Leaving the nest
Gabbay, Jamie; Ghica, Dan R.; Petrisan, Daniela

Published in:
24th EACSL Annual Conference on Computer Science Logic

DOI:
10.4230/LIPIcs.CSL.2015.374

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Link to publication in Heriot-Watt University Research Portal

Citation for published version (APA):
Leaving the Nest: Nominal Techniques for Variables with Interleaving Scopes

Murdoch J. Gabbay¹, Dan R. Ghica², and Daniela Petrişan³

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. \( \langle x \rangle M x \)) 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 \( \lambda \) 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 \( \alpha \)-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 \( \langle a \rangle \) and \( a \rangle \), respectively. These may be interleaved and need not match up; \( \langle a \rangle (ba)b \), \( \langle a \rangle (bb)\rangle a \), and indeed just \( a \rangle \) 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, \( \langle a \rangle a \rangle \), just as the nominal atoms-abstraction \( [a]a \), models the \( \alpha \)-equivalence behaviour of \( \lambda a.a \).
To define a mathematically well-behaved notion of $\alpha$-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 $\alpha$-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 $A$ be a countably infinite set of names or atoms. Given a bijection (permutation) $\pi : A \to A$ define its support by $\text{supp}(\pi) = \{ a \in A \mid \pi a \neq a \}$. Write $\text{Perm}(A)$ for the set of all permutations with finite support. Write $\iota$ for the identity permutation and $(a, b)$ for the swapping or transposition of $a$ and $b$.

If $X$ is a set with a $\text{Perm}(A)$-action, write this action infix as $-\cdot-$. An element $x \in X$ is supported by $A \subseteq X$ when for all $\pi \in \text{Perm}(A)$, $\forall a \in A. \pi a = a$ implies $\pi \cdot x = x$. Given $\pi \in \text{Perm}(A)$ and $x \in X$, we say that $\pi$ fixes $x$ when $\pi \cdot x = x$. A nominal set is a set with a $\text{Perm}(A)$-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 $\text{supp}(x)$.

We are interested in elements with finite support. If $a \in A$ such that $a \notin \text{supp}(x)$ we write $a \# x$. If $X, Y$ are sets with $\text{Perm}(A)$-action, call $f : X \to Y$ equivariant when $f(\pi \cdot x) = \pi \cdot (fx)$ for every $\pi \in \text{Perm}(A)$ and $x \in X$. Finally, if $\phi(c)$ is a predicate on names, write $\forall c. \phi$ for “$\phi(c)$ holds for all but finitely many $c \in A$”; this is the NEW-quantifier and we may read it as “for fresh $c$, $\phi(c)$”.

### Definition 1

Fix disjoint sets $A$ of atoms and $K$ of constants, writing $a, b, c \in A$ and $k \in K$. Define sets $T$ of tokens and $RSeq$ of raw sequences, writing $m \in T$ and $c \in RSeq$, inductively by: $m ::= a \mid k \mid \langle a \mid k \rangle$, $c ::= e \mid em$.

$RSeq$ is equivalently the set of lists of tokens and is the free monoid on $T$. The set $K$ can be equipped with a trivial $\text{Perm}(A)$-action: every permutation fixes all the elements of $K$. The permutation actions on $A$ and $K$ can be extended pointwise on the elements of raw sequences; for instance, $(c a) \cdot \langle cc \rangle b | c = \langle ca \rangle b | c / a$. 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 (nominal multiplication is equivariant and $\varepsilon$ has empty support and thus is fixed by all permutations.)

---

1 The set $\text{Perm}(A)$ can also be seen as a nominal set with the $\text{Perm}(A)$-action given by conjugation. The support of a permutation $\pi$ is indeed the set $\text{supp}(\pi)$ 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 \( \lambda \in K \) 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; \( \langle a \rangle \) does not ‘know’ where its matching \( a \) is, or vice versa, and indeed \( \langle a \rangle \) on its own and unpaired with any \( a \) is also a valid raw sequence, as are \( \langle a \rangle \), \( \langle aa \rangle \), 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 \( B \), which we call the binding monoid, and which is the free monoid over carrier \( \{ c, f, d \} \) modulo the following equations:

\[
\begin{align*}
   f \cdot f &= f & \text{absorption} \\
   c \cdot f &= c & \text{pre-absorption} \\
   f \cdot d &= d & \text{post-absorption} \\
   c \cdot d &= \varepsilon & \text{cancellation}
\end{align*}
\]

The freshness arity is assigned by a monoid homomorphism \( F_a : \mathbb{R} \rightarrow B \) defined by:

\[
F_a = \{(a \mapsto c, (b \mapsto \varepsilon, a \mapsto f, b \mapsto \varepsilon, a) \mapsto d, b) \mapsto \varepsilon, k \mapsto \varepsilon\} \text{ where } (a \neq b \in \mathbb{A}, k \in \mathbb{K}).
\]

The set of finitely supported functions denoted by \( B^k \) has a nominal monoid structure, with the multiplication of functions defined pointwise. The proof of the next lemmas is immediate.

**Lemma 2.** The map \( \mathcal{F} : \mathbb{R} \rightarrow B^k \) defined by \( e \mapsto \lambda a.F_a(e) \) is an equivariant monoid morphism.

**Lemma 3.** For any \( \beta \in B \) there are unique \( m, p \in \mathbb{N} \) and \( n \in \{0, 1\} \) such that \( \beta = B^m.f^n.e^p \).

**Definition 4.** Call the unique representation of \( \beta \in B \) 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 \( a \triangleright e \) when \( F_a e = d^n.f^m \), for some \( m, n \in \mathbb{N} \), i.e. there are no pending unmatched name creations \( \langle a \rangle \) in \( e \).
- We write \( a \triangleleft e \) when \( F_a e = f^m.e^n \), for some \( m, n \in \mathbb{N} \), i.e. there are no pending unmatched destructors \( a \rangle \) in \( e \).
- We write \( a \triangleright e \), and call \( a \) balanced in \( e \), when \( a \triangleright e \land a \triangleleft e \). That is, there are no un-matched \( a \)-creations or \( a \)-destructions, so any occurrence of \( a \) is, informally, either ‘bound’ (\( F_a e = \varepsilon \)) or ‘free’ (\( F_a e = f \)) in the conventional sense.
Example 6. Here are some 3-long sequences involving atom \( a \), showing various arities.

\[
\begin{align*}
F_a(\langle a(a) \rangle) &= d^c \\
F_a(\langle aa(a) \rangle) &= d \\
F_a(\langle a(a) a \rangle) &= d^f \\
F_a(\langle aa \rangle) &= f
\end{align*}
\]

2.2 \( \alpha \)-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 \( \alpha \)-equivalence relation, illustrated by the following examples and non-examples.

\[
\begin{align*}
\langle aa \rangle \langle bb \rangle &= \alpha \langle aa \rangle \langle aa \rangle \\
\langle a \langle ba \rangle b \rangle &= \alpha \langle b \langle cb \rangle c \rangle \\
\langle a \langle aa \rangle a \rangle &= \alpha \langle b \langle ce \rangle b \rangle \\
\langle a \langle ca \rangle \langle bc \rangle b \rangle &= \alpha \langle a \langle ca \rangle \langle ac \rangle a \rangle \\
\langle a \langle bba \rangle b \rangle \neq \alpha \langle a \langle bbb \rangle a \rangle \\
\langle a \langle babb \rangle a \rangle \neq \alpha \langle a \langle bbb \rangle a \rangle.
\end{align*}
\]

These sequences, in general sequences where \( F_a(e) = \varepsilon \) 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 \( \alpha \)-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 \( \alpha \)-equivalence.

Definition 7. Define alpha-equivalence \( =_\alpha \subseteq \text{RSeq} \times \text{RSeq} \) inductively by:

\[
\varepsilon =_\alpha \varepsilon \\
(\alpha e) e_1 =_\alpha e_2 \text{ } \text{ } \text{ } \text{ } m \in T \text{ } \text{ } \text{ } \text{ } (\alpha m) \\

\text{In } (\alpha e) \langle c \langle a \rangle b \rangle \cdot e_2 =_\alpha e_1' \langle c \langle a \rangle b \rangle \cdot e_2' \text{ } \text{ } \text{ } \text{ } a \otimes e_2, b \otimes e_2' \text{ } \text{ } \text{ } \text{ } (\alpha e)
\]
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 $F(e_1) = F(e_2)$.
- **Lemma 9.** $e_1\langle ae_2 a\rangle = e_1'\langle ae_2' a\rangle$ 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 $(ea) =_\alpha ga'$ and $ga' =_\alpha ha''$ are both obtained using $(\alpha \alpha)$. Then we have $e = e_1\langle ae_2, g = g_1\langle a'g_2 a\rangle$ with $a \triangleleft e_2, a' \triangleleft g_2$ such that $\mathcal{Nc}. e_1\langle c(c a) \cdot e_2 =_\alpha g_1\langle c(c a') \cdot g_2 \rangle$. Similarly, $g = g_1\langle a'g_2, h = h_1\langle a''h_2 \rangle$ with $a' \triangleleft g_2, a'' \triangleleft h_2$ such that $\mathcal{Nc}. g_1\langle c(c a') \cdot g_2 =_\alpha h_1\langle c(c a'') \cdot h_2 \rangle$. Using Lem. 9 it follows that $g_1 = h_1$ from which we derive $\mathcal{Nc}. e_1\langle c(c a) \cdot e_2 =_\alpha h_1\langle c(c a'') \cdot h_2 \rangle$ with $a \triangleleft e_2, a'' \triangleleft h_2$. Thus $(ea) =_\alpha ha''$ is obtained using $(\alpha \alpha)$.

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

**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 $e_2 = g_1\langle aha \rangle$ and $e_2' = g_1'\langle bha' \rangle$ such that $a \triangleleft g_2, b \triangleleft g_2'$ and $\mathcal{Nc}. g_1\langle c(c a) \cdot g_2 =_\alpha g_1'\langle c(c b) \cdot g_2' \rangle$. By the induction hypothesis we have $\mathcal{Nc}. e_1 g_1\langle c(c a) \cdot g_2 =_\alpha e_1' g_1'\langle c(c b) \cdot g_2' \rangle$, hence by $(\alpha \alpha)$ we obtain $e_1 e_2 =_\alpha e_1' e_2'$.

- **Definition 12.** Let $\mathcal{DSeq} = (\mathcal{RSeq} / _=\alpha)$ be the nominal set of sequences quotiented by $\alpha$-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 $a \triangleleft e, a \triangleright e, a \triangleleft e$ to dynamic sequences.

- **Lemma 13.** If $e \in \mathcal{DSeq}$ and $a \in \&$ then $F ae = \varepsilon$ if and only if $a \# e$.

### 2.3 On the congruence property of $\alpha$-equivalence

The congruence of $\alpha$-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 $(aa)(aa)$. Which of the two occurrences of $\langle a \text{ should match the destructor } a \rangle$? Rem. 9 and the equations of the binding monoid (1)–(4) uniquely identify it as the most recent unpaired $\langle a \text{ before the } a \rangle$ (so above, the rightmost $\langle a \text{ matches } a \rangle$). We call this late binding.

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

\[
\begin{align*}
\langle a(aa)a \rangle &=_\alpha \langle a(bb)a \rangle = \\
\langle a(aa)a \rangle \neq_\alpha \langle a(ba)b \rangle = \\
\end{align*}
\]

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 $\alpha$-equivalence:
We now give a concrete semantics in relations. This relies on the following definitions:

**Lemma 15.** For any sequence \( e \) and token \( m \) in \( T \), the relation \( [e] \circ [m] \) is defined.

**Proof.** Immediate from definitions.

**Theorem 16.** For any \( e, e' \) in \( \text{RSeq} \), \( [e] \circ [e'] = [e \circ e'] \).

**Remark.** For any binding policy other than late binding, any reasonably defined \( =_a \) is not a congruence.

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

Late binding preserves existing dynamic bindings whereas other binding policies do not, thus \( \alpha \)-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 \( \text{Stacks} = (\mathbb{A})^* \). Elements of \( \mathbb{A}^2 \) will be written as \( (a \leftrightarrow b) \). For each \( S \in \text{Stacks} \) we define stack-like operations read, add and remove, written as \( S^{(a)} \), \( S::(a \leftrightarrow b) \) and, respectively, \( S\backslash a \). Both reading and removal of a record involve the most recent record \( (a \leftrightarrow b) \) in the stack. Formally, if \( S = S_1::(a \leftrightarrow b) :: S_2 \) for stacks \( S_1, S_2 \), and \( (a \leftrightarrow c) \) does not occur in \( S_2 \) for any \( c \), then \( S^{(a)} = b \) and \( S\backslash a = S_1::S_2 \). Otherwise \( S^{(a)} \) and \( S\backslash a \) are undefined.

**Definition 14.** Define a relational semantics

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

on raw sequences as follows:

\[
\begin{align*}
[-] = & \left\{ ((S, X), (S, X)) : \forall S, X \right\} \\
[ea] = & \left\{ ((S, X), (S', X' :: S'(a))) : ((S, X), (S', X')) \in [-] \right\} \\
[ek] = & \left\{ ((S, X), (S', X' :: k)) : ((S, X), (S', X')) \in [-] \right\} \\
[e[a]] = & \left\{ ((S, X), (S' :: (a \leftrightarrow b), X' :: b)) : ((S, X), (S', X')) \in [-] \right\} b \# S', X' \\
[ea]] = & \left\{ ((S, X), (S\backslash a, X' :: S'(a))) : ((S, X), (S', X')) \in [-] \right\}
\end{align*}
\]

In (12) and (15) it is assumed that \( S'(a) \) and \( S\backslash a \) 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 causes 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 \( a \) (see (14)) adds a new entry \( (a \leftrightarrow b) \) to the stack and extends the current sequence with \( k; \) destroying a name \( a \) (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 \( e \in \text{RSeq} \) and token \( m \in T \), \( [em] = [e] \circ [m] \).

**Proof.** Immediate from definitions.

**Theorem 16.** For any \( e, e' \in \text{RSeq} \), \( [ee'] = [e] \circ [e'] \).
Nominal Techniques for Variables with Interleaving Scopes

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 \( e' \).

\begin{itemize}
  \item **Proposition 17.** If \( e_1, e_2 \in \text{RSeq} \) and \( m \in \mathbb{T} \) then \([em] = [e'm]\) implies \([e] = [e']\].
  
  \textbf{Proof.} By Lem. 15 and simple calculations.
\end{itemize}

\begin{itemize}
  \item **Theorem 18.** If \( e_1, e_2 \in \text{RSeq} \) then \( e_1 =_o e_2 \) iff \([e_1] = [e_2]\).
  
  In view of soundness and completeness (Thm. 18) we can also use \([-] : \text{DSeq} \to \mathcal{P}((\text{Stacks} \times (\mathbb{K} + \mathbb{K']))^2)\).
\end{itemize}

\section{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 \( o \)-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 \( \mathbb{B}^a \) 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 \( \eta \)-rule in untyped \( \lambda \)-calculus can be captured by \( a \# x \vdash (a)(x) = x \). 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 \( \mathbb{B} \)—if the reader prefers, these can also be seen as typing conditions.

\begin{definition}
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 \( \gamma : M \to \mathbb{B}^a \) is an equivariant monoid morphism, satisfying equations:

\begin{align}
\gamma_a(\langle a \rangle) &= a, \quad \gamma_a(a) = d, \quad a \# x \vdash \gamma_a(x) = \varepsilon, \quad \text{and} \nonumber \\
b \# m, \gamma_a(m) &= f^n \vdash \langle a :: m :: a \rangle = \langle b :: (b a) :: m :: b \rangle, \quad n \in \{0, 1\}. \tag{16}
\end{align}

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

\begin{lemma}
The set \( \text{DSeq} \) can be equipped with a dynamic binding monoid structure.
\end{lemma}

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, \( f \circ \langle = \langle' \), \( f \circ \rangle = \gamma' \), respectively \( \gamma' \circ f = \gamma \).

Thus dynamic binding monoids form a category denoted by \( \text{DBMon} \). Categories of nominal algebras described for example in [11, 21] have a forgetful functor to the category of underlying nominal sets \( \text{Nom} \), 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 Nom/B^A with objects pairs (X, γ) where X is a nominal set and γ : X → B^A is an equivariant function such that a#x ⊢ γ_a(x) = ε.

A morphism in BNom from (X, γ) to (X', γ') is an equivariant function f : X → X' such that γ' ◦ f = γ.

Let the forgetful functor U : DBMon → BNom send a dynamic binding monoid (M, ::, 1, ·, ∪, ⟨, ⟩, γ) to (M, γ).

**Theorem 22.** The forgetful functor U : DBMon → BNom has a left adjoint F.

**Proof.** Consider (X, γ) ∈ BNom, the set X + A + A = X ∪ {⟨a⟩ | a ∈ A} ∪ {a} | a ∈ A}. We define an equivariant map π : X + A + A → B^A that acts as γ on X and such that π_a(⟨a⟩) = d, π_a(b) = ε and π_a(b) = ε for b ≠ a. Then π can be extended uniquely to a monoid morphism γ^* : (X + A + A)^* → B^A. Define a relation ⊑ on (X + A + A)^* as the congruence generated by ⟨(aaa)⟩ = (b(ba) · wb), where a, b ∈ A, w ∈ (X + A + A)^*, b#w and γ^*_a(w) ∈ {ε, f}.

Construct F(x, y) as a dynamic binding monoid on the carrier nominal set (X + A + A)^*/alysis. 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 ⊑ w' then γ^*(w) = γ^*(w'). We must exhibit an isomorphism DBMon((X + A + A)^*/nosis, M) ≅ BNom((X, γ), (M, γ_M)). Starting with a morphism f : X → M in BNom, we can uniquely extend f + ⟨⟩ : X + A + A → M to an equivariant monoid morphism f_γ : (X + A + A)^* → M. We have that γ + f_γ = γ^*. It follows that for every w, w' ∈ (X + A + A)^* such that w ⊑ w' then f_γ(w) = f_γ(w'). Hence, f_γ factors through a dynamic binding monoid morphism T : (X + A + A)^*/nosis → M. Conversely, given g ∈ DBMon((X + A + A)^*/nosis, M) we consider g^* ∈ BNom((X, γ), (M, γ_M)) given by g^*(x) = g([x]) where [x] is the η-equivalence class of x.

**Theorem 23.** DSeq is the free dynamic binding monoid on (A ∪ K, γ_k) where γ_k(a)(a) = f, γ_k(a)(b) = ε for b ≠ a and γ_k(b)(a) = ε.

**Proof Sketch.** We have that DSeq = RSeq/≡_a and RSeq = (A ∪ K + A + A)*. Thus it suffices that ≡_a is equal to the relation ≡ described in the proof of Thm. 22. That ≡_a = ≡_a follows from the proof of Lem. 20. For the other inclusion, we prove by induction on the length of e that whenever e =_a e' then e ≡ e'. If the former equivalence was derived using the rules (αe) or (αm) then the proof is immediate by induction. Assume that e =_a e' was derived using (αa). That is, e = e_1(αe_2a) · e' = e'_1(βe'_2b) such that a ◦ e_2, b ◦ e'_2 and for any fresh c we have e_1(c ⟨c a⟩ · e_2 =_a e'_1(c ⟨c b⟩ · e'_2. By inductive hypothesis e_1(c ⟨c a⟩ · e_2 ≡_a e'_1(c ⟨c b⟩ · e'_2 for fresh c; we must prove e_1(αe_2a) ≡_a e'_1(βe'_2b).

We consider the case when e_1(c ⟨c a⟩ · e_2 = h_1(dh)h_2 and e'_1(c ⟨c b⟩ · e'_2 = h'_1(d'h' d · h')h'_2 such that h_1 ∋ h'_1, h_2 ∋ h'_2, d'h and d ◦ h. The most interesting case is when h = g_1(⟨cg_2,
We have that
\[ e_1(\alpha e_2 a) = h_1(g_1(a(a c) \cdot g_2 d)(a c)h_2 a) \]
\[ \equiv h_1(g_1(cg_2 d)h_2 c) \]
\[ \equiv h'_1(d'(d' d) \cdot g_1(c(d' d) \cdot g_2 d')h'_2 c) \]
\[ \equiv h'_1(d'(d' d) \cdot g_1(b(b c)(d' d) \cdot g_2 d')(b c)h'_2 b) \]
\[ = e'_1(be'_2 b). \]

It might be the case that \( e_1(c(c c a) \cdot e_2) \equiv e'_1(c(c b) \cdot e'_2 \) was obtained using the transitivity of \( \equiv \), for example \( d_1(d_2(c d_2) d_4) \equiv (d'_1(d'_2(c d_2) d'_4) \) can only be derived using the transitivity of \( \equiv \). For this case the proof that \( e_1(\alpha e_2 a) \equiv e'_1(be'_2 b) \) 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:

\[ \text{Corollary 24. The } \alpha \text{-equivalence relation } =_\alpha \text{ on } \mathcal{RSeq} \text{ is the least congruence closed under the following rule (a } \diamond \text{ e is from Definition 5)}: \]
\[ \frac{b \# e \quad a \odot e \quad (a c a) \equiv (b (b a) \cdot e b)}{(a c a) =_\alpha (b (b a) \cdot e b)} (\alpha). \]

\[ \text{Proof. From the proof of Thm. 23 it follows that } \alpha \text{-equivalence on } \mathcal{RSeq} \text{ is equal to a relation } \equiv, \text{ defined as the least congruence closed under the rule (} \alpha). \]

\[ \text{5 \hfill Examples} \]

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 \( \mathcal{DSeq} \) as the quotient of an inductive data type by an \( \alpha \)-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 \( \alpha \text{-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-} \alpha \text{ with no need for representatives.} \]

That is impossible for us here because by design we do not know \textit{a priori} when we write \( \langle a \rangle \) in a dynamic sequence where (if anywhere) the matching \( a \) will occur, and conversely, if we find \( a' \) in a dynamic sequence then we do not \textit{a priori} know where (if anywhere) a matching \( \langle a \rangle \) 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 \( \alpha \)-equivalence classes of raw terms (in our case: dynamic sequences). The required condition for this method to work is that what we call \( \alpha \)-equivalence has the property of being \textit{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 \textit{Barendregt representative}, due to the intended similarity with the \textit{Barendregt variable naming convention} [2]. Def. 25 will help us to calculate Barendregt representatives:

\[ \text{Definition 25. Suppose } e \in \mathcal{RSeq} \text{ is a raw sequence and } N \subseteq A \text{ is a finite set of atoms. Define a function } \text{freshen}(N, e) : \mathcal{RSeq} \rightarrow \mathcal{RSeq} \text{ inductively by}: \]
We specialise Thm. 28 to writing a fresh 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 $\text{size}(\lambda a.a)$ 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 $\alpha$-equivalence.
5.1 Operations on dynamic sequences

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

\[
\begin{align*}
|\varepsilon| &= 0 \\
|\varepsilon(a)| &= |\varepsilon| \\
|\varepsilon(k)| &= |\varepsilon|
\end{align*}
\]

\[a \triangleright e \Rightarrow |e(a)| = |\varepsilon|.
\]

\[a \triangleleft e' \Rightarrow |e(ae')| = |\varepsilon'| + 1
\]

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 \(n \in \mathbb{Z}\)).

\(|\varepsilon|\) counts the number of pairs of matched creation-destructors in \(\varepsilon\). The side-conditions ensure that the clauses pick out the correct creation for each destructor. We calculate \(|-|\) 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):

\[
|\langle a_1 a_2 \langle a_3 a_4 a_5 \rangle a_6 \rangle| = |\langle a_2 \langle a_3 a_4 a_5 \rangle a_6 \rangle| + 1 = |a_2 a_4| + 2 = 2.
\]

In fact, the side-condition \(a \triangleleft e'\) 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, \(\mathcal{H}|\varepsilon|\)).

Function \(|\varepsilon|\) 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 \(|-| : \text{DSeq} \to \mathbb{N}\) using Thm. 28 as follows:

\[
\begin{align*}
|\varepsilon| &= 0 \\
|\varepsilon(k)| &= |\varepsilon| \\
|\varepsilon(a)| &= |\varepsilon|
\end{align*}
\]

\[a \# e' \Rightarrow |e(ae')| = |\varepsilon|.
\]

\[a \# e', a \triangleleft e' \Rightarrow |e(ae'ae'a)| = |e(ae'e'ae')| + 1
\]

\[a \triangleright e \Rightarrow |ea| = |\varepsilon|.
\]

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.

\(|\varepsilon(k)|\) 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:

\[
|\langle a_1 a_2 \langle a_3 a_4 a_5 \rangle a_6 \rangle| = |\langle a_1 \langle a_3 a_4 a_5 \rangle a_6 \rangle| + 1 = |\langle a_3 a_5 \rangle| + 1 = |\langle a_3 a_5 \rangle| + 2 = 2
\]

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 \([-/-] : \text{DSeq} \times \mathbb{A} \times \mathbb{A} \to \text{DSeq}\) by:

\[
\begin{align*}
\varepsilon[a/b] &= \varepsilon \\
ek[a/b] &= e[a/b]k \\
ea[a/b] &= e[a/b] \\
ea(a)[a/b] &= e[a/b](b) \\
e\langle a \rangle[a/b] &= e\langle a \rangle[b](b) \\
e\langle a \rangle[a/b] &= e\langle a \rangle[b](b)
\end{align*}
\]

\[c \neq a \Rightarrow ec[a/b] = e[a/b]c
\]

\[c \neq a \Rightarrow e\langle a \rangle[a/b] = e\langle a \rangle[c
\]

\[a \triangleright e \Rightarrow [e(a)[a/b] = e[a/b]\]

\[\forall c. b \triangleleft e' \Rightarrow e(c[e']\langle a \rangle[b] = e[a/b](c(e'[a/b])c)
\]
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: $\langle aa \rangle[a/b] = \langle cc \rangle[a/b] = \langle cc \rangle$.

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

\[ +, \cdot : \mathcal{P}(\text{DSeq}) \times \mathcal{P}(\text{DSeq}) \to \mathcal{P}(\text{DSeq}) \]

\[ -^* : \mathcal{P}(\text{DSeq}) \to \mathcal{P}(\text{DSeq}) \]

by

\[ e \in E + F \text{ iff } e \in E \text{ or } e \in F \]

\[ e \in E \cdot F \text{ iff } \exists e_1, e_2 \in \text{DSeq}. e = e_1 e_2 \text{ and } e_1 \in E, e_2 \in F \]

\[ e \in E^* \text{ iff } e = \varepsilon \text{ or } e \in E \cdot E^* \].

▶ **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$, $\langle a \langle aa \rangle a \rangle$, $\langle a \langle ba \rangle b \rangle$\}. If we represent them diagrammatically, we have the interleaving of $\bullet \bullet \rightarrow \bullet \rightarrow \bullet \leftarrow \leftarrow \bullet \bullet$ producing the sequences $\bullet \bullet \rightarrow \bullet \rightarrow \bullet \leftarrow \leftarrow \bullet \bullet$ and $\bullet \bullet \rightarrow \bullet \rightarrow \bullet \leftarrow \leftarrow \bullet \bullet$. What happened to the sequence $\bullet \bullet \bullet \rightarrow \bullet \rightarrow \bullet \leftarrow \leftarrow \bullet \bullet$? 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 $\alpha$-equivalence.

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

\[ \varepsilon \| e = e \| \varepsilon \quad m \# e' m' \text{ and } m' \# e \varepsilon \Rightarrow e m \| e' m' = (e m) \cdot e' m' + (e m) \cdot e' m' \cdot m. \]

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 $\|$. Therefore

\[ \langle aa \rangle \| \langle aa \rangle = \{ \langle aa \rangle\langle aa \rangle, \langle a\langle aa \rangle a \rangle, \langle a\langle ba \rangle b \rangle \} \]

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 represent 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 $f(n)$.  

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 $\alpha$-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

\[ m_0 \overset{\alpha}{\rightarrow} m_1 \overset{\beta}{\rightarrow} m_2 \overset{\gamma}{\rightarrow} m_3 \overset{\delta}{\rightarrow} m_4 \overset{\epsilon}{\rightarrow} m_5 \]

would be represented using HO integer indices as the pair

\[(m_0m_1m_2m_3m_4m_5, (1\mapsto0, 2\mapsto0, 3\mapsto2, 4\mapsto1, 5\mapsto3)).\]

The raw sequence representation is $m_0[a] \leftrightarrow m_1[b]a \leftrightarrow m_2[c]a \leftrightarrow m_3[d]c \leftrightarrow m_4[e]b \leftrightarrow m_5[f]d$. 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: $r \vdash E \vdash P \vdash r_1 \vdash d_1 \vdash r_2 \vdash d_2 \vdash d$. The operational intuition is as follows, where $P$ is ‘the program’ and $E$ is ‘the environment’:

- $r \vdash E \vdash P$ in reply to $r$ (see arrow) asks $E$ to execute the first command;
- $r_1 \vdash E$ in reply to $r_1$ (see arrow) eventually reports the first command’s termination;
- $r_2 \vdash P$, justified by $r$ (see arrow), asks $E$ to execute the second command;
- $d_2 \vdash E$, in reply to $r_2$, eventually reports the first command’s termination;
- $d \vdash P$, in reply to $r$, reports that sequencing is completed.

The raw sequence representation of this play is $r[a] \leftrightarrow r_1[b]a \leftrightarrow d_1[b] \leftrightarrow r_2[d]a \leftrightarrow d_2 \leftrightarrow da$, 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 $r \vdash a \leftrightarrow r_1[a] \leftrightarrow d_1[a] \leftrightarrow r_2[a] \leftrightarrow d_2[a] \leftrightarrow da$. 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 $a \prec e$.

- Def. 2.15(iii) “Call a play $e$ strictly scoped when $a e \in e \implies a \notin \text{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: $a e \lazyclose b :: e'$.

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 $a(a(ba)ba)$.

The idea of splitting local binding into two brackets has been seen before. The Admbal syntax from [17] splits $\lambda$-binding specifically in the $\lambda$-calculus into an opening bracket $\lambda a$ and a closing bracket $\lambda a$. However that paper is focused on scope-balanced terms and assumes a jump semantics, that is, $\lambda a$ closes the scope of all intermediate $\lambda$s occurring before the matching $\lambda a$ in order to avoid interleaved scope. By contrast, in this paper $a \lazyclose$ lazily matches only the single most recent unmatched $\langle a$. It would be interesting to develop a categorical semantics for the $\lambda$-calculus and to explore further connections with dynamic sequences. It would be certainly interesting to extend the ideas of dynamic scope to trees, as Admbal 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 ⟨a, whereas a call to free() removes that name, which in a dynamic trace corresponds to a⟩. 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


