Leaving the Nest: Nominal techniques for variables with interleaving scopes

Murdoch J. Gabbay\textsuperscript{1}, Dan R. Ghica\textsuperscript{2}, and Daniela Petrişan\textsuperscript{3}

\textsuperscript{1} Heriot-Watt University, UK
\textsuperscript{2} University of Birmingham, UK
\textsuperscript{3} Radboud University, 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 out some preliminary applications to game semantics and trace 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.xxx.yyy.p

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, 23] 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 $\langle a \rangle$, respectively. These may be interleaved and need not match up; $\langle a(ba) \rangle b$, $\langle a(bb) a \rangle$, and indeed just $\langle a \rangle$ and $\langle a \rangle$ 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, $\langle a(a) \rangle$, just as the nominal atoms-abstraction $[a]a$, models the $\alpha$-equivalence behaviour of $\lambda a.a$. 

© Murdoch J. Gabbay, Dan R. Ghica and Daniela Petrişan; licensed under Creative Commons License CC-BY
Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Nominal techniques for variables with interleaving scopes

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. A second application – a simple trace semantics for a low-level language with explicit resource management – is outlined in Appendix B, but had to be omitted due to space constraints. We conclude in Sec. 7 with an overview of related work and directions for future research. All omitted proofs can be found in Appendix A.

2 Dynamic sequences

2.1 Preliminaries

Let $A$ be a countably infinite set of names or atoms. Given a bijection (permutation) $\pi : A \rightarrow A$ 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 $\epsilon$ 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 $A \subseteq A$ when for all $\pi \in \text{Perm}(A)$, $\forall a \in A. \pi a = a$ implies $\pi \cdot x = x$. 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 $\text{supp}(x)$.$^1$ 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 $f : X \rightarrow Y$ equivariant when $f(\pi \cdot x) = \pi \cdot (f x)$ for every $\pi \in \text{Perm}(A)$ and $x \in X$. Finally, if $\phi(c)$ is a predicate on names, write $\mu c. \phi$ 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 $c$, $\phi(c)$”. For more on the theory above see [15, 12, 23].

Definition 1. Fix disjoint sets $a, b, c \in A$ of atoms and $k \in K$ of constants. Define sets $m \in T$ of tokens and $e \in \text{RSeq}$ of raw sequences inductively by:

$$m := a \mid a \mid \langle a \mid k \mid (a \in A, k \in K) \text{ and } e := \varepsilon \mid em$$

$\text{RSeq}$ 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, in the obvious way. Then $\text{RSeq}$ is a nominal set and the support of a sequence is the set of names occurring in it. The set $\text{RSeq}$ also has a monoid structure given by concatenation,

$^1$ 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.
which is compatible with the permutation action (monoid multiplication is equivariant and \( \varepsilon \) has empty support and thus is fixed by all permutations.)

Our notation and terminology suggest we should read the raw sequence \( \langle a a a \rangle \) as “create \( a \), use \( a \) (in some manner), then destroy it”—so if we assume a constant \( \lambda \in K \) then \( \lambda \langle a a a \rangle \) is, informally, trying to 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 \( a \rangle \) is, or vice versa, and indeed \( \langle a \rangle \) on its own and unpaired with any \( a \rangle \) is also a valid raw sequence, as are \( a \rangle, \langle a a \rangle, \langle a a a \rangle \), and so forth.

We endow raw sequences with a binding structure using the following ideas, which will be illustrated in Ex. 5:

- **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 \{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 \( F_a : \mathsf{RSeq} \to \mathcal{B} \) defined by:

\[
F_a = \{ \{ a \mapsto c, \langle b \mapsto \varepsilon, a \mapsto f, b \mapsto \varepsilon, a \mapsto d, \langle d, b \mapsto \varepsilon, k \mapsto \varepsilon \} \}
\]

where \( a \neq b \in A, k \in K \).

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

- **Lemma 2.** The map \( F : \mathsf{RSeq} \to \mathcal{B}^A \) defined by \( e \mapsto \lambda a . F_a (e) \) 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 \( \beta =_{\mathcal{B}} d^m \cdot f^n \cdot c^p \).

- **Definition 4.** Call the unique representation of \( \beta \in \mathcal{B} \) its normal form.

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 \( F_a e = d^m \cdot f^n \), for some \( m, n \in \mathbb{N} \), i.e. there are no pending unmatched name creations \( \langle a \rangle \) in \( e \).
- We write \( a \triangleright e \) when \( F_a e = f^n \cdot c^m \), for some \( m, n \in \mathbb{N} \), i.e. there are no pending unmatched destructors \( a \rangle \) in \( e \).
- We write \( a \circ e \) when \( a \triangleright e \land a \triangleright e \), i.e. there are no un-matched \( a \rangle \)-creations or \( a \rangle \)-destinations, so any occurrence of \( a \rangle \) is, informally, either ‘bound’ \( (F_a e = \varepsilon) \) or ‘free’ \( (F_a e = f) \) in the conventional sense.
4 Nominal techniques for variables with interleaving scopes

\[ e = \alpha e \\quad e_1 \alpha e_2 \quad m \in \mathbb{T} \quad (\alpha m) \]

Figure 1 Pointer sequence representation of dynamic sequences with \( F_a(e) = \varepsilon \)

Example 5. Here are some 3-long sequences involving atom \( a \), showing various arities.

\[
\begin{align*}
F_a\langle a(a(a) = c^3 & \quad F_a\langle a(a) = dc^2 \\
F_a\langle (aa)\rangle = d & \quad F_a\langle aa\rangle = d^2 \\
F_a\langle a(a)\rangle = d^2 f & \quad F_a\langle aaa\rangle = f \\
F_a\langle (aa)\rangle = f & \quad F_a\langle (aaa)\rangle = \varepsilon
\end{align*}
\]

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.

\[
\begin{align*}
\langle aa\rangle\langle bb\rangle =_\alpha \langle aa\rangle\langle aa\rangle & \quad (5) \\
\langle a(ba)\rangle =_\alpha \langle b(cb)c\rangle & \quad (6) \\
\langle a(aa)\rangle =_\alpha \langle b(cc)b\rangle & \quad (7) \\
\langle a(ca)\langle bc\rangle b\rangle =_\alpha \langle a(ca)\langle ac\rangle a\rangle & \quad (8) \\
\langle a(bba)b\rangle \neq_\alpha \langle a(bbb)b\rangle & \quad (9) \\
\langle a(bab)b\rangle \neq_\alpha \langle a(bbb)b\rangle & \quad (10)
\end{align*}
\]

These sequences, in general sequences where \( F_a(e) = \varepsilon \) for any atom occurring in the sequence, can be given a name-free representation as "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 6. Define \textbf{alpha-equivalence} \( =_\alpha \subseteq RSeq \times RSeq \) inductively by:

For other characterisations of \( =_\alpha \) see Thm. 17 and Cor. 23.

The next two lemmas (proved in Appendix A) are instrumental in establishing that \( =_\alpha \) is an equivalence relation and a congruence.
we have
with
a
with the most recent creation preceding it, it follows that 
ence then 

α
preference then 

a
call this
a
destructor
Consider the raw sequence
so perhaps gain a better perspective on the design space in which dynamic sequences exist.

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

Definition 1. Let $D_{\alpha} = (R_{\alpha}/\alpha)$ be sequences quotiented by $\alpha$-equivalence; call these dynamic sequences.

Note that Lem. 7 ensures that we can extend the notations of $a \diamond e, a \triangleright e, a \triangleleft e$ to dynamic sequences.

Lemma 7. If $e_1 =_\alpha e_2$ then $F(e_1) = F(e_2)$.

Lemma 8. $e_1\langle ae_2\rangle = e'_1\langle ae'_2\rangle$ and $a \triangleright e_2, e'_2$ imply $e_1 = e'_1$ and $e_2 = e'_2$.

Lemma 9. $=_\alpha$ is an equivalence relation.

Proof sketch. A complete proof is in Appendix A. Transitivity is proved inductively using a case analysis of the possible rules ending the derivations. For example, assume that $e_1 = \alpha g\alpha$, then

$\begin{align*}
\text{Lemma 7: } & e_1 =_\alpha e_2 \Rightarrow F(e_1) = F(e_2) \\
\text{Lemma 8: } & e_1\langle ae_2\rangle =_\alpha e'_1\langle ae'_2\rangle \text{ and } a \triangleright e_2, e'_2 \Rightarrow e_1 = e'_1 \text{ and } e_2 = e'_2 \\
\text{Lemma 9: } & =_\alpha \text{ is an equivalence relation.}
\end{align*}$

Theorem 10 ($=_\alpha$ 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 $(\alpha a)$. In this case, we have $e_2 = g_1\langle a g_2 a\rangle$ and $e'_2 = g'_1\langle b g'_2 b\rangle$ such that $a \triangleright g_2, b \triangleright g'_2$ and $\text{Lemma 1}: g_1\langle c(a) \cdot g_2 \rangle = g'_1\langle c(b) \cdot g'_2 \rangle$. By the induction hypothesis we have $\text{Lemma 2}: e_1 g_1\langle c(a) \cdot g_2 \rangle = e_1 g'_1\langle c(b) \cdot g'_2 \rangle$, hence by $(\alpha a)$ we obtain $e_1 e_2 =_\alpha e'_1 e'_2$.

Definition 11. Let $D_{\alpha} = (R_{\alpha}/\alpha)$ be sequences quotiented by $\alpha$-equivalence; call these dynamic sequences.

Note that Lem. 7 ensures that we can extend the notations of $a \diamond e, a \triangleright e, a \triangleleft e$ to dynamic sequences.

Lemma 12. If $e \in D_{\alpha}$ and $a \in A$, then $F_a e = e$ 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 $(\alpha a)(\alpha a a)$. Which of the two occurrences of $\alpha a$ should match the destructor $a$? Rem. 8 and the equations of the binding monoid (1)–(4) uniquely identify it as the most recent unpaired $\alpha a$ (before the $a$) (so above, the rightmost $\alpha a$). We call this late binding.

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

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:

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

Informal argument. Whatever $\alpha$-equivalence is, we require $\langle a a \rangle =_{\alpha} \langle b b \rangle$. If $=_{\alpha}$ is a congruence then $\langle a a \rangle =_{\alpha} \langle a b b \rangle$. Given a binding policy which does not match a destructor for $\alpha a$ with the most recent creation preceding it, it follows that $\langle a a \rangle =_{\alpha} \langle b a b \rangle$, a contradiction.
Late binding preserves existing dynamic bindings whereas other binding policies do not, thus \( \alpha \)-equivalence is a congruence with late binding (Thm. 10), 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 \( \text{Stacks} = (\mathcal{A} \times \mathbb{K})^* \). A stack \( S \in \text{Stacks} \) is used as a stack, i.e. it can be **read from** or **written to** \( S(a) \), and a record can be **added** \( S :: (a \mapsto b) \) or **removed** \( S \setminus a \). 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 13.** Define a *relational semantics* \([\cdot] : \text{RSeq} \to \mathcal{P}((\text{Stacks} \times (\mathcal{A} \times \mathbb{K})^*))^2\).

on raw sequences as follows:

\[
\begin{align*}
[\varepsilon] & = \{((S, X), (S, X)) \mid \forall S, X\} \quad (11) \\
[ea] & = \{((S, X), (S', X' :: S'(a))) \mid ((S, X), (S', X')) \in [e]\} \quad (12) \\
[ek] & = \{((S, X), (S', X' :: k)) \mid ((S, X), (S', X')) \in [e]\} \quad (13) \\
[e\{a\}] & = \{((S, X), (S' :: (a \mapsto b), X' :: b)) \mid ((S, X), (S', X')) \in [e], b \# S', X'\} \quad (14) \\
[e\{a\}^{-}] & = \{((S, X), (S' \setminus a, X' :: S'(a))) \mid ((S, X), (S', X')) \in [e]\} \quad (15)
\end{align*}
\]

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

The intuition behind \( (S, X)[\varepsilon] (S', X :: X') \) 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 \( \langle a \rangle \) (see (14)) adds a new entry \( (a \mapsto b) \) to the stack and extends the current sequence with \( b \); 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 \([\cdot] \) is compositional, using pointwise relational composition \( - \circ - \).

▶ **Lemma 14.** For any sequence \( e \in \text{RSeq} \) and token \( m \in T \): \([em] = [e] \circ [m] \).

**Proof.** Immediate from definitions. ◀

▶ **Theorem 15.** For any \( e, e' \in \text{RSeq} \), \([ee'] = [e] \circ [e'] \)

**Proof sketch.** By Lemma 14 we have that for any token \( m \in T \) we have \([em] = [e] \circ [m] \). Then we can use induction on the structure of \( e' \). ◀

▶ **Proposition 16.** If \( e_1, e_2 \in \text{RSeq} \) and \( m \in T \) then \([em] = [e'm] \) implies \([e] = [e'] \).

**Proof.** By Lem. 14 and simple calculations. ◀

▶ **Theorem 17.** If \( e_1, e_2 \in \text{RSeq} \) then \( e_1 =_x e_2 \) iff \([e_1] = [e_2] \).

In view of soundness and completeness (Thm. 17) we can also use \([\cdot] \) to interpret dynamic sequences rather than raw sequences, i.e. \([\cdot] : \text{DSeq} \to \mathcal{P}((\text{Stacks} \times (\mathcal{A} \times \mathbb{K})^*))^2\).
4 Equational axiomatisation

We give an equational axiomatisation of the interleaved dynamic binding of this paper (Def. 18). The dynamic sequences of Def. 11 form a free dynamic binding monoid (Thm. 22), and α-equivalence gets a purely equational characterisation as an equality subject to freshness-arity side-conditions (Cor. 23).

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 $B^A$ that encompasses the interleaved binding laws—which we will call the freshness arity map, see Def. 18 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 $a\#x \vdash \text{lam}([a]\text{app}(x,a)) = x$. 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. 18 have side-conditions on the freshness arity of variables specified using the binding monoid $B$—if the reader prefers, these can also be seen as typing conditions.

Definition 18. A dynamic binding monoid is a tuple $(M, ::, 1, \cdot, \zeta, \zeta, \gamma)$ 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, $\zeta, \zeta : A \to M$ are equivariant functions, and $\gamma : M \to B^A$ is an equivariant monoid morphism, satisfying equations:

$$\gamma_a(a) = e, \quad a\#x \vdash \gamma_a(x) = e, \quad \text{and} \quad b\#m, \gamma_a(m) = f^n \vdash \zeta a :: \zeta m :: a = \zeta b :: (\zeta a \cdot \zeta m :: b), \quad n \in \{0, 1\}. \quad \text{(16)}$$

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

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

Lemma 19. The set $D\text{Seq}$ can be equipped with a dynamic binding monoid structure.

Thus dynamic binding monoids form a category denoted by $DB\text{Mon}$. 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 $\text{Nom}$ has a left adjoint. Such a free construction allows for deriving structural induction principles in the presence of binding, see for example [23, 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. 21 below.

We introduce the category $\text{BNom}$ of underlying nominal sets with a freshness arity map:

Definition 20. Let $\text{BNom}$ be the full subcategory of the slice category $\text{Nom}/B^A$ with objects pairs $(X, \gamma)$ where $X$ is a nominal set and $\gamma : X \to B^A$ is an equivariant function such that $a\#x \vdash \gamma_a(x) = e$. 


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

Let the forgetful functor $U : \text{DBMon} \rightarrow \text{BNom}$ send a dynamic binding monoid $(M, \gamma)$ to $(M, \gamma)$.

**Theorem 21.** The forgetful functor $U : \text{DBMon} \rightarrow \text{BNom}$ has a left adjoint $F$.

**Proof.** Consider $(X, \gamma) \in \text{BNom}$, the set $X + A + A = X \cup \{\{a \in A\} \cup \{a \mid a \in A\}$, and the equivariant map $\tau : X + A + A \rightarrow B^A$ that acts as $\gamma$ on $X$ such that $\tau_a(\{a \in A\}) = c$, $\tau_a(a) = d$, $\tau_a(b) = \varepsilon$ and $\tau_a(a) = \varepsilon$. Then $\tau$ can be extended uniquely to a monoid morphism $\gamma^* : (X + A + A)^* \rightarrow B^A$. Define a relation $\equiv$ on $(X + A + A)^*$ as the congruence generated by $\langle a \omega a \rangle = \langle b(a b) \cdot w b \rangle$, where $a, b \in A$, $w \in (X + A + A)^*$ and $b \# a \omega a$ and $\gamma^*_a(w) \in \{\varepsilon, f\}$.

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

**Theorem 22.** $\text{DSeq}$ is the free dynamic binding monoid on $(A \cup K, \gamma_A)$ where $\gamma_A(a) = f$, $\gamma_A(a) = \varepsilon$ for $a \neq a$ and $\gamma_A(k) = \varepsilon$.

**Proof Sketch.** We have that $\text{DSeq} = \text{RSeq}/\equiv$ and $\text{RSeq} = (A \cup K + A + A)^*$. Thus it suffices that $\equiv$ is equal to the relation $\equiv$ described in the proof of Thm. 21. That $\equiv \subseteq \equiv$ follows from the proof of Lem. 19. The other inclusion is proved in Appendix A. ▲

From the proof of Thm. 22 we obtain a new characterisation of $\equiv$ from Def. 6:

**Corollary 23.** The $\alpha$-equivalence relation $\equiv_\alpha$ on $\text{RSeq}$ is the least congruence closed under the rule

$$\frac{b \# \epsilon \quad a \odot \epsilon}{\langle a e a \rangle \equiv_\alpha \langle b(a b) \cdot e b \rangle} (\alpha).$$

**Proof.** From the proof of Theorem 22 it follows that $\alpha$-equivalence on $\text{RSeq}$ is equal to a relation $\equiv$, defined as the least congruence closed under the rule $(\alpha)$. ▲

## 5 Examples

Def. 11 defines a data type $\text{DSeq}$ 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 $\alpha$-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-$\alpha$ 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 $a$ 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 the technique described in \cite{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} \cite{2}. If we make a notational distinction between $e \in \text{RSeq}$ and its equivalence class $\lbrack e \rbrack_\alpha \in \text{DSeq}$ then we have:

\begin{lemma}
The map $e \mapsto \lbrack e \rbrack_\alpha$ is \textit{Barendregt\-abstractive}.
\end{lemma}

\begin{proof}
We just need to find the Barendregt representative of $\lbrack e \rbrack_\alpha$. Consider the function $\tau : \text{RSeq} \to \text{RSeq}$ by induction on $e$ such that $\tau = e, \ e_1(e_2a) = \tau_1(e(e_2a))e_2c$ for a fresh $c \in A$ if $a \oslash e_2$, and $\tau m = \tau n$ otherwise. It is easy to check that $\tau$ is well defined, $\tau \equiv e$, and it has maximal support.
\end{proof}

To define a function on dynamic sequences it suffices to define the function on Barendregt representative raw sequences:

\begin{theorem}
Suppose $X$ is a nominal set and $F : \text{RSeq} \to X$ and suppose for every $e \in \text{RSeq}$ that $\text{supp}(\tau) \setminus \text{supp}(\lbrack e \rbrack_\alpha) \neq \emptyset$ $\Rightarrow F(\tau)$. Then the map $\text{WF} : \text{DSeq} \to X$ defined by $\text{WF}(\lbrack e \rbrack_\alpha) = F(\tau)$, is well-defined.
\end{theorem}

\begin{proof}
From Lem. 24 and \cite[Thm. 27]{10}.
\end{proof}

When we use Thm. 25 we will tend to write $\text{WF}$ just as $F$, thus, notationally identifying the function-on-\(\alpha\)-equivalence-classes with the function-on-representatives. We obfuscate the distinction between $e$ and $\lbrack e \rbrack_\alpha$ 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.

\subsection{Operations on dynamic sequences}

\begin{example}
(Counting name creation-destruction pairs). Define a function $\lbrack - \rbrack : \text{DSeq} \to \mathbb{N}$ using Thm. 25 by:

\begin{align*}
\lbrack e \rbrack &= 0 \\
\lbrack ek \rbrack &= \lbrack e \rbrack \\
\lbrack e(a) \rbrack &= \lbrack e \rbrack \\
\oslash e' \Rightarrow \lbrack e(ae') \rbrack &= \lbrack e' \rbrack + 1 \\
\oslash e \Rightarrow \lbrack ea \rbrack &= \lbrack e \rbrack.
\end{align*}
\end{example}
To apply Thm. 25 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 $|\cdot|$ 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):

$$|(a_1 a_2 (a_3 a_4 a_5) a_6)| = |a_2 (a_3 a_4 a_5)| + 1 = |a_2 a_4| + 2 = 2.$$

In fact, the side-condition $a \diamond e'$ is superfluous, but it ensures that the calculation above is calculated sensibly in the sense that brackets get ‘eaten’ in well-matched pairs.

▶ Remark. The clauses above actually define an inductive function on raw sequences. Thm. 25 gives us freshness-based conditions to verify that this induces a function on dynamic sequences (formally, $N|e|$).

Function $|\cdot|$ 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. 28.

▶ Example 27 (Counting bound occurrences). Define a function $|\cdot| : \mathbb{DSeq} \to \mathbb{N}$ using Thm. 25 as follows:

$$
\begin{align*}
|\varepsilon| &= 0 \\
|ek| &= |e|
|e(a)| &= |e| \\
 a\# a' \Rightarrow |e(a'e')| &= |e| \\
 a\# a', a \diamond e'' \Rightarrow |e(a'e''a)| &= |e(a'e'a)| + 1 \\
 a \triangleright e \Rightarrow |e(a)| &= |e|.
\end{align*}
$$

To apply Thm. 25 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 $|a\langle a\langle a\diamond a \rangle \rangle a| = 1$ 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:

$$
\begin{align*}
|\langle a_1 a_2 (a_3 a_4 a_5) a_6 \rangle| &= |\langle a_1 (a_3 a_4 a_5) a_6 \rangle| + 1 = |\langle a_3 a_4 a_5 \rangle| + 1 = |\langle a_3 a_5 \rangle| + 2 = |\varepsilon| + 2 = 2
\end{align*}
$$

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 28 (Capture-avoiding substitution). We define \([-\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·
Application: Games with pointer sequences

In this section we sketch out a potential application of dynamic sequences, a more formal and more resource-sensitive representation of pointer sequences in game semantics. Note that reasons of space prevent this application to be fully worked out, but we hope that it illustrates the potential of dynamic sequences to give a rigorous representation of semantic models where interleaved name scopes must be dealt with.

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 \( \alpha \)-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

\[
\begin{align*}
&\quad m_0 \quad m_1 \quad m_2 \quad m_3 \quad m_4 \quad m_5 \\
&\quad m_1 \quad m_2 \quad m_3 \quad m_4 \quad m_5 \quad m_6
\end{align*}
\]

would be represented using HO integer indices as the pair

\[(m_0m_1m_2m_3m_4m_5, (1\mapsto 0, 2\mapsto 0, 3\mapsto 2, 4\mapsto 1, 5\mapsto 3)).\]

The raw sequence representation is \( m_0[a] \ast :: m_1[b]a :: m_2[c]a :: m_3[d]c :: m_4[e]b :: m_5[f]d \). 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:

\[
\begin{align*}
&\quad r \quad r_1 \quad d_1 \quad r_2 \quad d_2 \quad d \\
&\quad r \quad P \quad d \quad P
\end{align*}
\]

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.

\(^2\) 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.
The raw sequence representation of this play is $r[a] :: r_1[b]a :: d_1b :: r_2[d]a :: d_2d :: da$, 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: $r*[a :: r_1a :: d_1a] :: r_2a(a :: d_2a) :: da$. 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 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 $a < e$.
- Def. 2.15(iii) “Call a play $e$ strictly scoped when $a$[$a' \in e$ implies $a \notin supp(e')$.” 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: $aa(b :: e')$.

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 $(a$, and a name-destruction bracket $a)$. By interleaved we mean that brackets need not be perfectly nested, as in $(a(baba)b)$. 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 $a$ lazily matches only the single most recent unmatched $(a$. It would be interesting to develop a categorical semantics for the $\lambda$-calculus and to explore further connections with dynamic sequences.

Dynamic scope also appears in natural languages, in semantic models for indefinite articles [27]. 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. 29) and a very simple programming language (Sec. B) with explicit deallocation. Regular expressions and Kleene algebras with statically scoping nominal-style name-binding and -generation have been studied [13, 22, 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, 24]), 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 [25, 4, 19] to handle name destruction. We could then investigate
Nominal techniques for variables with interleaving scopes

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 [26], 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.³

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


³ In Appendix B we show the semantics of a toy language where resource-names can be allocated and freed, but in a setting where stateful behaviour is extremely restricted.
A Proofs

Proof of Lem. 7. We work by induction on the length of the sequences.

1. If $e_1 = e'_1 a$ for some $e'_1 \in \text{RSeq}$ then necessarily $e_2 = e'_2 a$ for some $e'_2$ such that $e'_1 =_a e'_2$. For $b \neq a$ we have $F_a(e_1) = F_a(e'_1)$, while $F_a(e_1) = F_a(e'_1) f$, where $i \in \{1, 2\}$. Using the induction hypothesis we derive $F(e_1) = F(e_2)$.

2. The cases when $e_1 = e'_1 (a \alpha_0)$ and $e_1 = e'_1 k$ are similar.

3. If $e_1 = e'_1 a$ we need to consider two sub cases, depending whether the normal form of $F_a(e'_1)$ is of the shape $d^m a f^n$ or $d^m a f^n c p^{+1}$.

In the former situation, the equivalence between $e_1$ and $e_2$ was necessarily inferred via the rule $(\alpha a)$, so there exists $e'_2 \in \text{RSeq}$ such that $e_2 = e'_2 a$ and $e_1 = e'_2$. We conclude that $F(e_1) = F(e_2)$ just as in the previous cases.

In the latter case, the rule $(\alpha a)$ was used. So we can write $e_1 = g'(ag'\alpha a)$ and $e_2 = e'_2 (be_2^a b)$ for $g', g''\alpha \alpha_0 e_1$ such that $a \alpha \alpha_0$ and $b \alpha e_2$ and for fresh $c$ we have $g'(c(a c) a) =_a e'_2 (b c) e_2'$. By induction we have

$$F(g'(c(a c) a) = F(e'_2 (b c) e_2').)$$

(17)

We have $F_a(e_1) = F_a(g'(c) a f) \cdot F_a(g'' a f) = F_a(g'(c) a f)$. On the other hand, $F_a(e_2) = F_a(e'_2) \cdot F_a(e'_2) \cdot F_a(e'_2) \cdot F_a(e'_2)$. From (17) we deduce $F_a(g'(c) a f) = F_a(g'(c) a f)$, or equivalently, $F_a(g'(c) a f) = F_a(e'_2)$. Since $c$ is fresh, we have $F_a(g'(c) a f) =_a e_2'$, hence $F_a(e'_2) = F_a(e'_2) F_a(e'_2)$. Therefore $F_a(e_1) = F_a(e_2)$. Similarly, for any $c$ we have $F_a(e_1) = F_a(e_2)$.

Proof of Lem. 8. Assume by contradiction that $e_2$ is longer than $e'_2$. Then there exists $e_3 \in \text{RSeq}$ such that $e_2 = e_3 (ae_2')$. This implies that $F_a(e_2) = F_a(\alpha e_2')$. Since $a \alpha e_2'$ we have $F_a(\alpha e_2') = \alpha$, hence $F_a(e_2) = F_a(e_3) = \alpha$. Assuming the normal form of $F_a(e_3)$ is $f^r$ with $r \in \{0, 1\}$. Assuming the normal form of $F_a(e_3)$ is $d^m a f^n c p^{+1}$ we obtain the equality of two normal forms $f' = d^m a f^n c p^{+1}$, a contradiction. Similarly, $e_2'$ is no longer than $e_2$, so they have the same length and $e_i = e'_i$.

Proof of Lem. 9. It is easy to prove $e =_a e$ by induction on $e$; we just use $(a \alpha)$ and $(\alpha a)$.

Likewise, symmetry is easy to prove.

We now prove by induction on derivations that if $e =_a e'$ and $e' =_a e''$ then $e =_a e''$.

There are nine cases; one for each of the $3 \times 3$ possible rules ending the derivations:

- Case $(a \alpha, \ast)$ for any rule $\ast$. If $e =_a e$ and $e =_a e$ then $e =_a e$. Case $\ast, (a \alpha)$ is similar.

- Case $(\alpha a), (a \alpha)$. Suppose $e =_a e$ by $(a \alpha)$, so $e =_a e$. Using Lem. 7 if follows that $F(ea) = F(e'a) = F(e'a')$. Since $e' =_a e' e' a$ with $a \alpha e_2'$ it follows then that $e' = e'(ae_2') =_a e$ with $a \alpha e_2'$. Using Lem. 7, again, it follows that $F(e' e' a)$, so $e = e'(a e_2') =_a e'(a e_2')$. From which $\text{We}, e'(a e_2') = e'(a e_2') =_a e'(a e_2') =_a e(a e_2') = e(a e_2')$. Thus $e'(a e_2') =_a e'(a e_2') = _a e'(a e_2') =_a e'(a e_2') =_a e'(a e_2')$. Because $e(a e_2')$ reduces the case $(\alpha a), (a \alpha)$ to $(a \alpha), (a \alpha)$, proved below.

Case $(\alpha a), (a \alpha)$. If $e =_a e'$ then $e =_a e'$ and $a \alpha e_2'$ such that for all but finitely many $c$, $e_1(c(a c) a) = e_1(e_1(a c) a)$. If $e(a e_2') =_a e(a e_2')$ using $(a \alpha)$ then $e =_a e' e'(a e_2') =_a e' e'(a e_2')$. Using Lem. 7 it follows that $e'_1 =_a e'_1$, from which it follows that it must be $e'_{1,1} = e'_{2,1} = e'_1$. It follows
We claim that 

that for all but finitely many $c$, $e_1(c \cdot a) \cdot e_2 = e_1(c \cdot a') \cdot e_2' \cdot e_2''$ with $a \sqcap e_2$, $a'' \sqcap e_2''$, so $ea = e' \cdot a''$ using $(\alpha a)$.

\begin{proof}

\textbf{Proof of Lem. 12.} We show the left-to-right implication by induction on the length of the sequence. It is easier to work with a representative $e \in \mathbb{RSeq}$. We have that $a$ is fresh for the equivalence class of $e$ if and only if $\forall b \cdot (b \cdot a) \cdot e = e$, so we shall prove that whenever $F_a(e) = e$ and $b$ is fresh for $e$ then $(b \cdot a) \cdot e = e$. For $e = e$ the statement is true by definition. For the induction step we must show that for any token $m \in T$, if $F_a(em) = e$ then $(b \cdot a) \cdot em = e$. We have the following cases:

1. If $a \# m$ then $F_a m = e$ which, from Lem. 3, requires that $F_a e = e$. The induction hypothesis along with the obvious fact that $(b \cdot a) \cdot m = m$ proves the statement.

2. The case $m \in \{a, b, \{a, b\}\}$ is impossible, from Lem. 3, given that $F_a(em) = F_b(em) = e$, since the freshness arities $f, c$ cannot be cancelled into $e$ from the left.

3. If $m = a^\#$, $F_a(ca) = e$ implies $F_a e = e$, since $F_a(a) = d$ (by (4) and Lem. 3). A simple argument shows there exist sequences $e_1, e_2$ with $F_a e_1 = e$ and $a \sqcap e_2$ such that $e = e_1(a \cdot e_2)$. We must show that for fresh $b$, $(b \cdot a) \cdot e_1(a \cdot e_2) = (b \cdot a) \cdot e_1(a \cdot e_2)$. Since $b$ was chosen fresh, we may assume $b \# e_1$, and by induction hypothesis $(b \cdot a) \cdot e_1 = e_1$. For fresh $c$, $(c \cdot b)(b \cdot a) \cdot e_2 = e_2$ because the permutations agree on the support of $e$. Congruence of $\alpha$-equivalence along with $a \sqcap e_2$ proved earlier allows us to use the rule $(\alpha a)$ and conclude the proof.

The other implication can also be proved by induction.

\end{proof}

\begin{theorem}

For $e, e' \in \mathbb{RSeq}$, if $e =_\alpha e'$ then $[e] = [e']$.

\end{theorem}

\begin{proof}

By induction on the derivation of $\alpha$-equivalence. The only non-trivial computation is when $e =_\alpha e'$ was derived from the rule $(\alpha a)$. In this case we know that $e = e_1(a \cdot e_2)$ and $e' = e'_1(b \cdot e_2)$ such that $a \sqcap e_2, b \sqcap e_2'$ and for any fresh $e$ we have $e_1(c \cdot a) \cdot e_2 = e_1(c \cdot b) \cdot e_2$. By induction we have that $[e_1(c \cdot a) \cdot e_2] = [e_1(c \cdot b) \cdot e_2']$, or equivalently: $[e_1(c) \cdot (a \cdot e_2)] = [e_1(c) \cdot (b \cdot e_2')]$. From Thm. 15 and the simple observations that $(a \cdot c) \cdot c = (b \cdot c) \cdot (a \cdot c) = (b \cdot c) \cdot a$ it follows that

$[e_1(c) \cdot [(ae_2)a] = [e_1(c) \cdot (be_2b)]$. (18)

We claim that

$[(ae_2)a] = [(ae_2)a]$. (19)

From (19) and (18) we conclude that $[e] = [e']$. Claim (19) follows by induction and makes use of $a \sqcap e_2$. It suffices to prove for fresh $c$ that $[\langle ae_2a \rangle] \subseteq [\langle c(a \cdot e_2c) \rangle]$. Assume $(S, X)[\langle ae_2a \rangle][\langle S_1, X_1 \rangle]$. Using the fact that $[\langle ae_2a \rangle] = [\langle a \rangle \cdot [e_2] \cdot [a]]$ we obtain that $S_1$ is of the form $S' \setminus a$ and $X_1 = X : b : X' \setminus b$, where $b$ is fresh and $S'$ and $X'$ are such that $(S' : (a \cdot b), X : b)[e_2][S', X : b : X'])$. To prove $(S, X)[\langle c(a \cdot e_2c) \rangle][\langle S_1, X_1 \rangle]$, it suffices to check

$(S' : (c, b), X : b)[\langle c(a) \cdot e_2 \rangle][S''', X : b : X']$ (20)

where $S'''$ is obtained from $S'$ by replacing the occurrence of the pair $(a \cdot b)$ by $(c \cdot b)$. If this is the case, then we can finish using the compositionality of $[\alpha]$ (Thm. 15) and that $S''' \setminus c = S' \setminus a$ and $S'''(c) = S'(a) = b$.

\end{proof}
Nominal techniques for variables with interleaving scopes

Proof of Thm. 17. The left-to-right direction is Thm. 31. So consider $e_1, e_2$ such that $e_1 = e_2 = \varepsilon$. First we note that the lengths of $e_1$ and $e_2$ must be equal, because if $(S, X) [e_1] (S', X : X')$ then the lengths of $e_1$ and $X'$ are also equal (the proof is a straightforward inductive argument on the length of $e_1$). We will use induction on the length of $X'$. The base case $\varepsilon$ is immediate, $e_1 = e_2 = \varepsilon$ and $(\alpha - \varepsilon)$.

For the inductive cases, if $X' = X'' : k$ the proof is immediate. The interesting case is $X' = X'' : b$. There are three further cases depending on which equation was used to derive $(S', X')$.

- $S' = S'' : (a \rightarrow b)$, which requires that (14) was used, so $e_1 = e_1', e_2 = e_2'$. Since $e_1 = e_2$, Prop. 16 implies that $e_1' = e_2'$. We apply the induction hypothesis and $(\alpha - \varepsilon)$.

- $S' = S + (a \rightarrow b)$ for some name $a$, but $(a \rightarrow b)$ is not at the top of the stack. This case is just like the previous except that this means (12) was used. We also use the rule $(\alpha - \varepsilon)$ together with Prop. 16 and the inductive hypothesis.

- $S' = S + (a \rightarrow b)$ is not defined, which means that (15) was used, so it must be that $e_1 = e_1'(a_1), e_1 = e_2'(a_2)$ for some $a_1, a_2 \in A$. This is the interesting case. Using the induction hypothesis and the compositionality of $[\cdot]$ (Thm. 15), for any $(S, X)$,

\[
(S, X) [e_1] (S_1, X_1) [a_1]) (S_1 \setminus a_1, X_1 : S_1(a_1)) \quad \Leftrightarrow (S, X) [e_2] (S_2, X_2) [a_2]) (S_2 \setminus a_2, X_2 : S_2(a_2))
\]

which means that $X_1 = X_2 = X', S_1 \setminus a_1 = S_2 \setminus a_2, S_1(a_1) = S_2(a_2) = b$, so we write (21) equivalently as

\[
(S, X) [e_1] (S_1, X') [a_1]) (S_1 \setminus a_1, X'' : b) \quad \Leftrightarrow (S, X) [e_2] (S_2, X') [a_2]) (S_2 \setminus a_2, X'' : b)
\]

Because $S_1$ and $S_2$ differ only in the inclusion of the $(a_1 \rightarrow b)$ and respectively $(a_2 \rightarrow b)$ and because this can only be introduced by the occurrence of name creation, it follows that $e_1 = e_1'(a_1 e_1)$ and $e_2 = e_2'(a_2 e_2)$. Moreover, $e_1 e_2$ and $e_2 e_2$ may not have occurrences of $a_1$ and $a_2$, respectively, because $S_1(a_1)$ and $S_2(a_2)$ must be defined. Therefore

\[
a_1 \odot e_1 e_2, a_2 \odot e_2 e_2
\]

Using the compositionality of $[\cdot]$ (Thm. 15) and (22):

\[
(S, X) [e_1] (S_1, X) [a_1]) (S_1 \setminus a_1, X_1 : b) \quad \Leftrightarrow (S, X) [e_2] (S_2, X) [a_2]) (S_2 \setminus a_2, X_2 : b)
\]

which, because $X_1 : b : X_2 = X_2 : b : X_2 = X'$ and $b$ is fresh, and because $S_1$ and $S_2$ are in agreement except for $a_1$ and $a_2$, can be rewritten as

\[
(S, X) [e_1] (S_1', X_1) [a_1]) (S_1' \setminus a_1, X_1 : b) \quad \Leftrightarrow (S, X) [e_2] (S_2', X_2) [a_2]) (S_2' \setminus a_2, X_2 : b)
\]

if and only if

\[
(S, X) [e_1] (S_1', X_1) \quad \Leftrightarrow (S, X) [e_2] (S_2', X_2)
\]

if and only if

\[
(S, X) [e_1] (S_1', X_1) \quad \Leftrightarrow (S, X) [e_2] (S_2', X_2)
\]
We note immediately that
$$[e_1] = [e_2]$$ so by the IH $e_1 =_\alpha e_2$. \hspace{1cm} (26)

We also note that $[(a_1 e_1)_2], [(a_2 e_2)_2]$ differ only by the entries $(a_1 \rightarrow b), (a_2 \rightarrow b)$ in their
dictionaries. Following a line of reasoning analogue to that in Thm. 31 and using (23) it
implies that for a fresh $c \in \mathcal{A}$, $[(c \cdot a_1) \cdot e_1)_2] = [(c \cdot a_2) \cdot e_2]$ which, from the IH, implies
$(c \cdot a_1 \cdot e_1)_2 =_\alpha (c \cdot a_2 \cdot e_2)_2$. Together with (26) and the congruence property of $=_\alpha$ (Thm. 10),
it follows that $e_{11} \cdot (c \cdot a_1 \cdot e_1)_2 = e_{21} \cdot (c \cdot a_2 \cdot e_2)_2$, so that $e_1 =_\alpha e_2$ by $(\alpha \alpha)$. 

\textbf{Proof of Lem. 19.} Clearly $\text{RSeq}$ is a nominal monoid and since $=_{\alpha}$ is an equivalence
relation and a congruence with respect to concatenation, we have a nominal monoid
structure on $\text{DSeq} = \text{RSeq}/=_{\alpha}$.

The functions $\langle \cdot \rangle$ and $\cdot$ map $a$ to the equivalence classes of $\langle a \rangle$ and $a$ respectively. The function $\gamma$ on $\text{RSeq}/=_{\alpha}$ is obtained from the function $\mathcal{F} : \text{RSeq} \rightarrow \mathcal{B}^\mathcal{A}$ of Lem. 2, which by
Lem. 7 factors through $\text{RSeq}/=_{\alpha}$.

It remains to check the equations in Def. 18 are satisfied. The first three are clear from
the definition of $\mathcal{F}$.

To prove (16), it suffices to show that for any $e \in \text{RSeq}$ and $a, b \in \mathcal{A}$ such that $b \# e$ and
$a \odot e$ we have $\langle a e a \rangle = =_{\alpha} (b(a \cdot b) \cdot eb)$. We obtain this as an instance of $(\alpha \alpha)$:

1. $e : (c \cdot a) \cdot e = =_{\alpha} \langle e \cdot (b(a \cdot b) \cdot eb) \cdot e \rangle$

The premise holds since $(c \cdot a) \cdot e = (c(ba) \cdot eb) \cdot e$ for fresh $b$.

\textbf{Proof of Thm. 21.} Consider $(X, \gamma) \in \text{BNom}$, the set $X + \mathcal{A} = X \cup \{\langle a \rangle | a \in \mathcal{A}\}$, and the equivariant map $\gamma : X + \mathcal{A} \rightarrow \mathcal{B}^\mathcal{A}$ that acts as $\gamma$ on $X$ such that $\gamma_a(\langle a \rangle) = c$, $\gamma_a(b) = e$ and $\gamma_a(a) = e$. Then $\gamma$ can be extended uniquely to a monoid
morphism $\gamma^* : (X + \mathcal{A})^* \rightarrow \mathcal{B}^\mathcal{A}$. Define a relation $\equiv$ on $(X + \mathcal{A})^*$ as the congruence
generated by $\langle a \cdot b \rangle = \langle b(a \cdot b) \cdot wb \rangle$, where $a, b \in \mathcal{A}$, $w \in (X + \mathcal{A})^*$ and $b \# w$ and $\gamma^* (w) \in \{e, f\}$.

Conduct $F(X, \gamma)$ as a dynamic binding monoid on the carrier nominal set $(X + \mathcal{A})^*/\equiv$. Left and right binders are defined in the obvious way and the specification function is induced by $\gamma^*$. It is easy to check that whenever $w \equiv w'$ then $\gamma^* (w) = \gamma^* (w')$. We must exhibit an isomorphism $\text{DBMon}((X + \mathcal{A})^*/\equiv, \mathcal{M}) \simeq \text{BNom}((X, \gamma), (\mathcal{M}, R Seq \mathcal{M}))$. Starting with a
morphism $f : X \rightarrow M$ in $\text{BNom}$, we can uniquely extend $f + \cdot : X + \mathcal{A} \rightarrow M$ to an equivariant monoid
morphism $f_a : (X + \mathcal{A})^* \rightarrow M$. We have that $\gamma \circ f_a = \gamma^*$. It follows that for every $w, w' \in (X + \mathcal{A})^*$ such that $w \equiv w'$ then $f_a (w) = f_a (w')$. Hence, $f_a$
factors through a dynamic binding monoid morphism $\mathcal{T} : (X + \mathcal{A})^*/\equiv \rightarrow \mathcal{M}$. Conversely, given $g \in \text{DBMon}((X + \mathcal{A})^*/\equiv, \mathcal{M})$ we consider $g^* \in \text{BNom}((X, \gamma), (\mathcal{M}, R Seq \mathcal{M}))$ given by $g^* (x) = g([x])$ where $[x]$ is the $\equiv$-equivalence class of $x$.

\textbf{Proof of Thm. 22.} We have that $\text{RSeq} = (\mathcal{A} \cup \mathcal{B} + \mathcal{A})^*$, thus it suffices that $=_{\alpha}$ is equal to the relation $\equiv$ described in the proof of Thm. 21. That $\equiv =_{\alpha}$ follows from the proof of
Lem. 19. 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 $(\alpha e)$ or $(\alpha \alpha)$
then the proof is immediate by induction. Assume that $e =_{\alpha} e'$ was derived using $(\alpha \alpha)$. That is, $e \equiv e_1 (ae_2 a), e' = e'_1 (be'_2 b)$ such that $a \odot e_2, b \odot e'_2$ and for any fresh $c$ we have
e_1 (c(c \cdot a)) \cdot e_2 = e'_1 (c(c \cdot b) \cdot e'_2$. By inductive hypothesis $e_1 \cdot (c(c \cdot b)) \cdot e_2 \equiv e'_1 \cdot (c(c \cdot b) \cdot e'_2$ for fresh$c; we must prove $e_1 (ae_2 a) \equiv e'_1 (be'_2 b)$.
We consider the case when \(e_1(c \cdot c \cdot a) \cdot e_2 = h_1(\text{hdh})h_2\) and \(e'_1(c \cdot c \cdot b) \cdot e'_2 = h'_1(\text{d'(d' d) · hd'})h'_2\) such that \(h_1 \equiv h'_1, \ h_2 \equiv h'_2, \ d' \equiv h'\) and \(d \equiv h\). The most interesting case is when \(h = g_1(c \cdot g_2)\). We have that
\[
\begin{align*}
\langle ae2a \rangle &= h_1(dg_1(a \cdot c) \cdot g_2d(a \cdot c))h_2a) \\
&\equiv h_1(dg_1(cg_2d)h_2c) \\
&\equiv h'_1(d'(d' d) \cdot g_1(c(d' d) \cdot g_2d')h'_2c) \\
&\equiv h'_1(d'(d' d) \cdot g_1(b(b \cdot c)(d' d) \cdot g_2d')(b(c \cdot c)h'_2b) \\
&= e'_1(b \cdot c' \cdot b).
\end{align*}
\]

It might be the case that \(e_1(c \cdot c \cdot a) \cdot e_2 \equiv e'_1(c \cdot c \cdot b) \cdot e'_2\) was obtained using the transitivity of \(\equiv\), for example \(\langle d_1(\text{d_2(c d_2)}d_1) \equiv (d'_1(\text{d'_2(c d_2)}d'_1)\) can only be derived using the transitivity of \(\equiv\). For this case the proof that \(e_1(\text{ae2a}) \equiv e'_1(\text{be2b})\) requires several steps and transitivity, but it reduces to the basic case resolved above.

\section{Languages with explicit allocation and deallocation}

It seems natural to use dynamic sequences to model programming languages with explicit resource allocation and deallocation. We now give a toy example of such a language: consider an imperative language with resource management primitives whose terms are defined by:
\[
C ::= \text{alloc}(x) \mid \text{use}(x) \mid \text{free}(x) \mid \text{new} \ x.C \mid C; C \mid C parallel C' \mid \text{skip}.
\]

Resources are bound to (local) variables chosen from a set \(X\), and can be allocated, used, or released. Commands can be sequenced or interleaved. We also include the trivial command \(\text{skip}\). The nominal treatment of local variables, chosen from set of atoms \(X\), is conventional and the definitions of free or bound variable in a term are the standard ones, as is alpha equivalence on terms. The set of terms (up to \(\alpha\)-equivalence) is denoted by \(\text{Prog}\). We give a denotational trace semantics of \(\text{Prog}\). Let \(X\) be the set of constants used in defining raw sequences (Def. 1). A program trace will consist of a sequence of symbols indicating at what variable an action occurred, and what that action was (allocation, use, deallocation). Define the denotational semantics \([\cdot]: \text{Prog} \rightarrow P(\text{DSeq})\) by:
\[
\begin{array}{l}
\langle \text{skip} \rangle = \{\varepsilon\} \\
\langle \text{alloc}(x) \rangle = \{x\langle a \mid a \in A\} \\
\langle \text{use}(x) \rangle = \{xa \mid a \in A\} \\
\langle \text{free}(x) \rangle = \{xa \mid a \in A\}
\end{array}
\]

Command composition, sequential or parallel, is defined as:
\[
\begin{align*}
\langle C; C' \rangle &= \{ee' \mid e \in [C], e' \in [C'], \#e'\} \\
\langle C parallel C' \rangle &= \{e || e' \mid e \in [C], e' \in [C'], \#e'\}.
\end{align*}
\]

This is perhaps surprising because it forces mismatches between actions associated with the same variable, e.g.
\[
\langle \text{alloc}(x); \text{free}(x) \rangle = \{x\langle a :: xb \mid a, b \in A, a \neq b\}
\]

This ‘bad variable’ behaviour is inspired by game-semantic models of state. As in game semantics, order is restored by the local variable binder \([1]\), replacing all names of actions associated with \(x\) with a fresh name \(c\), then removing all occurrences of \(x\) from the trace:
\[
\langle \text{new} \ x.C \rangle = \{e' \setminus x \mid e \in [C], \#e' = e[\text{supp}(e|x) \setminus c]\}
By $e[x$ we mean sequence $e$ restricted to a concatenation of sub-sequences of form $xm$. The substitution notation is extended pointwise to a set of atoms. For instance,

$$[\text{new } x.\text{alloc}(x); \text{free}(x)] = \{\langle aa \rangle\},$$

a singleton set. We can also prove equations such as

$$(\text{new } x.\text{alloc}(x); \text{free}(x)) \parallel (\text{new } x.\text{alloc}(x); \text{free}(x))$$

$$\equiv \text{new } x.\text{new } y.\text{alloc}(x); \text{free}(y)) \parallel \text{alloc}(y); \text{free}(y))$$

which both are interpreted as $\langle aa \rangle \parallel \langle aa \rangle$. Straightforward proofs are possible that sequential and parallel composition form monoids on $\text{Prog}$ with unit $\text{skip}$.

Due to design decisions discussed in Sec. 2.3 this language satisfies equivalences that C-like languages do not, such as:

$$\text{new } x.\text{alloc}(x); \text{alloc}(x); \text{free}(x) \equiv \text{new } x.\text{new } y.\text{alloc}(x); \text{alloc}(y); \text{free}(y); \text{free}(x)$$

which are both $\{\langle a(aa)\rangle\}$. 