e_2 =_\alpha e_2'$ *then* $e_1e_2 =_\alpha e_1'e_2'$.

Proof sketch. By induction on e_2' with the only interesting case being when the second equivalence was obtained using $(\alpha\alpha)$. In this case, we have $e_2 = g_1\langle ag_2a\rangle$ and $e_2' = g_1'\langle bg_2'b\rangle$ such that $a \diamond g_2$, $b \diamond g_2'$ and $\text{И}c.\, g_1\langle c\,(c\ a)\cdot g_2 =_\alpha g_1'\langle c\,(c\ b)\cdot g_2'$. By the induction hypothesis we have $\text{И}c.\, e_1g_1\langle c\,(c\ a)\cdot g_2 =_\alpha e_1'g_1'\langle c\,(c\ b)\cdot g_2'$, hence by $(\alpha\alpha)$ we obtain $e_1e_2 =_\alpha e_1'e_2'$. ◀

▶ **Definition 12.** Let $\mathsf{DSeq} = (\mathsf{RSeq}/{=_\alpha})$ be the nominal set of sequences quotiented by α-equivalence, with the natural permutation action given by the action on representatives; call these *dynamic sequences.*

Note that Lem. 8 ensures that we can extend the notations of $a \diamond e, a \rhd e, a \lhd e$ to dynamic sequences.

▶ **Lemma 13.** *If* $e \in \mathsf{DSeq}$ *and* $a \in \mathbb{A}$ *then* $\mathcal{F}_a e = \varepsilon$ *if and only if* $a\#e$.

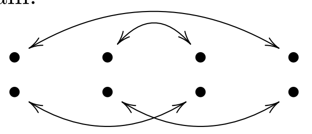
2.3 On the congruence property of α-equivalence

The congruence of α-equivalence is an essential mathematical property which has motivated design decisions in our definition of dynamic scope. We take a moment to discuss them, and so perhaps gain a better perspective on the design space in which dynamic sequences exist. Consider the raw sequence $\langle aa\langle aaa\rangle$. Which of the two occurrences of $\langle a$ should match the destructor $a\rangle$? Rem. 9 and the equations of the binding monoid (1)–(4) uniquely identify it as the most recent unpaired $\langle a$ before the $a\rangle$ (so above, the rightmost $\langle a$ matches $a\rangle$). We call this *late binding.*

Some diagrams for the slightly more complex example of $\langle a\langle aa\rangle a\rangle$ illustrate this. We prefer the upper diagram to the lower diagram:

$\langle a\langle aa\rangle a\rangle =_\alpha \langle a\langle bb\rangle a\rangle =$

$\langle a\langle aa\rangle a\rangle \neq_\alpha \langle a\langle ba\rangle b\rangle =$



The lower diagram (which we might call *early* or *eager* binding) is not obviously mathematically wrong, but it is unreasonable in the sense that it invalidates congruence of α-equivalence:

▶ Remark. For any binding policy other than late binding, any reasonably defined $=_\alpha$ is not a congruence.

Informal argument. Whatever α-equivalence is, we require $\langle aa\rangle =_\alpha \langle bb\rangle$. If $=_\alpha$ is a congruence then $\langle a\langle aa\rangle =_\alpha \langle a\langle bb\rangle$. Given a binding policy which does not match a destructor for a with the most recent creation preceding it, it follows that $\langle a\langle aa\rangle =_\alpha \langle b\langle ab\rangle \neq_\alpha \langle a\langle bb\rangle$, a contradiction. ◀

Late binding preserves existing dynamic bindings whereas other binding policies do not, thus α-equivalence is a congruence with late binding (Thm. 11), whereas other dynamic binding policies are, in this sense, ill-behaved.

3 Relational semantics

We now give a concrete semantics in relations. This *relational semantics* is sound, complete, and compositional.

Call a *stack* a list of pairs of atoms, i.e., an element of the set $\mathit{Stacks} = (\mathbb{A}^2)^*$. Elements of $\mathbb{A}^2$ will be written as $(a \mapsto b)$. For each $S \in \mathit{Stacks}$ we define stack-like operations *read*, *add* and *remove*, written as $S(a)$, $S :: (a\mapsto b)$ and, respectively, $S \setminus a$. Both reading and removal of a record involve the most recent record $(a\mapsto b)$ in the stack. Formally, if $S = S_1 :: (a\mapsto b) :: S_2$ for stacks S_1, S_2, and $(a\mapsto c)$ does not occur in S_2 for any c, then $S(a) = b$ and $S\setminus a = S_1 :: S_2$. Otherwise $S(a)$ and $S\setminus a$ are undefined.

▶ **Definition 14.** Define a *relational semantics*

$$[\![-]\!] : \mathsf{RSeq} \to \mathcal{P}((\mathit{Stacks} \times (\mathbb{A} + \mathbb{K})^*)^2).$$

on raw sequences as follows:

$$[\![\varepsilon]\!] = \{((S,X),(S,X)) \mid \forall S, X\} \tag{11}$$
$$[\![ea]\!] = \{((S,X),(S',X' :: S'(a))) \mid ((S,X),(S',X')) \in [\![e]\!]\} \tag{12}$$
$$[\![ek]\!] = \{((S,X),(S',X' :: k)) \mid ((S,X),(S',X')) \in [\![e]\!]\} \tag{13}$$
$$[\![e\langle a]\!] = \{((S,X),(S' :: (a\mapsto b), X' :: b)) \mid ((S,X),(S',X')) \in [\![e]\!], b \,\#\, S', X'\} \tag{14}$$
$$[\![ea\rangle]\!] = \{((S,X),(S'\setminus a, X' :: S'(a))) \mid ((S,X),(S',X')) \in [\![e]\!]\} \tag{15}$$

In (12) and (15) it is assumed that $S'(a)$ and $S'\setminus a$ respectively are well-defined.

The intuition behind $(S,X)\,[\![e]\!]\,(S', X :: X')$ is quite operational. The stack S is to be thought of as a stack of name replacements, and the sequence X as a context in which e is interpreted. S' is an updated stack, since creation and destruction of names cause it to change and X' is a sequence which "interprets" e given the updated stack S' and the context X. Using a name a (see (12)) extends the current sequence with its stack value $S(a)$; creating a name $\langle a$ (see (14)) adds a new entry $(a\mapsto b)$ to the stack and extends the current sequence with b; destroying a name $a\rangle$ (see (15)) removes it from the stack and extends the current sequence with its dictionary value. A constant k is simply added to the sequence (see (13)).

The interpretation $[\![-]\!]$ is compositional, using pointwise relational composition $- \circ -$.

▶ **Lemma 15.** *For any sequence $e \in \mathsf{RSeq}$ and token $m \in \mathbb{T}$: $[\![em]\!] = [\![e]\!] \circ [\![m]\!]$.*

Proof. Immediate from definitions. ◀

▶ **Theorem 16.** *For any $e, e' \in \mathsf{RSeq}$, $[\![ee']\!] = [\![e]\!] \circ [\![e']\!]$*

Proof sketch. By Lemma 15 we have that for any token $m \in \mathbb{T}$ we have $[\![em]\!] = [\![e]\!] \circ [\![m]\!]$. Then we can use induction on the structure of e'. ◀

▶ **Proposition 17.** *If* $e_1, e_2 \in \mathsf{RSeq}$ *and* $m \in \mathbb{T}$ *then* $[\![em]\!] = [\![e'm]\!]$ *implies* $[\![e]\!] = [\![e']\!]$.

Proof. By Lem. 15 and simple calculations. ◀

▶ **Theorem 18.** *If* $e_1, e_2 {\in} \mathsf{RSeq}$ *then* $e_1 {=_\alpha} e_2$ *iff* $[\![e_1]\!] {=} [\![e_2]\!]$.

In view of soundness and completeness (Thm. 18) we can also use $[\![\text{-}]\!]$ to interpret dynamic sequences rather than raw sequences, i.e. $[\![\text{-}]\!] : \mathsf{DSeq} \to \mathcal{P}((\mathit{Stacks} \times (\mathbb{A} + \mathbb{K})^*)^2)$.

4 Equational axiomatisation

We give an equational axiomatisation of the interleaved dynamic binding of this paper (Def. 19). The dynamic sequences of Def. 12 form a free dynamic binding monoid (Thm. 23), and α-equivalence gets a purely equational characterisation as an equality subject to freshness-arity side-conditions (Cor. 24).

The central idea is to use a monoid structure equipped with a compatible permutation action, left and right 'binders' and a function with co-domain $\mathcal{B}^{\mathbb{A}}$ that encompasses the interleaved binding laws—which we will call the *freshness arity map*, see Def. 19 below. We will call such structures *dynamic binding monoids.*

Several approaches in the literature investigate notions of algebraic theories and equational reasoning in a nominal setting [14, 7, 6, 21]. A common denominator is that equations are presented with freshness side-conditions. For example, the η-rule in untyped λ-calculus can be captured by $a\#x \vdash \mathsf{lam}([a]\mathsf{app}(x,a)) = x$. An algebraic theory of dynamic binding monoids must interpret interleaved scope, and so the freshness side-conditions familiar from e.g. nominal unification, rewriting, and universal algebra must be suitably enriched to interpret freshness-*arity* side-conditions. Thus, some equations in Def. 19 have side-conditions on the freshness arity of variables specified using the binding monoid $\mathcal{B}$—if the reader prefers, these can also be seen as *typing* conditions.

▶ **Definition 19.** A *dynamic binding monoid* is a tuple $(M, ::, 1, \cdot, \langle, \rangle, \gamma)$ where $(M, \cdot)$ is a nominal set, $(M, ::, 1)$ is a monoid such that the binary operation is equivariant and 1 is an element of M with empty support, $\langle, \rangle : \mathbb{A} \to M$ are equivariant functions, and $\gamma : M \to \mathcal{B}^{\mathbb{A}}$ is an equivariant monoid morphism, satisfying equations:

$$\gamma_a(\langle a) = \mathsf{c}, \quad \gamma_a(a\rangle) = \mathsf{d}, \quad a\#x \vdash \gamma_a(x) = \varepsilon, \quad \text{and}$$

$$b\#m, \gamma_a(m){=}\mathsf{f}^n \vdash \langle a :: m :: a\rangle = \langle b :: (b\ a)\cdot m :: b\rangle, \quad n \in \{0,1\}. \tag{16}$$

Above, given $a \in \mathbb{A}$ we write $\gamma_a : M \to \mathcal{B}$ for the map $\lambda m.\gamma(m)(a)$, and $a\rangle$ for $\rangle$ at a (instead of $\rangle a$). We may omit the monoid multiplication $::$ when clear from the context.

▶ **Lemma 20.** *The set* DSeq *can be equipped with a dynamic binding monoid structure.*

A *morphism* between dynamic binding monoids $(M, ::, 1, \cdot, \langle, \rangle, \gamma)$ and $(M', ::, 1, \cdot, \langle', \rangle', \gamma')$ is an equivariant monoid morphism $f : M \to M'$ that preserves the left and right binders and the freshness arity map, that is, $f \circ \langle = \langle'$, $f \circ \rangle = \rangle'$, respectively $\gamma' \circ f = \gamma$.

Thus dynamic binding monoids form a category denoted by DBMon. Categories of nominal algebras described for example in [11, 21] have a forgetful functor to the category of underlying nominal sets Nom, and admit a free construction. That is, the forgetful functor from a category of nominal algebras to Nom has a left adjoint. Such a free construction allows for deriving structural induction principles in the presence of binding, see for example [24, Cor. 8.22]. Because of the side-conditions involving the freshness arities, dynamic binding monoids lie outside the scope of nominal universal algebra. If we consider as the underlying structure of a binding monoid, not only the carrier nominal set, but also the freshness arity map, we can still obtain a free construction and hence a structural induction principle, see Thm. 22 below.

We introduce the category BNom of underlying nominal sets with a freshness arity map:

▶ **Definition 21.** Let BNom be the full subcategory of the slice category $\mathsf{Nom}/\mathcal{B}^{\mathbb{A}}$ with objects pairs (X, γ) where X is a nominal set and $\gamma : X \to \mathcal{B}^{\mathbb{A}}$ is an equivariant function such that $a\#x \vdash \gamma_a(x) = \varepsilon$.

A *morphism* in BNom from (X, γ) to (X', γ') is an equivariant function $f : X \to X'$ such that $\gamma' \circ f = \gamma$.

Let the *forgetful functor* $U : \mathsf{DBMon} \to \mathsf{BNom}$ send a dynamic binding monoid $(M, ::, 1, \cdot, \langle, \rangle, \gamma)$ to (M, γ).

▶ **Theorem 22.** *The forgetful functor $U : \mathsf{DBMon} \to \mathsf{BNom}$ has a left adjoint F.*

Proof. Consider $(X, \gamma) \in \mathsf{BNom}$, the set $X + \mathbb{A} + \mathbb{A} = X \cup \{\langle a | a \in \mathbb{A}\} \cup \{a\rangle | a \in \mathbb{A}\}$. We define an equivariant map $\overline{\gamma} : X + \mathbb{A} + \mathbb{A} \to \mathcal{B}^{\mathbb{A}}$ that acts as γ on X and such that $\overline{\gamma}_a(\langle a) = \mathsf{c}$, $\overline{\gamma}_a(a\rangle) = \mathsf{d}$, $\overline{\gamma}_a(\langle b) = \varepsilon$ and $\overline{\gamma}_a(b\rangle) = \varepsilon$ for $b \neq a$. Then $\overline{\gamma}$ can be extended uniquely to a monoid morphism $\gamma^* : (X + \mathbb{A} + \mathbb{A})^* \to \mathcal{B}^{\mathbb{A}}$. Define a relation $\equiv$ on $(X + \mathbb{A} + \mathbb{A})^*$ as the congruence generated by $\langle awa\rangle = \langle b(b\ a) \cdot wb\rangle$, where $a, b \in \mathbb{A}$, $w \in (X + \mathbb{A} + \mathbb{A})^*$, $b\#w$ and $\gamma^*_a(w) \in \{\varepsilon, \mathsf{f}\}$.

Construct $F(X, \gamma)$ as a dynamic binding monoid on the carrier nominal set $(X+\mathbb{A}+\mathbb{A})^*/{\equiv}$. Left and right binders are defined in the obvious way and the freshness arity map function is induced by γ^*. It is easy to check that whenever $w \equiv w'$ then $\gamma^*(w) = \gamma^*(w')$. We must exhibit an isomorphism $\mathsf{DBMon}((X + \mathbb{A} + \mathbb{A})^*/{\equiv}, M) \cong \mathsf{BNom}((X, \gamma), (M, \gamma_M))$. Starting with a morphism $f : X \to M$ in BNom, we can uniquely extend $f + \langle + \rangle : X + \mathbb{A} + \mathbb{A} \to M$ to an equivariant monoid morphism $f_\# : (X + \mathbb{A} + \mathbb{A})^* \to M$. We have that $\gamma \circ f_\# = \gamma^*$. It follows that for every $w, w' \in (X + \mathbb{A} + \mathbb{A})^*$ such that $w \equiv w'$ then $f_\#(w) = f_\#(w')$. Hence, $f_\#$ factors through a dynamic binding monoid morphism $\overline{f} : (X + \mathbb{A} + \mathbb{A})^*/{\equiv} \to M$. Conversely, given $g \in \mathsf{DBMon}((X + \mathbb{A} + \mathbb{A})^*/{\equiv}, M)$ we consider $g^\flat \in \mathsf{BNom}((X, \gamma), (M, \gamma_M))$ given by $g^\flat(x) = g([x])$ where $[x]$ is the $\equiv$-equivalence class of x. ◀

▶ **Theorem 23.** DSeq *is the free dynamic binding monoid on* $(\mathbb{A} \cup \mathbb{K}, \gamma_{\mathbb{A}})$ *where* $\gamma_{\mathbb{A}}(a)(a) = \mathsf{f}$, $\gamma_{\mathbb{A}}(a)(b) = \varepsilon$ *for* $b \neq a$ *and* $\gamma_{\mathbb{A}}(k)(a) = \varepsilon$*.*

Proof Sketch. We have that $\mathsf{DSeq} = \mathsf{RSeq}/{=_\alpha}$ and $\mathsf{RSeq} = (\mathbb{A} \cup \mathbb{K} + \mathbb{A} + \mathbb{A})^*$. Thus it suffices that $=_\alpha$ is equal to the relation $\equiv$ described in the proof of Thm. 22. That $\equiv \subseteq =_\alpha$ follows from the proof of Lem. 20. For the other inclusion, we prove by induction on the length of e that whenever $e =_\alpha e'$ then $e \equiv e'$. If the former equivalence was derived using the rules $(\alpha\varepsilon)$ or $(\alpha\mathbf{m})$ then the proof is immediate by induction. Assume that $e =_\alpha e'$ was derived using $(\alpha\alpha)$. That is, $e = e_1\langle ae_2a\rangle$, $e' = e'_1\langle be'_2b\rangle$ such that $a \diamond e_2$, $b \diamond e'_2$ and for any fresh c we have $e_1\langle c\,(c\ a) \cdot e_2 =_\alpha e'_1\langle c\,(c\ b) \cdot e'_2$. By inductive hypothesis $e_1\langle c\,(c\ a) \cdot e_2 \equiv e'_1\langle c\,(c\ b) \cdot e'_2$ for fresh c; we must prove $e_1\langle ae_2a\rangle \equiv e'_1\langle be'_2b\rangle$.

We consider the case when $e_1\langle c\,(c\ a) \cdot e_2 = h_1\langle dhd\rangle h_2$ and $e'_1\langle c\,(c\ b) \cdot e'_2 = h'_1\langle d'(d'\ d) \cdot hd'\rangle h'_2$ such that $h_1 \equiv h'_1$, $h_2 \equiv h'_2$, $d'\#h$ and $d \diamond h$. The most interesting case is when $h = g_1\langle cg_2$.

We have that

$$\begin{array}{rcl} e_1\langle ae_2a\rangle & = & h_1\langle dg_1\langle a(a\ c)\cdot g_2d\rangle(a\ c)h_2a\rangle \\ & \equiv & h_1\langle dg_1\langle cg_2d\rangle h_2c\rangle \\ & \equiv & h_1'\langle d'(d'\ d)\cdot g_1\langle c(d'\ d)\cdot g_2d'\rangle h_2'c\rangle \\ & \equiv & h_1'\langle d'(d'\ d)\cdot g_1\langle b(b\ c)(d'\ d)\cdot g_2d'\rangle(b\ c)h_2'b\rangle \\ & = & e_1'\langle be_2'b\rangle. \end{array}$$

It might be the case that $e_1\langle c\,(c\ a)\cdot e_2 \equiv e_1'\langle c\,(c\ b)\cdot e_2'$ was obtained using the transitivity of $\equiv$, for example $\langle d_1\langle d_2\langle cd_2\rangle d_1\rangle \equiv \langle d_1'\langle d_2'\langle cd_2'\rangle d_1'\rangle$ can only be derived using the transitivity of $\equiv$. For this case the proof that $e_1\langle ae_2a\rangle \equiv e_1'\langle be_2'b\rangle$ requires several steps and transitivity, but it reduces to the basic case resolved above. ◀

From the proof of Thm. 23 we obtain a new characterisation of $=_\alpha$ from Def. 7:

▶ **Corollary 24.** *The α-equivalence relation $=_\alpha$ on* RSeq *is the least congruence closed under the following rule ($a \diamond e$ is from Definition 5):*

$$\frac{b\#e \quad a\diamond e}{\langle aea\rangle =_\alpha \langle b\,(b\ a)\cdot eb\rangle}\ (\alpha).$$

Proof. From the proof of Thm. 23 it follows that α-equivalence on RSeq is equal to a relation $\equiv$, defined as the least congruence closed under the rule (α). ◀

5 Examples

In the examples to follow we see our formalism at work. These examples are elementary and, because of the way our framework is set up, the definitions are suitably simple.

Def. 12 defines a data type DSeq as the quotient of an inductive data type by an α-equivalence relation. We can reason on it by taking representatives of equivalence classes and working inductively on those representatives. This comes from how we define the set.

The reader familiar with nominal techniques might ask why we do not use *nominal abstract syntax* [15], where α-equivalence is built into the inductive definition of the data type at every inductive stage (using *atoms-abstraction*; a type constructor naturally present in the nominal universe), so we can work inductively up-to-α with no need for representatives.

That is impossible for us here because by design we do not know *a priori* when we write $\langle a$ in a dynamic sequence where (if anywhere) the matching $a\rangle$ will occur, and conversely, if we find $a\rangle$ in a dynamic sequence then we do not *a priori* know where (if anywhere) a matching $\langle a$ will occur.

Instead we will use a technique from [10] which allows us to lift function definitions from raw terms (in our case: raw sequences) to α-equivalence classes of raw terms (in our case: dynamic sequences). The required condition for this method to work is that what we call α-equivalence has the property of being *Barendregt-abstractive*; that every equivalence class must contain a member with maximum support. In our case it means that in the set of all raw sequences that represent the same dynamic sequence, there is one with a maximum number of atoms. We may call a representative of such an orbit a *Barendregt representative*, due to the intended similarity with the *Barendregt variable naming convention* [2]. Def. 25 will help us to calculate Barendregt representatives:

▶ **Definition 25.** Suppose $e \in$ RSeq is a raw sequence and $N \subseteq \mathbb{A}$ is a finite set of atoms. Define a function *freshen*(N, e) : RSeq $\to$ RSeq inductively by:

- $\mathit{freshen}(N, \varepsilon) = \varepsilon$,
- $\mathit{freshen}(N, e_1\langle a e_2 a\rangle) = \mathit{freshen}(N, e_1)\langle c\, \mathit{freshen}(N, (c\, a)\cdot e_2)\, c\rangle$ for a fresh $c \in \mathbb{A}$ (so $c\#e_1, e_2$ and $c \notin N$) provided that $a \diamond e_2$ holds, and
- $\mathit{freshen}(N, em) = \mathit{freshen}(N, e)m$ otherwise.

It is easy to check that $\mathit{freshen}(N, e)$ is well-defined, $\mathit{freshen}(N, e) =_\alpha e$, it has maximal support, and fresh atoms are distinct from N; the reason for this last condition will become clear in a moment.

It is convenient now to make a notational distinction between $e \in \mathsf{RSeq}$ and its equivalence class $[e]_\alpha \in \mathsf{DSeq}$. We have:

▶ **Lemma 26.** *Given $k, l \geq 0$, the map $\alpha(k, l) : (\mathsf{RSeq}^k \times \mathbb{A}^l) \to (\mathsf{DSeq}^k \times \mathbb{A}^l)$ defined by*

$$(e_1, \dots, e_k, n_1, \dots, n_l) \longmapsto ([e_1]_\alpha, \dots, [e_k]_\alpha, n_1, \dots, n_l)$$

is Barendregt-abstractive.

Proof. We must construct a Barendregt representative of $([e_1]_\alpha, \dots, [e_k]_\alpha, n_1, \dots, n_l)$. Write $N = \{n_1, \dots, n_l\}$. Then we take $e'_1 = \mathit{freshen}(N, e_1)$, and $e'_2 = \mathit{freshen}(N \cup \mathsf{supp}(e'_1), e_2)$, and $e'_3 = \mathit{freshen}(N \cup \mathsf{supp}(e'_1) \cup \mathsf{supp}(e'_2), e_3)$, and so on. It clear that $(e'_1, \dots, e'_k, n_1, \dots, n_l)$ has maximal support and that it is a Barendregt representative of (the inverse image of) $([e_1]_\alpha, \dots, [e_k]_\alpha, n_1, \dots, n_l)$. Write this representative $\mathit{freshen}(e_1, \dots, e_k, n_1, \dots, n_l)$. ◀

▶ **Definition 27** ([22]). Suppose X and Y are nominal sets and $f : Y \to X$ is a function. Define $\mathit{bv}_f(y) = \mathsf{supp}(y) \setminus \mathsf{supp}(f(y))$.

For instance $\mathit{bv}_{\alpha(1,0)}(\langle aa\rangle) = \{a\}$.

Lem. 26 allows us to tailor [10, Thm. 27] to functions taking k dynamic sequences and l atoms as input (all our examples below will have this form) as follows:

▶ **Theorem 28.** *Suppose X is a nominal set and $k, l \geq 0$ and $F : \mathsf{RSeq}^k \times \mathbb{A}^l \to X$, and suppose for every $e_1, \dots, e_k \in \mathsf{RSeq}$ and $n_1, \dots, n_l \in \mathbb{A}$ that*

$$\mathit{bv}_{\alpha(k,l)}(\mathit{freshen}(e_1, \dots, e_k, n_1, \dots, n_l))\#F(\mathit{freshen}(e_1, \dots, e_k, n_1, \dots, n_l)).$$

Then the map $\text{И}F : \mathsf{DSeq}^k \times \mathbb{A}^l \to X$ defined by

$$\text{И}F([e_1]_\alpha, \dots, [e_k]_\alpha, n_1, \dots, n_l) = F(\mathit{freshen}(e_1, \dots, e_k, n_1, \dots, n_l))$$

is well-defined.

Proof. From Lem. 26 and [10, Thm. 27]. ◀

We specialise Thm. 28 to $k{=}l{=}1$ for illustration's sake: $\text{И}F([e]_\alpha, n)$ is equal to $F(e', n)$ where bound atoms in e' are chosen distinct and not equal to n, and $\text{И}F([\langle aa\rangle]_\alpha, a) = F(\langle bb\rangle, a)$. An equivalent phrasing of the condition in Thm. 28 is this:

$$\mathsf{supp}(F(\mathit{freshen}(e_1, \dots, e_k, n_1, \dots, n_l))) \subseteq \mathsf{supp}([e_1]_\alpha, \dots, [e_k]_\alpha, n_1, \dots, n_l).$$

When we use Thm. 28 we will tend to write $\text{И}F$ just as F, thus, notationally identifying the function-on-α-equivalence-classes with the function-on-representatives. We obfuscate the distinction between e and $[e]_\alpha$ and write our definitions 'as if' they were by induction on dynamic sequences. Doing this is consistent with informal practice: for instance, we are used to writing $\mathit{size}(\lambda a.a)$ and saying "size of $\lambda a.a$" but actually meaning "pick a representative and calculate the size of that representative". Thus, the reader who cares about such things can unpick this obfuscation back to the raw sequences and maximally distinct representatives; the reader who does not care, should be able to read the text just as they would any 'inductive' definition on syntax quotiented by α-equivalence.

5.1 Operations on dynamic sequences

▶ **Example 29** (Counting name creation-destruction pairs)**.** Define a function $|\text{-}| : \mathsf{DSeq} \to \mathbb{N}$ using Thm. 28 by:

$$
\begin{aligned}
|\varepsilon| &= 0 & a \rhd e \Rightarrow |ea\rangle| &= |e|.\\
|e\langle a| &= |e| & a \diamond e' \Rightarrow |e\langle ae'a\rangle| &= |ee'| + 1\\
|ek| &= |e|
\end{aligned}
$$

To apply Thm. 28 we must check that any maximally distinct choice of bound names in the argument is fresh for the result. This is indeed the case (since $a\#n$ for any atom and any $n{\in}\mathbb{Z}$). $|e|$ counts the number of pairs of matched creation-destructors in e. The side-conditions ensure that the clauses pick out the correct creation for each destructor. We calculate $|\text{-}|$ for an example sequence; in this example we mark instances of the atom a with subscripts to see how the answer is calculated (so a_1 and a_2 are the same atom; just different instances):

$$|\langle a_1a_2\langle a_3a_4a_5\rangle a_6\rangle| = |a_2\langle a_3a_4a_5\rangle| + 1 = |a_2a_4| + 2 = 2.$$

In fact, the side-condition $a \diamond e'$ is superfluous, but it ensures that brackets are consumed in well-matched pairs.

▶ Remark. The clauses above actually define an inductive function on raw sequences. Thm. 28 gives us freshness-based conditions to verify that this induces a function on dynamic sequences (formally, $\text{И}|e|$).

Function $|e|$ happens to make sense for all raw sequences whether bound names are maximally distinct or not; for an example of where this not the case, see Ex. 31.

▶ **Example 30** (Counting bound occurrences)**.** Define a function $\|\text{-}\| : \mathsf{DSeq} \to \mathbb{N}$ using Thm. 28 as follows:

$$
\begin{aligned}
\|\varepsilon\| &= 0 & a\#e' \Rightarrow \|e\langle ae'a\rangle\| &= \|ee'\|\\
\|ek\| &= \|e\| & a\#e', a \diamond e'' \Rightarrow \|e\langle ae'ae''a\rangle\| &= \|e\langle ae'e''a\rangle\| + 1\\
\|e\langle a\| &= \|e\| & a \rhd e \Rightarrow \|ea\rangle\| &= \|e\|.
\end{aligned}
$$

To apply Thm. 28 we must check that any maximally distinct choice of bound names in the argument is fresh for the result. This is indeed the case.

$\|ek\|$ counts how many names occur 'bound' in a dynamic sequence, i.e. between a matched pair of a creation and destructor. For example $\|a\langle aaa\rangle a\| = 1$ because there is only one occurrence of a between its creation and destruction. Two other occurrences of a are outside the scope. For the same example sequence as above we have:

$$\|\langle a_1a_2\langle a_3a_4a_5\rangle a_6\rangle\| = \|\langle a_1\langle a_3a_4a_5\rangle a_6\rangle\|{+}1 = \|\langle a_3a_4a_5\rangle\|{+}1 = \|\langle a_3a_5\rangle\|{+}2 = \|\varepsilon\|{+}2 = 2$$

The side-conditions ensure that brackets get 'eaten' in well-matched pairs, and are also used to identify the first free occurrence of the bound name.

▶ **Example 31** (Capture-avoiding substitution)**.** We define $\text{-}[\text{-}/\text{-}] : \mathsf{DSeq} \times \mathbb{A} \times \mathbb{A} \to \mathsf{DSeq}$ by:

$$
\begin{aligned}
\varepsilon[a/b] &= \varepsilon & c{\neq}a \Rightarrow ec[a/b] &= e[a/b]c\\
ek[a/b] &= e[a/b]k & c{\neq}a \Rightarrow e\langle c[a/b] &= e[a/b]\langle c\\
ea[a/b] &= e[a/b]b & a \rhd e \Rightarrow ea\rangle[a/b] &= e[a/b]b\rangle\\
e\langle a[a/b] &= e[a/b]\langle b & \text{И}c.b \diamond e' \Rightarrow e\langle ce'c\rangle[a/b] &= e[a/b]\langle c(e'[a/b])c\rangle
\end{aligned}
$$

Previous examples made sense on raw sequences even if bound names were not chosen maximally distinct, but this function uses more of the power of Thm. 28, licensing us in effect to rename bound atoms and so avoid accidental name-clashes: $\langle aa\rangle[a/b] = \langle cc\rangle[a/b] = \langle cc\rangle = \langle aa\rangle$.

In fact, no rule above can be applied in $\langle aa\rangle[a/b]$, but we do not care because we only care about Barendregt representatives of triples (e, a, b) and such a representative will choose the bound names in e distinct from a and b. If we wish to notice that we are technically working with raw sequences and not dynamic sequences then we choose some junk value for the non-maximally-distinct cases.

For the final example we first introduce some notation, a regular-expression-like language for dynamic sequences.

▶ **Definition 32.** Define functions

$$+, \cdot : \mathcal{P}(\mathsf{DSeq}) \times \mathcal{P}(\mathsf{DSeq}) \to \mathcal{P}(\mathsf{DSeq})$$
$$-^* : \mathcal{P}(\mathsf{DSeq}) \to \mathcal{P}(\mathsf{DSeq}) \qquad \text{by}$$

$$e \in E + F \text{ iff } e \in E \text{ or } e \in F$$
$$e \in E \cdot F \text{ iff } \exists e_1, e_2 \in \mathsf{DSeq}.e = e_1e_2 \text{ and } e_1 \in E, e_2 \in F$$
$$e \in E^* \text{ iff } e = \varepsilon \text{ or } e \in E \cdot E^*.$$

▶ **Example 33** (Capture-avoiding interleaving)**.** Another phenomenon resembling variable capture can occur when interleaving sequences. When we interleave a raw sequence such as $\langle aa\rangle$ with itself, we obtain the set of sequences $\{\langle aa\rangle\langle aa\rangle, \langle a\langle aa\rangle a\rangle\}$. If we represent them diagrammatically, we have the interleaving of $\begin{tikzcd}\bullet \arrow[r, bend left, <->] & \bullet\end{tikzcd}$ producing the sequences $\begin{tikzcd}\bullet \arrow[r, bend left, <->] & \bullet & \bullet \arrow[r, bend left, <->] & \bullet\end{tikzcd}$ and $\begin{tikzcd}\bullet \arrow[rrr, bend left, <->] & \bullet \arrow[r, bend left, <->] & \bullet & \bullet\end{tikzcd}$. What happened to the sequence $\begin{tikzcd}\bullet \arrow[rr, bend left, <->] & \bullet \arrow[rr, bend left, <->] & \bullet & \bullet\end{tikzcd}$? Because the names are equal the wrong creation is 'captured' by the wrong destructor in the course of interleaving. This can be avoided if we interleave the sequences up to α-equivalence.

Define $\parallel : \mathsf{DSeq} \times \mathsf{DSeq} \to \mathcal{P}(\mathsf{DSeq})$ using Thm. 28 and the notations from Def. 32:

$$\varepsilon \parallel e = e = e \parallel \varepsilon \qquad m\#e'm', m'\#em \Rightarrow em \parallel e'm' = (em \parallel e') \cdot m' + (e \parallel e'm') \cdot m.$$

To use Thm. 28 it suffices to check of each clause that maximally distinct choice of bound names do not affect the result, and we can assume that bound names in e_1 are chosen distinct from those in e_2, since this is a Barendregt representative of the input (e_1, e_2) to $\parallel$. Therefore

$$\langle aa\rangle \parallel \langle aa\rangle = \{\langle aa\rangle\langle aa\rangle, \langle a\langle aa\rangle a\rangle, \langle a\langle ba\rangle b\rangle\}$$

6 Application: Games with pointer sequences

In this section we sketch a potential application of dynamic sequences, a more formal and more resource-sensitive representation of pointer sequences in game semantics. Space restrictions prevent us from fully working this out but we hope it illustrates the potential of dynamic sequences to rigorously representat semantic models that require interleaved name scopes.

One of the original presentations of game semantics, by Hyland and Ong, represented *plays* as sequences of *moves* annotated with arrows between moves [18]. Formally, plays were formalised as sequences equipped with a function f from natural numbers to natural numbers indicating that the move at index n points at move at index $f(n)$. [2]

Ghica and Gabbay [9] already gave a formulation of plays using raw sequences, which turned out to streamline key definitions and simplified many proofs. This paper did not consider α-equivalence and dynamic sequences, although some of our ideas are foreshadowed.

A pointer sequence is represented diagrammatically as an ordinary sequence decorated with pointed arrows, for example

$$\begin{tikzcd}
m_0 & m_1 \arrow[l, bend right] & m_2 \arrow[ll, bend right] & m_3 \arrow[l, bend right] & m_4 \arrow[lll, bend right] & m_5 \arrow[ll, bend right]
\end{tikzcd}$$

would be represented using HO integer indices as the pair

$$(m_0 m_1 m_2 m_3 m_4 m_5, (1\mapsto 0, 2\mapsto 0, 3\mapsto 2, 4\mapsto 1, 5\mapsto 3)).$$

The raw sequence representation is $m_0[a]* :: m_1[b]a :: m_2[c]a :: m_3[d]c :: m_4[e]b :: m_5[f]d$. Each move has a name, freshly introduced, indicated in square brackets, serving as an address, and it uses a previously introduced name to indicate the point of the arrow.

Game semantics requires complex operations on pointer sequences, such as swapping moves (while preserving pointers) to model reordering of actions in asynchronous concurrency [16] or extracting sub-sequences to model restricted history sensitiveness in languages without effects (*innocence* [18]). With integer indices, the pointer map needs to be re-indexed, an awkward operation which can be formalised in principle but it never was in practice due to sheer tedium. Using names, the same definitions are straightforward, as the names stay attached to the moves, making a more precise formalisation possible.

Although the raw sequence formalisation is from a mathematical point of view effective, from a conceptual and operational point of view is too profligate in its use of names. This is best illustrated with a simple example. The standard interpretation of the sequencing operator in HO games for ALGOL-like languages [1] is the set of even-length prefixes of this pointer sequence:

$$\begin{tikzcd}
\mathsf{r} & \mathsf{r}_1 \arrow[l, bend right] & \mathsf{d}_1 \arrow[l, bend right] & \mathsf{r}_2 \arrow[lll, bend right] & \mathsf{d}_2 \arrow[l, bend right] & \mathsf{d} \arrow[lllll, bend right]
\end{tikzcd}$$

. The operational intuition is as follows, where P is 'the program' and E is 'the environment':

r E asks P to start sequencing the two commands;
r$_1$ P in reply to r (see arrow) asks E to execute the first command;
d$_1$ E in reply to r_1 (see arrow) eventually reports the first command's termination;
r$_2$ P, justified by r (see arrow), asks E to execute the second command;
d$_2$ E, in reply to r_2, eventually reports the first command's termination;
d P, in reply to r, reports that sequencing is completed.

The raw sequence representation of this play is $\mathsf{r}[a]* :: \mathsf{r}_1[b]a :: \mathsf{d}_1 b :: \mathsf{r}_2[d]a :: \mathsf{d}_2 d :: \mathsf{d}a$, which requires 3 names. Certain moves, called *answers*, are never pointed at, so they need not introduce a name (they can, but it will simply be wasted). In certain game models answers have the additional property that after they point to a move no other subsequent move can ever point to it either—like name destructors, in fact! Using dynamic sequences with explicit name creation and destruction and late binding, the same sequence can be represented as: $\mathsf{r}*\langle a :: \mathsf{r}_1 a\langle a :: \mathsf{d}_1 a\rangle :: \mathsf{r}_2 a\langle a :: \mathsf{d}_2 a\rangle :: \mathsf{d}a\rangle$. The raw sequence representative of the dynamic sequence above uses just the name a. This is more aesthetically pleasing but it is also helpful for two reasons related to representing game models for program verification

[2] This section is best understood by readers familiar with Hyland-Ong-style game semantics, but it is written so that it can be also accessible to the casual reader.

or compilation: Dynamic sequences allow an improvement of the mathematical presentation of game semantics as well, beyond the raw-sequence formalisation. For example, in [9]:

- Def. 2.12, formalises the concept of "enabled sequence" e, in which any name of a move must be introduced before being used. This definition becomes just $a \lhd e$.
- Def. 2.15(iii) "*Call a play* e strictly scoped *when* $\mathsf{a}a[b]e' \in e$ *implies* $a \notin \mathsf{supp}(e')$." says that once the "answer" move a uses a name a, that name should never be used again. This condition can be removed now and plays be made strictly scoped by construction, because a can be destructed by the answer move: $\mathsf{a}a\rangle\langle b :: e'$.

Note that, as detailed in Sec. 5, the various raw-sequence operations used in game semantics can be lifted to dynamic sequences in an elegant way.

7 Related and future work

The main contribution of this paper is the syntactic notion of *dynamic sequence* that models *interleaved* scope by splitting binding into two more primitive syntactic constructs: a *name-creation* bracket $\langle a$, and a *name-destruction* bracket $a\rangle$. By *interleaved* we mean that brackets need not be perfectly nested, as in $\langle a\langle baba\rangle b\rangle$.

The idea of splitting local binding into two brackets has been seen before. The Adbmal syntax from [17] splits λ-binding specifically in the λ-calculus into an opening bracket λa and a closing bracket $\reflectbox{$\lambda$}a$. However that paper is focused on scope-balanced terms and assumes a *jump* semantics, that is, $\reflectbox{$\lambda$}a$ closes the scope of all intermediate λs occurring before the matching λa in order to avoid interleaved scope. By contrast, in this paper $a\rangle$ lazily matches only the single most recent unmatched $\langle a$. It would be interesting to develop a categorical semantics for the $\reflectbox{$\lambda$}$-calculus and to explore further connections with dynamic sequences. It would be certainly interesting to extend the ideas of dynamic scope to trees, as Adbmal is set up to do, but this presents significant conceptual challenges, even before considering the technical ones. For example, matching brackets in a non-linear structure seems to require a notion of traversal for the structure. This remains to be investigated.

Dynamic scope also appears in natural languages, in semantic models for indefinite articles [28]. An opening bracket corresponds to the creation of a new 'file' for storing subsequent information and anchoring references. A closing bracket corresponds to the deletion of the 'file' and the destruction of the context. That paper takes a radically different approach based on a variation of monoidal categories and Grothendieck constructions. Working out the precise connection with our setting is left as future work.

In Sec. 5 we introduced a number of concepts such as regular expressions over dynamic sequences (Def. 32). Regular expressions and Kleene algebras with statically scoping nominal-style name-binding and -generation have been studied [13, 23, 20] and it would be interesting to investigate versions with dynamic scope. Languages with allocation have been extensively studied, including in the nominal setting (e.g. [3, 25]), but those with deallocation not so much as far as we know. This too could be future work. It would also be interesting to extend nominal automata [26, 4, 19] to handle name destruction. We could then investigate whether dynamic binding monoids play a similar role in understanding the algebraic theory of languages accepted by such automata, just as orbit-finite nominal monoids do for nominal languages, see [5].

Our original motivation was to apply dynamic sequences as a notation for the pointer sequences of game semantics, to simplify the formalisation of definitions of operations on pointer sequences and proofs of their properties. Raw nominal sequences are a step in this direction [9]; dynamic sequences take this further by introducing appropriate rules for scope and α-equivalence. This may help to formalise parts of game semantics—think "game semantics in Nominal Isabelle" analogously to the current extensive implementation of nominal abstract syntax in Isabelle [27], or "rewriting on game semantics using nominal rewriting" (or a suitable generalisation with freshness side-conditions generalised to freshness-arity side-conditions) similar to [8]—and to tighten the connection between the game-semantic and abstract-machine models.

Perhaps the most significant challenge, but also the most exciting opportunity, is the use of dynamic sequences to model explicit resource management in C-like languages. Intuitively, a call to `malloc()` introduces a new name for a memory location, which in a dynamic trace corresponds to $\langle a$, whereas a call to `free()` removes that name, which in a dynamic trace corresponds to $a\rangle$. Clearly the scopes of the memory locations thus managed can be arbitrarily interleaved. However, the nominal aspects are only one aspect required to understand `malloc/free`. The stateful effects, the possibility of dangling pointers and garbage require significant amounts of further work. To conclude, we believe that interleaved name scopes are an interesting phenomenon which appears in several contexts: game semantics (our initial motivation), natural languages and low-level languages with explicit resource management. However, beyond these actual and potential applications, dynamic sequences seem to also be a novel nominal phenomenon, interesting in its own right.

Acknowledgements. We thank Bertram Wheen for a partial Agda formalisation of the syntactic model.

References

1 Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3:2–14, 1996.

2 Henk P. Barendregt. *The Lambda Calculus: its Syntax and Semantics (revised ed.)*. North-Holland, 1984.

3 Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *Typed Lambda Calculi and Applications*, pages 86–101. Springer, 2005.

4 Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. Automata theory in nominal sets. *Logical Methods in Comp. Sci.*, 10(3), 2014.

5 Mikołaj Bojańczyk. Nominal monoids. *Theory of Computing Systems*, 53(2):194–222, 2013.

6 Ranald Clouston. Nominal lawvere theories: A category theoretic account of equational theories with names. *J. Comput. Syst. Sci.*, 80(6):1067–1086, 2014.

7 Ranald A. Clouston and Andrew M. Pitts. Nominal equational logic. *Electr. Notes Theor. Comput. Sci.*, 172:223–257, 2007.

8 Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting (journal version). *Information and Computation*, 205(6):917–965, June 2007.

9 Murdoch Gabbay and Dan R. Ghica. Game semantics in the nominal model. *Electr. Notes Theor. Comput. Sci.*, 286:173–189, 2012.

10 Murdoch J. Gabbay. A general mathematics of names. *Information and Computation*, 205(7):982–1011, July 2007.

11 Murdoch J. Gabbay. Nominal algebra and the HSP theorem. *Journal of Logic and Computation*, 19(2):341–367, April 2009.

12 Murdoch J. Gabbay. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 17(2):161–229, 2011.

13 Murdoch J. Gabbay and Vincenzo Ciancia. Freshness and name-restriction in sets of traces with names. In *FOSSACS*, volume 6604 of *Lec. Notes in Comp. Sci.*, pages 365–380. Springer, 2011.

14 Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: equational logic with names and binding. *Journal of Logic and Computation*, 19(6):1455–1508, December 2009.

15 Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2001.

16 Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. *Ann. Pure Appl. Logic*, 151(2-3):89–114, 2008.

17 Dimitri Hendriks and Vincent van Oostrom. adbmal. In *CADE*, volume 2741 of *Lec. Notes in Comp. Sci.*, pages 136–150. Springer, 2003.

18 J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.

19 Dexter Kozen, Konstantinos Mamouras, Daniela Petrisan, and Alexandra Silva. Nominal Kleene coalgebra. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Proc. 42nd Int. Colloq. Automata, Languages, and Programming, Part II (ICALP 2015)*, volume 9135 of *Lecture Notes in Computer Science*, pages 290–302, Kyoto, Japan, July 6–10 2015. EATCS, Springer.

20 Dexter Kozen, Konstantinos Mamouras, and Alexandra Silva. Completeness and incompleteness in nominal Kleene algebra. Technical report, Computing and Information Science, Cornell University, November 2014.

21 Alexander Kurz and Daniela Petrisan. On universal algebra over nominal sets. *Math. Struc. in Comp. Sci.*, 20(2):285–318, 2010.

22 Alexander Kurz, Daniela Petrisan, Paula Severi, and Fer-Jan de Vries. Nominal coalgebraic data types with applications to lambda calculus. *Logical Methods in Computer Science*, 9(4), 2013.

23 Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. On nominal regular languages with binders. In *FOSSACS*, volume 7213 of *Lec. Notes in Comp. Sci.*, pages 255–269. Springer, 2012.

24 Andrew M Pitts. *Nominal Sets: Names and Symmetry in Comp. Sci.*, volume 57. Cambridge University Press, 2013.

25 Mark R Shinwell and Andrew M Pitts. On a monadic semantics for freshness. *Theor. Comp. Sci.*, 342(1):28–55, 2005.

26 Nikos Tzevelekos. Fresh-register automata. In *POPL*, pages 295–306. ACM, 2011.

27 Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *J. of Automatic Reasoning*, 40(4):327–356, 2008.

28 Albert Visser, Kees Vermeulen, et al. Dynamic bracketing and discourse representation. *Notre Dame Journal of Formal Logic*, 37(2):321–365, 1996.