Bracket Algebra
A nominal theory of interleaved scopes*

Paul Brunet[1], Daniela Petrişan[2], and Alexandra Silva[1]

[1] University College London, United Kingdom
[2] IRIF, France

Abstract. In this paper we present *bracket algebra*, a nominal framework that can be used in reasoning about programs with interleaved scopes. We construct a hierarchy of languages based on their memory and binding power. We present a decision procedure for program equivalence and containment for a large class of programs.

1 Introduction

Program equivalence is a core subject in programming language semantics and formal verification. Despite being an undecidable problem in general, it is easier to solve in many cases of practical interest and therefore has been studied by many authors from different communities (see e.g. recent overview by Strichman [7]).

In this paper we will focus our attention in programs where resources, stored in variables being manipulated by the program, can be explicitly allocated and freed generating so-called *dynamically-scoped binding* of variables. The choice of name for the variables, between the allocation and the release of the resource, should not affect program equivalence. In addition, and more challengingly, possible scope interleaving has to be accounted for in any techniques developed for such programs.

Nominal techniques have been developed for reasoning about abstract syntax with *statically-scoped binding* and very little is known about how these can be extended to dynamically-scoped binding. In this paper we take inspiration from previous work by Gabbay, Ghica, and Petrişan [2] who introduced a syntactic notion of dynamic sequences and develop an algebraic framework to reason about program equivalence in the presence of interleaved scopes.

To motivate the work in the paper consider the following three programs, which manipulate the contents of variables `x` and `y` and which we would like to prove equivalent.

```
int i;
int j;
j:=x;
x:=y;
for(i=1,i<=10,i++)
   x++;
x++;
free(i);
y:=j;
free(j);
```

P1

```
int i;
int j;
j:=x;
x:=y;
x++;
for(i=1,i<=10,i++)
   x++;
free(i);
y:=j;
free(j);
```

P2

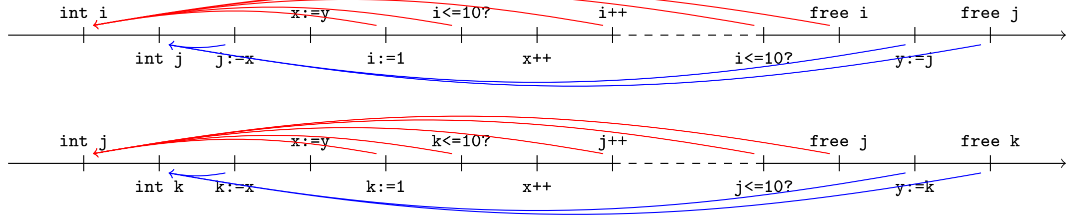
```
int j;
int k;
k:=x;
x:=y;
for(j=1,j<=10,j++)
   x++;
x++;
free(j);
y:=k;
free(k);
```

P3

* This work was partially supported by ERC starting grant ProFoundNet (679127) and a Leverhulme Prize (PLP-2016-129).

All three programs start with variables `x` and `y` holding values x_0 and y_0, respectively, and finish with variables `x` and `y` holding value $y_0 + 11$ and x_0, respectively.

The first two programs – `P1` and `P2` – can easily be proven equivalent without any nominal techniques using, for instance, (plain) Kleene algebra [4]. `P2` and `P3` are however out of the scope of the current algebraic techniques, both because of the renaming in local variables and the fact that those names are used in between (de-)allocation of different resources. The challenge can neatly be summarised if we depict the the variable references of the program along a timeline.



As the figures above demonstrate the reference structure of the variables in both programs is exactly the same. However, the fact that the scopes are interleaved make reasoning about equivalence of these programs out of reach of existing methods.

In this paper we will develop a framework – *bracket algebra* – where we can reason algebraically about the programs above in the same way as we would using Kleene algebra to reason about equivalence of simple imperative programs. At the core of our development will be considering allocation and deallocation of resources as integral part of the program traces. Our traces will be built from two (nominal) alphabets, one containing program instructions $x, y, z, \ldots \in \mathbb{X}$ and the other one containing allocation and deallocation instructions – $\langle_a$ and $_a\rangle$ – for every resource name $a \in \mathbb{A}$. For instance the first program above would have traces of the shape $\langle_i \langle_j a^j b\ c^i d\ _i\rangle e^j\ _j\rangle$. Here we use the superscript to denote the fact that those letters stand for parts of the program that use a certain local variable. Proving equivalence of the above three programs would amount to checking equivalence of the expressions:

$$\langle_i \langle_j a^j b\ c^i d\ _i\rangle e^j\ _j\rangle \quad (\texttt{P1}) \qquad \langle_i \langle_j a^j b\ dc^i\ _i\rangle e^j\ _j\rangle \quad (\texttt{P2}) \qquad \langle_j \langle_k a^k b\ c^j d\ _j\rangle e^k\ _k\rangle \quad (\texttt{P3})$$

It is clear that for (`P1`) and (`P2`) we just need to show $c^i d \equiv dc^i$, which follows easily when expanding c^i to the expression increasing the value of x 10 times (that is, executing instruction d 10 times). For (`P2`) and (`P3`), in addition to the above swap, we need to rename in (`P3`) j to i and k to j. To have algebraic laws doing this renaming in a sound (and complete) way we need to be careful with the scope of the variables and the fact that their allocation and de-allocation is not done in the same order (hence the mismatched brackets in the above expressions).

This paper develops a general framework to reason about program traces like the above. We develop the framework also for sets of traces – languages – and this lead us to naturally look at properties of languages and their expressive power when taking into account several properties stemming from their nominal nature – e.g. finite support vs bounded support – and from their binding power. We will show that variations on these generate very different classes of languages and organise those in a hierarchy. This is a first detailed account of a nominal language hierarchy, which has been an open problem for some years now in the field of nominal language theory. The reason why this is more challenging that in the classical case is that in the presence of nominal alphabets the hierarchy turns out to be much finer (e.g. non-deterministic and deterministic nominal automata are not equi-expressive). In a companion paper [1], we provide a Kleene theorem for (non-deterministic) nominal-automata, bridging the syntactic and semantic description of nominal regular languages.

We structure the paper as follows. We first set up the basic definitions to reason about traces/words that contain allocation and deallocation binders, possibly interleaved (Section 2). This leads to the development of a new *nominal transducer model* that is used to devise an algorithm to decide α-equivalence of words (Section 3). The framework is then extended to be able to reason about languages (Section 4), that is sets of traces, and this leads us in an exploration of the different classes of languages that can be defined in this setting. We present a hierarchy based on on the languages memory and binding power (Section 5). Finally, we present a decision procedure for language equivalence and containment for a subclass of languages (Section 6).

2 Preliminaries

Words, languages, and sets The set of words over an alphabet Σ is written $\Sigma^\star$. The empty word is denoted by ε, the concatenation of words u and v is written uv, and $|u|$ is the length of u. If w is a word over the alphabet Σ and $x \in \Sigma$, then $|w|_x$ is the number of occurrences of x in w. We sometimes identify words with the set of letters occurring in them, writing $x \in w$ to denote that x appears in w, i.e. $|w|_x \neq 0$. We denote the ith letter of a word u by u_i, for $0 < i \leqslant |u|$.

We denote the set of finite subsets of A by $\mathcal{P}_f(A)$ and, for A finite, its cardinal by $\#A \in \mathbb{N}$.

Nominal sets We fix an infinite set $\mathbb{A}$ of names, and write $\mathfrak{S}_\mathbb{A}$ the set of finitely supported permutations over $\mathbb{A}$. These are bijections π such that there is a finite set $A \subseteq \mathbb{A}$ such that $a \notin A \Rightarrow \pi(a) = a$. The inverse of a permutation π is written π^{-1} . The permutation exchanging a and b, and leaving every other name unchanged, is written $(a\ b)$. A set X is called *nominal* if there are functions $- \cdot - : \mathfrak{S}_\mathbb{A} \times X \to X$ and $\mathbf{supp}(-) : X \to \mathcal{P}_f(\mathbb{A})$, respectively called the *action* and the *support*, such that $\forall x \in X,\ \forall \pi, \pi' \in \mathfrak{S}_\mathbb{A}$, we have:

$$(\forall a \in \mathbf{supp}(a),\ \pi(a) = a) \Rightarrow \pi \cdot x = x. \tag{1}$$

$$\mathbf{supp}(\pi \cdot x) = \left\{ a \in \mathbb{A} \;\middle|\; \pi^{-1}(a) \in \mathbf{supp}(x) \right\}. \tag{2}$$

$$\pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x. \tag{3}$$

Intuitively, this means that we may replace a name by another in any element of X, and that each element of X only depends on a finite number of names. We say that the name a is *fresh* for the variable x, and write $a \# x$, whenever $a \notin \mathbf{supp}(x)$.

Remark 2.1. In [6] a nominal set is defined as a $\mathfrak{S}_\mathbb{A}$-action such that every element has some finite support. From conditions (1) and (3) we infer that X is a nominal set as in [6]. Furthermore, condition (2) enforces that $\mathbf{supp}(x)$ is the least finite set that supports x, so our notion of support coincides with the the one introduced in [6]. For `Coq` implementation considerations, we opted for explicitly including the support function in the definition.

We fix for the remainder of the paper a nominal set $\mathbb{X}$ of variables, representing atomic instructions or events.

3 Words over an alphabet with binders

In this section, we will set up the scene for our algebraic framework. We will introduce the basic definitions on words over an alphabet with binders and alpha-equivalence (Section 3.1) which are straightforward adaptations from [2]. We will then present a novel transducer model (Section 3.2) that is powerful enough to be a sound and complete mechanism (Section 3.3) to capture alpha-equivalence of words. Section 3.4 contains the details of the decision procedure

for deciding alpha equivalence of words – in essence we present a compilation procedure from a pair of words to a transition relation of the respective transducer. We study the properties of the labelled transition system induced by the transducer and show that the several ways to compare the states of this LTS (simulation, bisimulation, etc) are equally expressive.

3.1 α-equivalence of words with binders

The traces we will consider in this paper will consist of words built out of an alphabet consisting of variables $x \in \mathbb{X}$ and left and right binders, respectively written $\langle_a$ and ${}_a\rangle$. These binders represent the creation and destruction of names. For instance, one might think of $\langle_a$ as an instruction allocating a new memory location, and linking it to a variable named a. Conversely the instruction ${}_a\rangle$ would free any space allocated to a.

In this what follows we introduce a notion of α-equivalence for these sequences. Because we are committed to a compositional semantics, we want this relation to be a congruence: if two words are equivalent, appending to both the same prefix and the same suffix should yield a pair of equivalent words. We also want to be able to substitute "local" names: for instance the sequence $\langle_a\ {}_a\rangle$ should be equivalent to the sequence $\langle_b\ {}_b\rangle$.

Formally, we define our alphabet by $\Sigma^{\mathbb{A}}_{\mathbb{X}} := \mathbb{X} \cup \{\langle_a \mid a \in \mathbb{A}\} \cup \{{}_a\rangle \mid a \in \mathbb{A}\}$. If the sets of atoms and variables are clear from the context, we simply write Σ. This alphabet can be endowed with a nominal structure, by setting $\pi \cdot \langle_a = \langle_{\pi(a)}$, $\pi \cdot {}_a\rangle = {}_{\pi(a)}\rangle$, and $\mathbf{supp}(\langle_a) = \mathbf{supp}({}_a\rangle) = \{a\}$. In the following, a word will be an element of $\Sigma^\star$, that is a finite sequence of letters from the alphabet Σ. Words naturally support a nominal structure: the action is defined by applying the alphabet action letter by letter, and the support is the union of the supports of its letters.

Before we define α-equivalence, we need to introduce the notion of *binding power* of a word. The purpose of this notion is to keep track of the occurrences of each name along a word, and enable us to decide whether a particular name is local to the word, and more generally to get a precise account of the way the name is used in the word, from the point of view of the context. The *binding monoid* $\mathcal{B}$ is defined as the free monoid over the three element set $\{\mathbf{c}, \mathbf{f}, \mathbf{d}\}$, quotiented by the following identities: $\mathbf{f} \cdot \mathbf{f} = \mathbf{f}$, $\mathbf{c} \cdot \mathbf{f} = \mathbf{c}$, $\mathbf{f} \cdot \mathbf{d} = \mathbf{d}$, and $\mathbf{c} \cdot \mathbf{d} = \varepsilon$. The letters $\mathbf{c}$, $\mathbf{f}$, $\mathbf{d}$ represent that a name might be *created*, *free* or *destroyed*. The first identity reflects the fact that seeing several free occurrences of the same name in a row is equivalent to seeing a single free occurrence. For the second and third identities may be interpreted as saying that the information that a name is created (respectively destroyed) contains the fact that the name is free. The last law states that creating and then deleting a name hides it from the context: it is as if the name had not been used at all. An important property of this monoid is the following, as noticed in [2]: every element of $\mathcal{B}$ can be uniquely represented in the form $\mathbf{d}^m\mathbf{f}^n\mathbf{c}^p$, with $\langle m, n, p\rangle \in \mathbb{N} \times \{0, 1\} \times \mathbb{N}$.

The *binding power* of a letter $l \in \Sigma$ with respect to a name $a \in \mathbb{A}$, written $\mathcal{F}_a(l)$, as follows:

$$\mathcal{F}_a(\langle_b) := \begin{cases} \mathbf{c}\ (a = b) \\ \varepsilon\ (a \neq b) \end{cases} \qquad \mathcal{F}_a({}_b\rangle) := \begin{cases} \mathbf{d}\ (a = b) \\ \varepsilon\ (a \neq b) \end{cases} \qquad \mathcal{F}_a(x) := \begin{cases} \mathbf{f}\ (a \in \mathbf{supp}(x)) \\ \varepsilon\ (a \mathrel{\#} x) \end{cases}$$

The function $\mathcal{F}$ extends to words naturally as a monoid homomorphism, by setting $\mathcal{F}_a(\varepsilon) = \varepsilon$ and $\mathcal{F}_a(lw) = \mathcal{F}_a(l) \cdot \mathcal{F}_a(w)$. If $\mathcal{F}_a(u) = \mathbf{d}^m\mathbf{f}^n\mathbf{c}^p$ with $n \in \{0, 1\}$, we define $d_a(u) := m$, $f_a(u) := n$, and $c_a(u) := p$. Notice that this is well defined thanks to the uniqueness of such representations. This function commutes with the action of permutations, in the sense that $\mathcal{F}_{\pi(a)}(\pi \cdot u) = \mathcal{F}_a(u)$.

We use the binding power to define the following: a is *balanced* in the word w, written $a \diamond w$, if $\mathcal{F}_a(w) \in \{\mathbf{f}, \varepsilon\}$; a is *α-fresh* in w, written $a \mathrel{\#_\alpha} w$, is $\mathcal{F}_a(w) = \varepsilon$; the *$\alpha$-support* of w, written $\mathbf{supp}_\alpha(w)$, is the set of names a such that $\mathcal{F}_a(w) \neq \varepsilon$. Notice that $\mathbf{supp}_\alpha(w) \subseteq \mathbf{supp}(w)$. Using the previous remark, we get that $\pi(a) \mathrel{\#_\alpha} \pi \cdot u$ if and only if $a \mathrel{\#_\alpha} u$, and similarly for $\pi(a) \in \mathbf{supp}_\alpha(\pi \cdot u)$ and $\pi(a) \diamond \pi \cdot u$.

We may now define the *α-equivalence* relation over words. It is the smallest congruence such that applying the transposition $(a\ b)$ to a word where a and b are α-fresh yields an equivalent word. Formally, the relation $=_\alpha$ is defined inductively by:

$$\frac{}{\varepsilon =_\alpha \varepsilon}\ (\alpha\varepsilon) \qquad \frac{u =_\alpha v \quad v =_\alpha w}{u =_\alpha w}\ (\alpha\mathbf{t})$$

$$\frac{w_1 =_\alpha w_2 \quad l \in \Sigma}{w_1 l =_\alpha w_2 l}\ (\alpha\mathbf{r}) \qquad \frac{w_1 =_\alpha w_2 \quad l \in \Sigma}{l w_1 =_\alpha l w_2}\ (\alpha\mathbf{l}) \qquad \frac{a \diamond u \quad b \#_\alpha u}{\langle_a u\, _a\rangle =_\alpha \langle_b (a\ b) \cdot u\, _b\rangle}\ (\alpha\alpha)$$

The following statements hold:

$$u =_\alpha v \Rightarrow v =_\alpha u \quad (4) \qquad u =_\alpha v \Rightarrow \forall \pi \in \mathfrak{S}_{\mathbb{A}},\ \pi \cdot u =_\alpha \pi \cdot v \quad (7)$$

$$u =_\alpha v \wedge u' =_\alpha v' \Rightarrow uu' =_\alpha vv' \quad (5) \qquad u =_\alpha v \Rightarrow |u| = |v| \quad (8)$$

$$u =_\alpha v \Rightarrow \forall a \in \mathbb{A},\ \mathcal{F}_a(u) = \mathcal{F}_a(v) \quad (6)$$

The propositions (4) and (5) state that $=_\alpha$ is symmetric and that concatenation is compatible with $=_\alpha$, which together with $(\alpha\varepsilon)$ and $(\alpha\mathbf{t})$ establishes $=_\alpha$ as a congruence, while (6), (7), and (8) are necessary preservation properties of $=_\alpha$. The proofs of these results follow a simple induction of proof trees.

Note that the deduction system we provided for $=_\alpha$ is not a priori equivalent to the informal description we gave before. However, the correspondence can be proved in the sense that the same relation is obtained if we replace rule $(\alpha\alpha)$ with the following rule:

$$\frac{a \#_\alpha u \quad b \#_\alpha u}{u =_\alpha (a\ b) \cdot u}\ (\alpha\alpha')$$

However, this proof is not straightforward: $(\alpha\alpha')$ obviously implies $(\alpha\alpha)$ (as the latter may be seen as an instance of the former), but the converse direction is more subtle. Unfortunately, this is the most important direction, as it is necessary to show that words quotiented by $=_\alpha$ form a nominal set, with the support function $\mathbf{supp}_\alpha()$. To obtain this property we will rely on the transducer presented in the next section.

3.2 A transducer for α-equivalence-checking

The problem that arises when trying to prove statements like $(\alpha\alpha)$ is that α-equivalence is not preserved in the inductive calls: the property $ux =_\alpha vy$ does not entail $u =_\alpha v$. In this section we introduce a nominal transducer recognising the relation $=_\alpha$. The reachability relation in this transducer will give us more powerful proof techniques, allowing us to perform proofs by induction. We will use this transducer for several purposes: it will provide us with a decision procedure for $=_\alpha$ and it will enable us to show that $(\alpha\alpha')$ is admissible.

Stacks The states of this transducer will consist of lists of pairs of atoms, called stacks in the following. Before we define the transducer, we introduce some useful notations. *Stacks* are generated by the following grammar: $s \in \mathbb{S} ::= [] \mid s :: \langle a, b\rangle$, where a, b range over names. Hence $\mathbb{S}$ is isomorphic to $(\mathbb{A} \times \mathbb{A})^\star$. We will also use the notation $s :: t$ for the concatenation of the two stacks $s, t \in \mathbb{S}$. We write $\mathfrak{p}_1(s)$ for the word over $\mathbb{A}$ obtained by erasing the second components of every pair in s, and symmetrically $\mathfrak{p}_2(s)$ when we erase the first components. For instance $\mathfrak{p}_1([] :: \langle a, b\rangle :: \langle c, d\rangle) = ac$, and $\mathfrak{p}_2([] :: \langle a, b\rangle :: \langle c, d\rangle) = bd$.

Stacks can be endowed with a canonical nominal structure defined by:

$$\pi \cdot [] := [] \qquad \pi \cdot (s :: \langle a, b\rangle) := \pi \cdot s :: \langle \pi(a), \pi(b)\rangle$$

$$\mathbf{supp}([]) := \emptyset \qquad \mathbf{supp}(s :: \langle a, b\rangle) := \mathbf{supp}(s) \cup \{a, b\}\,.$$

Note that $\mathbf{supp}(s) = \mathbf{supp}(\mathfrak{p}_1(s)) \cup \mathbf{supp}(\mathfrak{p}_2(s)) = \mathfrak{p}_1(s) \cup \mathfrak{p}_2(s)$, the last identity using our shorthand identifying words with the set of letters occurring in them.

The pivotal notions for stacks are the validates predicate and the pop function. We say that a stack s *validates* the pair $\langle a, b\rangle$, written $s \models \langle a, b\rangle$, when the following holds:

$$(a = b \wedge a \notin \mathbf{supp}(s)) \vee (\exists s', s'' \in \mathbb{S} : s = s' :: \langle a, b\rangle :: s'' \wedge a \notin \mathfrak{p}_1(s'') \wedge b \notin \mathfrak{p}_2(s''))\,.$$

When s validates $\langle a, b\rangle$, we may *pop* the pair from s, yielding the stack $s \ominus \langle a, b\rangle$ defined by:

$$\frac{a \notin \mathbf{supp}(s)}{s \ominus \langle a, a\rangle := s} \qquad \frac{a \notin \mathfrak{p}_1(s') \wedge b \notin \mathfrak{p}_2(s')}{(s :: \langle a, b\rangle :: s') \ominus \langle a, b\rangle := s :: s'.}$$

We now define the *equivalence transducer* $\mathcal{T}$. Its state space is $\mathbb{S}$, with initial state $[]$, and the set of *accepting* states $\mathbb{S}^{acc}$ consists of all stacks s containing only reflexive pairs, i.e. such that $\mathfrak{p}_1(s) = \mathfrak{p}_2(s)$. The transition relation $\rightarrow_{\mathcal{T}}$ is defined by:

$$\frac{}{s \,-[\langle_a / \langle_b]\!\rightarrow_{\mathcal{T}}\, s :: \langle a, b\rangle} \qquad \frac{s \models \langle a, b\rangle}{s \,-[\,_a\rangle / \,_b\rangle]\!\rightarrow_{\mathcal{T}}\, s \ominus \langle a, b\rangle} \qquad \frac{\forall a \in \mathbf{supp}(x),\ s \models \langle a, \pi(a)\rangle}{s \,-[x / \pi \cdot x]\!\rightarrow_{\mathcal{T}}\, s}$$

Note that this relation is functional, in the sense that for every triple $\langle s, l, l'\rangle \in \mathbb{S} \times \Sigma \times \Sigma$ there exists at most one stack s' such that $s \,-[l/l']\!\rightarrow_{\mathcal{T}}\, s'$. We extend the transition relation to words, defining as usual $s \,-[u/v]\!\rightarrow_{\mathcal{T}}\, s'$ by saying that $s \,-[\varepsilon/\varepsilon]\!\rightarrow_{\mathcal{T}}\, s$ and if $s \,-[u/v]\!\rightarrow_{\mathcal{T}}\, s'$ and $s' \,-[l/l']\!\rightarrow_{\mathcal{T}}\, s''$, then $s \,-[ul/vl']\!\rightarrow_{\mathcal{T}}\, s''$. This transducer over an infinite state space is nominal, as one can easily check that $s \,-[u/v]\!\rightarrow_{\mathcal{T}}\, s'$ entails $\pi \cdot s \,-[\pi \cdot u / \pi \cdot v]\!\rightarrow_{\mathcal{T}}\, \pi \cdot s'$. However, it is not orbit finite. This seems to be unavoidable since there are infinitely many α-equivalence classes.

3.3 Soundness and completeness

In this section, we show that two words u, v are α-equivalent if and only if there is a path in the transducer labelled with u/v from $[]$ to some accepting stack.

Theorem 3.1. *The relation accepted by the equivalence transducer is exactly* $=_\alpha$.

The full proof has been done in `Coq`, so we will only give a sketch of it here.

Completeness We start by the left-to-right implication: $u =_\alpha v \Rightarrow \exists s \in \mathbb{S}^{acc} : [] \,-[u/v]\!\rightarrow_{\mathcal{T}}\, s$. This proof is done by induction on the derivation $u =_\alpha v$.

Empty word. If the last rule applied was $(\alpha\varepsilon)$, we have $u = v = \varepsilon$ so we may just pick s to be $[]$.

Transitivity. For transitivity (rule $(\alpha\mathbf{t})$), we prove a slightly stronger lemma. Given two lists of atoms of the same length, say $A = a_1 \dots a_n$ and $B = b_1 \dots b_n$, we write $A \otimes B$ for the stack $[] :: \langle a_1, b_1\rangle \cdots :: \langle a_n, b_n\rangle$. Given two stacks s, s' of the same length, the stack $s \bowtie s'$ is defined as $\mathfrak{p}_1(s) \otimes \mathfrak{p}_2(s')$.

Lemma 3.2. *Let $s_1, \ldots, s_4$ be stacks and $l_1, l_2, l_3 \in \Sigma$ be letters such that $s_1 \ -[l_1/l_2]\!\!\to_{\mathcal{T}}\ s_2$, $s_3 \ -[l_2/l_3]\!\!\to_{\mathcal{T}}\ s_4$ and $\mathfrak{p}_2(s_1) = \mathfrak{p}_1(s_3)$, then the following hold:*

$$\mathfrak{p}_2(s_2) = \mathfrak{p}_1(s_4) \qquad \text{and} \qquad s_1 \bowtie s_3 \ -[l_1/l_3]\!\!\to_{\mathcal{T}}\ s_2 \bowtie s_4.$$

Proof. By case analysis on the letters l_1, l_2, l_3, using the following auxiliary result: if $s \models \langle a, b\rangle$, $s' \models \langle b, c\rangle$, and $\mathfrak{p}_2(s) = \mathfrak{p}_1(s')$, then $s \bowtie s' \models \langle a, c\rangle$ and:

$$\mathfrak{p}_2(s \ominus \langle a, b\rangle) = \mathfrak{p}_1(s' \ominus \langle b, c\rangle), \qquad (s \bowtie s' \ominus \langle a, c\rangle) = (s \ominus \langle a, b\rangle) \bowtie (s' \ominus \langle b, c\rangle).$$

This is done by a simple case analysis on $s \models \langle a, b\rangle$ and $s' \models \langle b, c\rangle$ (remember that the predicate $\models$ is defined as a disjunction), unfolding the definitions in each case. □

We may now prove transitivity as a matter of routine, first extending the above Lemma from letters to words, and then instantiating s_1 and s_3 with $[]$. The last ingredient consists in noticing that if $s_2, s_4 \in \mathbb{S}^{acc}$ and $\mathfrak{p}_2(s_2) = \mathfrak{p}_1(s_4)$, then $s_2 \bowtie s_4 \in \mathbb{S}^{acc}$, since:

$$\mathfrak{p}_1(s_2 \bowtie s_4) = \mathfrak{p}_1(s_2) = \mathfrak{p}_2(s_2) = \mathfrak{p}_1(s_4) = \mathfrak{p}_2(s_4) = \mathfrak{p}_2(s_2 \bowtie s_4).$$

Right congruence. If the last rule applied was (α**r**): $w_1 =_\alpha w_2 \Rightarrow w_1 l =_\alpha w_2 l$, then by induction hypothesis we simply need to check that if s is accepting, then there is another accepting stack s' such that $s \ -[l/l]\!\!\to_{\mathcal{T}}\ s'$. This is shown by case analysis on l without effort.

Left congruence. The case of rule (α**l**) starts similarly, by a case analysis on l. If l is either ${}_a\rangle$ or x, the result is straightforward. The remaining case of $\langle_a$ is slightly more involved: we know that $[] \ -[\langle_a / \langle_a]\!\!\to_{\mathcal{T}}\ [] :: \langle a, a\rangle$ and that $[] \ -[w_1/w_2]\!\!\to_{\mathcal{T}}\ s \in \mathbb{S}^{acc}$, and we need to find $s' \in \mathbb{S}^{acc}$ such that $[] :: \langle a, a\rangle \ -[w_1/w_2]\!\!\to_{\mathcal{T}}\ s'$. We prove by induction on paths that such an s' indeed exists, and is equal to either s or $\langle a, a\rangle :: s$.

α-Freshness. This is the most subtle case. We start with the following lemma, relating the binding powers and stacks.

Lemma 3.3. *Whenever $s \ -[u/v]\!\!\to_{\mathcal{T}}\ s'$ the following identities hold:*

$$|\mathfrak{p}_1(s')|_a = (|\mathfrak{p}_1(s)|_a \dot{-} \mathscr{d}_a(u)) + \mathscr{c}_a(u) \qquad |\mathfrak{p}_2(s')|_a = (|\mathfrak{p}_2(s)|_a \dot{-} \mathscr{d}_a(v)) + \mathscr{c}_a(v).$$

(Where $\dot{-}$ is the truncated subtraction.)

Proof. By induction on paths. □

For a permutation π and a stack s, we write $\pi \bullet_2 s$ for the stack $\mathfrak{p}_1(s) \otimes \pi \cdot \mathfrak{p}_2(s)$, obtained by applying π on the second component of s.

Lemma 3.4. *If $s \ -[u/v]\!\!\to_{\mathcal{T}}\ s'$, and if for every name a such that $\pi(a) \neq a$ we have $\mathscr{d}_a(v) + \mathscr{f}_a(v) \leqslant |\mathfrak{p}_2(s)|_a$, then there is a path $\pi \bullet_2 s \ -[u/\pi \cdot v]\!\!\to_{\mathcal{T}}\ \pi \bullet_2 s'$.*

Proof. By induction on u, using Lemma 3.3. □

This allows us to prove the following corollary, which can then be specialised into the case ($\alpha\alpha$):

Corollary 3.5. *If π is a permutation such that every name $a \in \mathbb{A}$ modified by π (i.e. $\pi(a) \neq a$) is α-fresh for u, then there is an accepting stack s such that $[] \ -[u/\pi \cdot u]\!\!\to_{\mathcal{T}}\ s$.*

Proof. From the cases of $(\alpha\varepsilon)$ and $(\alpha\mathbf{r})$, we can build inductively an stack s such that $[] -[u/u]\!\to_{\mathcal{T}} s$. The premise of Lemma 3.4 is met: if $\pi(a) \neq a$, then by assumption $a \#_\alpha u$, meaning that $d_a(u) + f_a(u) = 0 = |\mathfrak{p}_2([])|_a$. We apply the lemma and get $\pi \bullet_2 [] -[u/\pi\cdot u]\!\to_{\mathcal{T}} \pi \bullet_2 s$. Since $\pi \bullet_2 [] = []$, we only need to check that $\pi \bullet_2 s$ is accepting, i.e. that $\mathfrak{p}_1(s) = \pi \cdot \mathfrak{p}_2(s)$. This holds because for every $a \in \mathfrak{p}_2(s)$, we have $\pi(a) = a$. Indeed, if this was not the case, by hypothesis we would have $\mathcal{F}_a(u) = \mathbf{d}^0\mathbf{f}^0\mathbf{c}^0$, and by Lemma 3.3 this means that $|\mathfrak{p}_2(s)|_a = (|\mathfrak{p}_2([])|_a \dot{-} 0)+0 = 0$, meaning $a \notin \mathfrak{p}_2(s)$. □

Soundness We now endeavour to extract from a path $[] -[u/v]\!\to_{\mathcal{T}} s \in \mathbb{S}^{acc}$ a derivation of $u =_\alpha v$. We start by an induction on the length of u.

If $u = \varepsilon$, then v must also be the empty word, and $s = []$. We may thus conclude by applying the rule $(\alpha\varepsilon)$:

$$\frac{}{\varepsilon =_\alpha \varepsilon}\ (\alpha\varepsilon)$$

For the inductive case, we are in a situation where $[] -[u/v]\!\to_{\mathcal{T}} s -[l/l']\!\to_{\mathcal{T}} s' \in \mathbb{S}^{acc}$, and the induction hypothesis (I.H.) is:

$$\forall w_1, w_2 \in \Sigma^\star, \forall s \in \mathbb{S}^{acc},\ \text{if } |w_1| \leqslant |u| \text{ and } [] -[w_1/w_2]\!\to_{\mathcal{T}} s,\ \text{then } w_1 =_\alpha w_2.$$

We make a case distinction on the pair (l, l').

If $(l, l') = (\langle_a, \langle_b)$, then $s' = s :: \langle a, b\rangle$. Since $s' \in \mathbb{S}^{acc}$, we know that $s \in \mathbb{S}^{acc}$ and $a = b$. We build a derivation as follows:

$$\dfrac{\dfrac{[] -[u/v]\!\to_{\mathcal{T}} s \in \mathbb{S}^{acc}}{u =_\alpha v}\ (IH) \qquad \langle_a \in \Sigma}{u\langle_a =_\alpha v\langle_a}\ (\alpha\mathbf{r})$$

If $(l, l') = (x, y)$, then $s' = s$, and there is a permutation π such that $y = \pi \cdot x$ and $\forall a \in \mathbf{supp}(x), s \models \langle a, \pi(a)\rangle$. Since s is accepting we know that whenever $s \models \langle a, b\rangle$, we have $a = b$, which means that $\forall a \in \mathbf{supp}(x), \pi(a) = a$. Therefore, we have $x = \pi \cdot x = y$, so we are in the same situation as in the previous case:

$$\dfrac{\dfrac{[] -[u/v]\!\to_{\mathcal{T}} s \in \mathbb{S}^{acc}}{u =_\alpha v}\ (IH) \qquad x \in \Sigma}{ux =_\alpha vx}\ (\alpha\mathbf{r})$$

The remaining case is much more involved. Assume $(l, l') = ({}_a\rangle, {}_b\rangle)$, then $s \models \langle a, b\rangle$ and $s' = s \ominus \langle a, b\rangle$. Remember that there are two cases for $s \models \langle a, b\rangle$: either $a = b$ and $a \notin \mathbf{supp}(s)$, in which case $s' = s$ and we may conclude like in the two other cases; or s may be written as $s_1 :: \langle a, b\rangle :: s_2$ such that $a \notin \mathfrak{p}_1(s_2)$ and $b \notin \mathfrak{p}_2(s_2)$. In this case, $s' = s_1 :: s_2$. Note however that a may be different from b, in which case although s' is accepting, s is not. We rely here on the following result:

$$\exists u_1, u_2, v_1, v_2 \in \Sigma^\star : u = u_1\langle_a u_2 \ \wedge\ v = v_1\langle_b v_2 \ \wedge\ |u_1| = |v_1| \ \wedge\ a \diamond u_2 \ \wedge\ b \diamond v_2. \qquad (9)$$

This is a simple consequence of a stronger auxiliary lemma (together with Lemma 3.3):

Lemma 3.6. *If $[] -[u/v]\!\to_{\mathcal{T}} s_1 :: \langle a, b\rangle :: s_2$, then u may be written as $u_1\langle_a u_2$ and v as $v_1\langle_b v_2$, such that $|u_1| = |v_1|$. Furthermore, the following identities hold:*

$$|\mathfrak{p}_1(s_1)|_a = c_a(u_1) \qquad |\mathfrak{p}_1(s_2)|_a = c_a(u_2) \qquad |\mathfrak{p}_2(s_1)|_a = c_a(v_1) \qquad |\mathfrak{p}_2(s_2)|_a = c_a(v_2).$$

Proof. By induction on u, making the inductive case about the last letter. $\square$

Using (9), we u_1, u_2, v_1, v_2. We pick a name c, fresh for both u_2 and v_2. This implies that c is α-fresh for u_2 and v_2. Remark that here we need an unbounded number of atoms. We may now form the following derivations:

$$\dfrac{\dfrac{a \diamond u_2 \quad c \#_\alpha u_2}{\langle_a u_2 {}_a\rangle =_\alpha \langle_c (a\ c) \cdot u_2 {}_c\rangle}(\alpha\alpha)}{u_1 \langle_a u_2 {}_a\rangle =_\alpha u_1 \langle_c (a\ c) \cdot u_2 {}_c\rangle}(\alpha\mathbf{l}) \qquad \dfrac{\dfrac{b \diamond v_2 \quad c \#_\alpha v_2}{\langle_b v_2 {}_b\rangle =_\alpha \langle_c (b\ c) \cdot v_2 {}_c\rangle}(\alpha\alpha)}{v_1 \langle_b v_2 {}_b\rangle =_\alpha v_1 \langle_c (b\ c) \cdot v_2 {}_c\rangle}(\alpha\mathbf{l})$$

Since we already proved completeness, we can convert these derivations into paths in the transducer:

$$[] -[u_1 \langle_a u_2 {}_a\rangle / u_1 \langle_c (a\ c) \cdot u_2 {}_c\rangle] \rightarrow_{\mathcal{T}} t \in \mathbb{S}^{acc} \qquad [] -[v_1 \langle_b v_2 {}_b\rangle / v_1 \langle_c (b\ c) \cdot v_2 {}_c\rangle] \rightarrow_{\mathcal{T}} t' \in \mathbb{S}^{acc}$$

Using the tools developed for the transitivity part of our completeness proof, we may combine these with the path $[] -[u_1 \langle_a u_2 {}_a\rangle / v_1 \langle_b v_2 {}_b\rangle] \rightarrow_{\mathcal{T}} s' \in \mathbb{S}^{acc}$ into a single path

$$[] -[u_1 \langle_c (a\ c) \cdot u_2 {}_c\rangle / v_1 \langle_c (b\ c) \cdot v_2 {}_c\rangle] \rightarrow_{\mathcal{T}} (t \bowtie s') \bowtie t'.$$

Unravelling the last step of this path, we can check that we obtain a path:

$$[] -[u_1 \langle_c (a\ c) \cdot u_2 / v_1 \langle_c (b\ c) \cdot v_2] \rightarrow_{\mathcal{T}} s'' \in \mathbb{S}^{acc}.$$

We may therefore conclude:

$$\dfrac{\dfrac{\dfrac{a \diamond u_2 \quad c \#_\alpha u_2}{\langle_a u_2 {}_a\rangle =_\alpha \langle_c (a\ c) \cdot u_2 {}_c\rangle}(\alpha\alpha)}{u_1 \langle_a u_2 {}_a\rangle =_\alpha u_1 \langle_c (a\ c) \cdot u_2 {}_c\rangle}(\alpha\mathbf{l}) \quad \dfrac{\dfrac{\dfrac{[] -[u_1 \langle_c (a\ c) \cdot u_2 / v_1 \langle_c (b\ c) \cdot v_2] \rightarrow_{\mathcal{T}} s'' \in \mathbb{S}^{acc}}{u_1 \langle_c (a\ c) \cdot u_2 =_\alpha v_1 \langle_c (b\ c) \cdot v_2}(I.H.)}{u_1 \langle_c (a\ c) \cdot u_2 {}_c\rangle =_\alpha v_1 \langle_c (b\ c) \cdot v_2 {}_c\rangle}(\alpha\mathbf{r}) \quad \dfrac{\dfrac{b \diamond v_2 \quad c \#_\alpha v_2}{\langle_b v_2 {}_b\rangle =_\alpha \langle_c (b\ c) \cdot v_2 {}_c\rangle}(\alpha\alpha)}{v_1 \langle_b v_2 {}_b\rangle =_\alpha v_1 \langle_c (b\ c) \cdot v_2 {}_c\rangle}(\alpha\mathbf{l})}{u_1 \langle_c (a\ c) \cdot u_2 {}_c\rangle =_\alpha v_1 \langle_b v_2 {}_b\rangle}(\alpha\mathbf{t})}{u_1 \langle_a u_2 {}_a\rangle =_\alpha v_1 \langle_b v_2 {}_b\rangle}(\alpha\mathbf{t})$$

3.4 Decision procedure

Thanks to Theorem 3.1, we know that we can obtain an algorithm to decide α-equivalence if we can implement the transitions of the transducer. The task is as follows: given an input $\langle s, l_1, l_2\rangle \in \mathbb{S} \times \Sigma \times \Sigma$, we need to check if there exists a transition from s labelled with l_1/l_2, and compute the successor stack. We can make a case analysis on the pair $\langle l_1, l_2\rangle$:

- if the letters are of different "types" (e.g. $\langle_a$ and ${}_b\rangle$), reject;
- if the letters are of the shape $\langle_a, \langle_b$, return $s :: \langle a, b\rangle$;
- if the letters are of the shape ${}_a\rangle$, ${}_b\rangle$, check whether $s \models \langle a, b\rangle$:
  - if it does, return $s \ominus \langle a, b\rangle$;
  - otherwise, reject;
- finally, if $l_1 = x$ and $l_2 = y$, with $x, y \in \mathbb{X}$, we need to check if there exists a permutation π such that $y = \pi \cdot x$ and $\forall a \in \mathbf{supp}(x)$, $s \models \langle a, \pi(a)\rangle$. If π exists, we return s, otherwise we reject the input.

Checking $s \models \langle a, b\rangle$ is straightforward with a linear algorithm, as is computing $s \ominus \langle a, b\rangle$. Therefore, the only missing ingredient is a way to check for the existence of a permutation sending one variable to another, and compatible with a given stack.

Algorithm 1: $\texttt{var_perm}(s, A)$

```
Input: s ∈ 𝕊 and A ∈ P_f(𝔸);
Output: Either ⊥ to signify failure or a permutation in 𝔖_𝔸;
match s :
    case [] : return Id_𝔸;
    case s :: ⟨a, b⟩ :
        match var_perm(s, A \ a) :
            case ⊥ : return ⊥;
            case π :
                if b ∈ π · (A \ a) then return ⊥;
                else if a ∈ A then return (π(a) b) ∘ π;
                else return π;
```

The key observation for the algorithm is the following: given a stack s and a name a, there is at most one b such that $s \models \langle a, b\rangle$. This b may be computed in linear-time as $\texttt{image}(s, a)$:

$$\texttt{image}([], a) := a \qquad \texttt{image}(s :: \langle b, c\rangle, a) := \begin{cases} c & \text{if } a = b \\ \texttt{image}(s, a) & \text{otherwise.} \end{cases}$$

Notice that it is not always the case that $s \models \langle a, \texttt{image}(s, a)\rangle$: for instance consider the stack $s = [] :: \langle a, b\rangle :: \langle c, b\rangle$; we have $\texttt{image}(s, a) = b$, but $s \not\models \langle a, b\rangle$. However, if $s \models \langle a, b\rangle$, then b must be equal to $\texttt{image}(s, a)$.

If π is a permutation witnessing $s \;{-[x/y]\to_{\mathcal{T}}}\; s$, by the compatibility condition for each $a \in \mathbf{supp}(x)$ we know that $\pi(a) = \texttt{image}(s, a)$. Since $\pi \cdot x$ is uniquely determined by the values of π on $\mathbf{supp}(x)$, and since the compatibility condition depends upon the same values, we may check the validity of the transition as follows:

1. for each $a \in \mathbf{supp}(x)$, compute $\texttt{image}(s, a)$, and check if $s \models \langle a, \texttt{image}(s, a)\rangle$;
2. pick any permutation $\pi_s^{\mathbf{supp}(x)}$ such that $\forall a \in \mathbf{supp}(x), \pi_s^{\mathbf{supp}(x)}(a) = \texttt{image}(s, a)$;
3. check that $y = \pi_s^{\mathbf{supp}(x)} \cdot x$.

The first two steps may be done in one sweep of the stack s using $\texttt{var_perm}(-, -)$, as defined in Algorithm 1. The effect of this function can be summarised by the following observations:

$$\texttt{var_perm}(s, A) = \bot \Leftrightarrow \exists a \in A : \forall b, s \not\models \langle a, b\rangle \tag{10}$$

$$\exists \pi : \texttt{var_perm}(s, A) = \pi \Rightarrow \forall a \in A, s \models \langle a, \pi(a)\rangle\,. \tag{11}$$

From these we deduce that $s \;{-[x/y]\to_{\mathcal{T}}}\; s$ if and only if $\texttt{var_perm}(s, \mathbf{supp}(x))$ is a permutation π and $y = \pi \cdot x$. We now use this to formulate Algorithm 2 to decide α-equivalence. This algorithm uses linear space.

3.5 The bisimulation collapse

In this section we investigate the properties of the transducer $\mathcal{T}$, considered as a labelled transition system. We show that several ways of comparing states of the system (i.e. stacks) happen to coincide: simulation, bisimulation, semantic equivalence, semantic containment... In this section, we call *labels* pairs of letters, and let α, β range over labels, and *traces* pairs of words, using σ, τ

Algorithm 2: Decision procedure for α-equivalence

```
Input: A pair of words ⟨u, v⟩;
Output: A boolean corresponding to the property u =_α v;
if |u| ≠ |v| then return false;                          /* By (8), u ≠_α v. */
s ← [];                                            /* Start with the empty stack. */
for i = 1 to |u| do                                     /* Scan the pairs ⟨u_i, v_i⟩. */
    match u_i, v_i :
        case ⟨_a , ⟨_b : s ← s :: ⟨a, b⟩;
        case _a⟩, _b⟩ :
            if s ⊭ ⟨a, b⟩ then return false;
            s ← s ⊖ ⟨a, b⟩;
        case x, y :
            match var_perm (s, supp(x)) :
                case ⊥ : return false;
                case π :
                    if y ≠ π · x then return false;
        otherwise : return false;
/* After the main loop we have ⊥ −[u/v]→_T s.                                      */
foreach ⟨a, b⟩ ∈ s do                                     /* Check if s ∈ S^acc. */
    if a ≠ b then return false;
return true ;                                /* Now we have s ∈ S^acc, hence u =_α v. */
```

to denote traces. The semantics of a stack s, written $[\![s]\!]$, is the set of traces labelling runs from s to an accepting stack: $[\![s]\!] := \{\sigma \in \Sigma^\star \times \Sigma^\star \mid \exists s' \in \mathbb{S}^{acc} : s \xrightarrow{[\sigma]}_{\mathcal{T}} s'\}$.

The stacks s_1 and s_2 are said to be *similar*, written $s_1 \preccurlyeq s_2$, if for any transition from s_1 there is a transition from s_2 with the same label, and the target stacks are similar.

$$(\forall \langle \alpha, s\rangle \in (\Sigma \times \Sigma) \times \mathbb{S}, s_1 \xrightarrow{[\alpha]}_{\mathcal{T}} s \Rightarrow \exists s', s_2 \xrightarrow{[\alpha]}_{\mathcal{T}} s' \wedge s \preccurlyeq s') \Rightarrow s_1 \preccurlyeq s_2.$$

Bisimilarity is defined with an analogous coinductive definition: to show that $s_1 \sim s_2$ we need to prove that for any label α and any stack s:

1. if $s_1 \xrightarrow{[\alpha]}_{\mathcal{T}} s$, there is a stack s' such that $s \sim s'$ and $s_2 \xrightarrow{[\alpha]}_{\mathcal{T}} s'$.
2. if $s_2 \xrightarrow{[\alpha]}_{\mathcal{T}} s$, there is a stack s' such that $s' \sim s$ and $s_1 \xrightarrow{[\alpha]}_{\mathcal{T}} s'$.

Some simple observations about the definitions we have so far are listed below:

Lemma 3.7. *Bisimilarity is an equivalence relation. Similarity is a preorder. Bisimilarity implies similarity.*

We may also characterise bisimilarity in terms of path.

Lemma 3.8. *Let s, s' be a pair of stacks, $s \sim s'$ if and only if for any trace σ we have:*

$$(\exists t : s \xrightarrow{[\sigma]}_{\mathcal{T}} t) \Leftrightarrow (\exists t : s' \xrightarrow{[\sigma]}_{\mathcal{T}} t).$$

Proof (Sketch). The proof uses the following fact:

$$s \sim s' \text{ and } s \xrightarrow{[\sigma]}_{\mathcal{T}} t \Rightarrow \exists t' : s' \xrightarrow{[\sigma]}_{\mathcal{T}} t' \text{ and } t \sim t'. \tag{12}$$

This may be shown by induction on the path $s \xrightarrow{[\sigma]}_{\mathcal{T}} t$.

For the left-to-right implication, we may simply apply (12). For the converse direction, we proceed by coinduction, relying on the fact that $-[]\rightarrow_{\mathcal{T}}$ is deterministic in the sense that if $s -[\sigma]\rightarrow_{\mathcal{T}} s_1$ and $s -[\sigma]\rightarrow_{\mathcal{T}} s_2$ then $s_1 = s_2$. □

We call two stacks *statically equivalent*, and write $s == s'$, if they are related by the following inductive relation: $s \in \mathbb{S}^{acc} \Rightarrow [] == s$ and $s' \models \langle a,b\rangle$ and $s == s' \ominus \langle a,b\rangle \Rightarrow s :: \langle a,b\rangle == s'$. Note that this last relation is straightforward to decide: it is defined by induction on the first argument. Before we can show that this relation is indeed an equivalence, we make the following observation:

$$s == s' \Rightarrow (s \models \langle a,b\rangle \Leftrightarrow s' \models \langle a,b\rangle)\,. \tag{13}$$

Lemma 3.9. $==$ *is an equivalence relation.*

Proof. Reflexivity is straightforward. To prove the relation symmetric, assuming $s_1 == s_2$ we first do an induction on s_2. The base case being straightforward, we need to show that if $s_1 \models \langle a,b\rangle$ and $s_1 == s_2 :: \langle a,b\rangle$, then $s_1 \ominus \langle a,b\rangle == s_2$. This is shown by induction on s_1. Transitivity is also shown by induction on stacks, relying on (13), the fact that the relation is symmetric, and the fact that $\ominus$ is monotonic with respect to $==$. □

It will also be convenient to associate to each stack s a characteristic trace $tr(s)$, defined inductively as follows: $tr([]) := (\varepsilon,\varepsilon)$ and $tr(s :: (a,b)) := ({}_{a}\rangle\, tr(s)_1\,,\ {}_{b}\rangle\, tr(s)_2)$. For instance $tr([] :: \langle a,b\rangle :: \langle c,d\rangle)_1 = {}_{c}\rangle\ {}_{a}\rangle$ and $tr([] :: \langle a,b\rangle :: \langle c,d\rangle)_2 = {}_{d}\rangle\ {}_{b}\rangle$. We also have that $tr(s) \in [\![s]\!]$.

Theorem 3.10. *For any pair of stacks s_1, s_2, the following are equivalent: (i) $s_1 \preccurlyeq s_2$; (ii) $s_1 \sim s_2$; (iii) $[\![s_1]\!] = [\![s_2]\!]$; (iv) $[\![s_1]\!] \subseteq [\![s_2]\!]$; (v) $s_1 == s_2$; (vi) $tr(s_1) \in [\![s_2]\!]$.*

Proof (Sketch). The strategy to prove this is outlined in Figure 1. The implications labelled with (0) hold trivially. Those labelled with (1) are proved by induction on s_1. (2) is proved by coinduction on the bisimilarity predicate. (3) is relatively straightforward by induction on paths. (4) was shown in Lemma 3.7. □

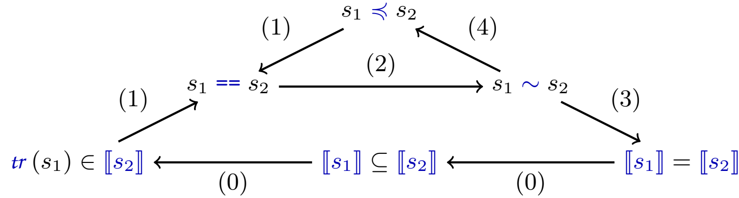


Fig. 1: Diagram of the proofs

3.6 Quantitative measures of memory consumption

In the next section, we will look at languages and provide a classification of their expressive power. One of the things we will consider is a measure on how much memory words in the language need (intuitively, think of how many resources are allocated at a certain point in the program). The following definitions and result over binding monoids and words will be useful.

We define the *size* of a binding element $b \in \mathcal{B}$ as $|\mathbf{d}^m\mathbf{f}^n\mathbf{c}^p| = m + p$[3]. This is well defined since any element of $\mathcal{B}$ has a unique normal form of the shape $\mathbf{d}^m\mathbf{f}^n\mathbf{c}^p$, where $m, p \in \mathbb{N}$ and $p \in \{0,1\}$ [2]. Binding monoids have an interesting boundedness property:

[3] Since the size of a Boolean is constant, we do not count n in the size of $\mathbf{d}^m\mathbf{f}^n\mathbf{c}^p$. This simplifies a number of computations.

Lemma 3.11. *For any number $N \in \mathbb{N}$, there is only finitely many elements of the binding monoid with size less than N, i.e. we have that $\mathcal{B}^{\leqslant N} := \{\beta \in \mathcal{B} \mid |\beta| \leqslant N\}$ is finite.*

The *weight* of a word u is the sum of the sizes of its binding powers: $\|u\| := \sum_{a\in\mathbb{A}} |\mathcal{F}_a(u)|$. This sum is finite, since for every name a outside the finite set $\mathbf{supp}(u)$ we know that the binding power of u with respect to a is ε, so $|\mathcal{F}_a(u)| = 0$. The *memory* of a word u is the maximum weight of a prefix of u, i.e. $\mathbf{m}(u) := \max_{vw=u} \|v\|$.

From Lemma 3.3 we deduce that if $[] -[u/v]\rightarrow_{\mathcal{T}} s$, then $|s| \leqslant \|u\|$. Furthermore, every stack visited in the intermediate steps has length less than $\mathbf{m}(u)$.

4 Languages up-to α-equivalence

In this section we extend the study of the previous section from words to languages over alphabets with binders. We start by extending the definitions of Section 3.1 to languages (Section 4.1). We then study *regular languages over* Σ and provide a characterisation of two well-behaved subclasses – memory-finite and binding-finite languages. The former that are those regular languages such that the memory of any word in the language is bounded by a fixed N (Section 4.2).

In classical formal language theory the hierarchy between languages classes is very clear and expressed equivalently in terms of their syntactic description (regular expressions, context-free grammars, etc) or the acceptance power of corresponding finite state machines (finite (non)-deterministic automata, push-down automata, etc). In the presence of nominal alphabets the characterisation of various classes of languages is not as straightforward and as it turns out the hierarchy is much finer. In this section we will present a classification of languages that takes into account several properties stemming from their nominal nature – e.g. finite support vs bounded support – and from their binding power. We show that variations on these generate very different classes of languages and organise those in a hierarchy (Section 5).

4.1 Lifting the binding structure to languages

In this section, a language is simply a subset of $\Sigma^\star$. Since we study α-equivalence, we will be interested in languages L that do not distinguish α-equivalent words, in the sense that if $u =_\alpha v$ then $u \in L$ if and only if $v \in L$.

The simplest way to obtain such languages is to take any language, and close it by α-equivalence. Formally, the *α-closure* of a language L is defined as

$$L^\alpha := \{u \in \Sigma^\star \mid \exists v \in L,\ u =_\alpha v\}.$$

This enables us to define α-equivalence of languages: two languages are equivalent if their α-closures are equal.

We may also lift binding powers, weights and memory to languages, in the following way:

$$\mathcal{F}_a(L) := \{\mathcal{F}_a(u) \mid u \in L\} \qquad \|L\| := \sup_{u\in L} \|u\| \qquad \mathbf{m}(L) := \sup_{u\in L} \mathbf{m}(u).$$

Notice that for a regular language $L = [\![e]\!]$, $\mathbf{m}(L) = \|[\![\mathbf{prefix}(e)]\!]\|$.

We will write $\mathcal{M}_\Sigma^{\leqslant N}$ for the language of all words over the finite alphabet Σ with memory less than N:

$$\mathcal{M}_\Sigma^{\leqslant N} := \{u \in \Sigma^\star \mid \mathbf{m}(u) \leqslant N\}.$$

For technical reasons, we will actually need another family of languages, that only bound the quantity $c_a(u)$ for all prefixes u.

$$\mathcal{C}_{\Sigma}^{\leqslant N} := \left\{ w \in \Sigma^\star \;\middle|\; \forall u \in \mathbf{prefix}(w), \sum_a c_a(u) \leqslant N \right\}.$$

Notice that $\mathcal{M}_{\Sigma}^{\leqslant N} \subseteq \mathcal{C}_{\Sigma}^{\leqslant N}$, since $c_a(u) \leqslant |\mathcal{F}_a(u)|$.

4.2 Nominal regular languages

In this section we consider regular languages over Σ, i.e. languages $[\![e]\!]$ for some expression $e \in \mathrm{Reg}\langle\Sigma\rangle$. For convenience, we will sometimes identify expressions with their language, writing for instance $\mathcal{F}_a(e)$ instead of $\mathcal{F}_a([\![e]\!])$. We lift the support function from Σ to $\mathrm{Reg}\langle\Sigma\rangle$ in the canonical way: for letters in Σ we use the $\mathbf{supp}(-)$ function from Section 3.1, the support of 0 and 1 is the empty set, the support of $e^\star$ is that of e and the support of both $e+f$ and $e \cdot f$ is $\mathbf{supp}(e) \cup \mathbf{supp}(f)$. This definition corresponds to the pointwise lifting of the support function on words: indeed if e does not contain 0, we have $\mathbf{supp}(e) = \bigcup_{u \in [\![e]\!]} \mathbf{supp}(u)$. Note that $\mathbf{supp}(e)$ is always finite, and supports $[\![e]\!]$ in the sense that for any permutation π that does not modify any name inside $\mathbf{supp}(e)$, we have $\pi \cdot [\![e]\!] = [\![e]\!]$.

Characterisation of memory-finite languages We will now characterise those regular languages that are memory finite, meaning there exists a natural number N such that the memory of any word in the language is less than N. Towards that end, we prove the following lemmas.

Lemma 4.1. *For any two non-empty sets $A, B \subseteq \mathcal{B}$, the set $A \cdot B$ is finite if and only if both A and B are finite.*

Proof. The right to left implication being obvious, assume $A \cdot B$ is finite, and let us choose $\alpha_0 \in A$. By definition, the set $\{\alpha_0\} \cdot B$ is contained in $A \cdot B$, therefore it must be finite. This means this set has bounded size, in the sense that there exists a number $N \in \mathbb{N}$ such that $\gamma \in \alpha_0 \cdot B$ entails $|\gamma| \leqslant N$. Thus for each $\beta \in B$ we have $|\alpha_0 \cdot \beta| \leqslant N$. To conclude, notice that the following inequalities hold for every pair $\langle \alpha, \beta \rangle \in \mathcal{B} \times \mathcal{B}$:

$$|\alpha| \leqslant |\alpha \cdot \beta| + |\beta| \qquad\qquad |\beta| \leqslant |\alpha \cdot \beta| + |\alpha|.$$

Therefore, we get that $B \subseteq \mathcal{B}^{\leqslant N + |\alpha_0|}$ which is finite (see the remark at the beginning of Section 3.6), hence B itself must be finite. A symmetric argument shows that A is also finite. □

Lemma 4.2. *Let $A \subseteq \mathcal{B}$, the set $A^\star$ is finite if and only if A consists of a finite number square bindings, i.e. of elements of the shape $\mathbf{d}^n\mathbf{f}^p\mathbf{c}^n$. Furthermore, in this case $A^\star = A \cup \{\varepsilon\}$.*

Proof. Since $A \subseteq A^\star$, if $A^\star$ is finite so is A. Consider now the product of $\alpha = \mathbf{d}^k\mathbf{f}^n\mathbf{c}^k$ and $\beta = \mathbf{d}^{k'}\mathbf{f}^{n'}\mathbf{c}^{k'}$.

$$\mathbf{d}^k\mathbf{f}^n\mathbf{c}^k \cdot \mathbf{d}^{k'}\mathbf{f}^{n'}\mathbf{c}^{k'} = \begin{cases} \mathbf{d}^k\mathbf{f}^n\mathbf{c}^{k'+k-k'} = \alpha & k > k' \\ \mathbf{d}^{k+k'-k}\mathbf{f}^{n'}\mathbf{c}^{k'} = \beta & k < k' \\ \mathbf{d}^k\mathbf{f}^{\max(n,n')}\mathbf{c}^{k'} \in \{\alpha, \beta\} & k = k' \end{cases}$$

This implies that if A consists of a finite number of elements of the shape $\mathbf{d}^k\mathbf{f}^n\mathbf{c}^k$, then $A^\star = A \cup \{\varepsilon\}$. If on the other hand there exists $\beta = \mathbf{d}^m\mathbf{f}^n\mathbf{c}^p \in A$ with $m \neq p$, we have:

$$(\mathbf{d}^m\mathbf{f}^n\mathbf{c}^p)^k = \begin{cases} \mathbf{d}^m\mathbf{f}^n\mathbf{c}^{p+k\times(p-m)} & p > m \\ \mathbf{d}^{m+k\times(m-p)}\mathbf{f}^n\mathbf{c}^p & p < m \end{cases}$$

In both cases, the set of powers of β is infinite, thus $A^\star$ is infinite. □

We can now give a characterisation of memory-finite regular languages.

Proposition 4.3 (Memory-finiteness). *Let* $e \in Reg\langle\Sigma\rangle$*.* $[\![e]\!]$ *is memory-finite if and only if for any name* $a \in \mathbb{A}$ *the set* $\mathcal{F}_a([\![e]\!])$ *is finite.*

Proof. The left-to-right implication is straightforward: if $[\![e]\!]$ is memory-finite there must be a bound $N \in \mathbb{N}$ such that $\forall u, v, \in \Sigma^\star, uv \in [\![e]\!] \Rightarrow \|u\| \leqslant N$. For any $u \in [\![e]\!]$ and $a \in \mathbb{A}$, by definition of the weight of a word we know that $|\mathcal{F}_a(u)| \leqslant \|u\|$, hence $|\mathcal{F}_a(u)| \leqslant N$. Therefore, $\mathcal{F}_a(e)$ is contained in $\mathcal{B}^{\leqslant N}$, which is finite.

For the other direction, we proceed by induction on e. If e is a letter or a constant, since its language is finite the result holds trivially. In the case $e = f_1 + f_2$, we have $\mathcal{F}_a(e) = \mathcal{F}_a(f_1) \cup \mathcal{F}_a(f_2)$, so if for every $a \in \mathbb{A}$ $\mathcal{F}_a(e)$ is finite then both $\mathcal{F}_a(f_1)$ and $\mathcal{F}_a(f_2)$. Using the induction hypothesis we get that both f_1 and f_2 are memory finite, and since $\mathbf{m}(e) = \max(\mathbf{m}(f_1), \mathbf{m}(f_2))$ so is e. If $e = f_1 \cdot f_2$ and $\mathcal{F}_a(e)$ is finite for any $a \in \mathbb{A}$, there are two possibilities:

(i) either $[\![e]\!]$ is empty, in which case e is trivially memory finite;
(ii) or both $\mathcal{F}_a(f_1)$ and $\mathcal{F}_a(f_2)$ are non-empty, then according to Lemma 4.1 and the induction hypothesis both f_1 and f_2 are memory finite. Hence e must be memory-finite, since we have:

$$\begin{aligned}\mathbf{m}(e) = \|\mathbf{prefix}(e)\| &= \|\mathbf{prefix}(f_1) \cup f_1 \cdot \mathbf{prefix}(f_2)\| \\ &= \max(\mathbf{m}(f_1), \|f_1 \cdot \mathbf{prefix}(f_2)\|) \\ &\leqslant \max(\mathbf{m}(f_1), \|f_1\| + \mathbf{m}(f_2)) \leqslant \mathbf{m}(f_1) + \mathbf{m}(f_2).\end{aligned}$$

For the last case, consider $e = f^\star$. Because of Lemma 4.2 if for every name $a \in \mathbb{A}$ the set $\mathcal{F}_a(e) = \mathcal{F}_a(f^\star)$ is finite, then the set $\mathcal{F}_a(f)$ must be finite, hence by the induction hypothesis f is memory-finite. We can also check that $\|f^\star\|$ is finite: since for any name a the set $\mathcal{F}_a(f^\star)$ is finite, there are bounds $N_a \in \mathbb{N}$ such that $\mathcal{F}_a(f^\star) \subseteq \mathcal{B}^{\leqslant N_a}$, so for any word $w \in [\![f^\star]\!]$ the following holds:

$$\|w\| = \sum_{a \in \mathbb{A}} |\mathcal{F}_a(w)| = \sum_{a \in \mathbf{supp}(f^\star)} |\mathcal{F}_a(w)| \leqslant \sum_{a \in \mathbf{supp}(f^\star)} N_a.$$

Hence we have a uniform bound on the weights of words in the language of $f^\star$, ensuring that $\|f^\star\|$ is finite. We may now conclude that e is memory finite, since:

$$\mathbf{m}(e) = \|\mathbf{prefix}(f^\star)\| = \|f^\star \cdot \mathbf{prefix}(f)\| \leqslant \|f^\star\| + \mathbf{m}(f). \qquad \square$$

Remark 4.4. This lemma could be reformulated by saying that $\mathcal{F}_a(e)$ is finite if and only if $\mathcal{F}_a(\mathbf{prefix}(e))$ is finite.

In what follows, we show that in fact one can prove linear bounds on the weight and memory of memory finite languages.

Lemma 4.5. *If* e *is memory finite, then* $\|e\| \leqslant |e|$ *and* $\mathbf{m}(e) \leqslant 2 \times |e|$*, where* $|e|$ *is the size of* e*, i.e. the number of occurrences of letters in* e*.*

Proof. First, we split the size of the expression as

$$\sum_{a \in \mathbf{supp}(e)} |e|_a \leqslant |e|,$$

where $|e|_a$ is the number of occurrences of letters $\langle_a$ and $_a\rangle$ in e. To get the bound on the weight of e, we simply prove by induction on e that if e is memory-finite and $\beta \in \mathcal{F}_a(e)$, then $|\beta| \leqslant |e|_a$.

For the other bound, we need a stronger induction hypothesis. What we end up showing by induction on e is that if e is memory-finite then for any name a, and every $\mathbf{d}^m\mathbf{f}^k\mathbf{c}^n \in \mathcal{F}_a(\mathbf{prefix}(e))$, we have: $m+n \leqslant 2|e|_a$ and $|m-n| \leqslant |e|_a$. This enables us to check that:

$$\begin{aligned}\mathbf{m}(e) = \|\mathbf{prefix}(e)\| = \sup_{u\in[\![\mathbf{prefix}(e)]\!]}\|u\| = \sup_{u\in[\![\mathbf{prefix}(e)]\!]}\sum_a |\mathcal{F}_a(u)| &\leqslant \sum_a \sup_{u\in[\![\mathbf{prefix}(e)]\!]}|\mathcal{F}_a(u)| \\ &= \sum_a \sup_{\beta\in\mathcal{F}_a(\mathbf{prefix}(e))}|\beta| \\ &\leqslant \sum_a 2|e|_a = 2\cdot\sum_a |e|_a \leqslant 2|e|.\end{aligned}$$

□

Characterisation of binding finite languages We first define the function $\mathbf{bindings} : \mathbb{A} \to \mathrm{Reg}\langle\Sigma\rangle \to \mathcal{P}_f(\mathcal{B})$, that computes the binding powers $\mathcal{F}_a(e)$. We say that e is binding-finite if $\mathcal{F}_a(e)$ is finite for all $a \in \mathbb{A}$ [4].

$$\begin{aligned}\mathbf{bindings}_a(0) &:= \emptyset \qquad \mathbf{bindings}_a(1) := \{\varepsilon\} \qquad \mathbf{bindings}_a(l) := \{\mathcal{F}_a(l)\}\\ \mathbf{bindings}_a(e+f) &:= \mathbf{bindings}_a(e) \cup \mathbf{bindings}_a(f)\\ \mathbf{bindings}_a(e\cdot f) &:= \mathbf{bindings}_a(e) \cdot \mathbf{bindings}_a(f)\\ \mathbf{bindings}_a(e^\star) &:= \{\varepsilon\} \cup \{\beta \in \mathbf{bindings}_a(e) \mid \beta \text{ is square}\}\end{aligned}$$

The two main properties of this function are summarised in Lemma 4.6.

Lemma 4.6. *Let $e \in \mathrm{Reg}\langle\Sigma\rangle$ and $a \in \mathbb{A}$, then $\mathbf{bindings}_a(e) \subseteq \mathcal{F}_a(e)$. If e is binding-finite, then we also have $\mathcal{F}_a(e) \subseteq \mathbf{bindings}_a(e)$.*

Proof. The first containment is easily checked by an induction on e. The second one is not much more difficult, essentially relying on Lemmas 4.1 and 4.2. □

This allows us to define a Boolean predicate $\mathbf{isBF}(e)$ to check binding-finiteness on an expression e:

$$\begin{aligned}\mathbf{isBF}(0) = \mathbf{isBF}(1) = \mathbf{isBF}(l) &:= \top\\ \mathbf{isBF}(e+f) &:= \mathbf{isBF}(e) \wedge \mathbf{isBF}(f)\\ \mathbf{isBF}(e\cdot f) &:= \mathbf{isZero}(e) \vee \mathbf{isZero}(f) \vee (\mathbf{isBF}(e) \wedge \mathbf{isBF}(f))\\ \mathbf{isBF}(e^\star) &:= \mathbf{isBF}(e) \wedge (\forall a \in \mathbf{supp}(e), \forall\beta \in \mathbf{bindings}_a(e), \beta \text{ is square}).\end{aligned}$$

Lemma 4.7 (Binding-Finiteness). *An expression $e \in \mathrm{Reg}\langle\Sigma\rangle$ is binding-finite if and only if $\mathbf{isBF}(e) = \top$.*

Proof. We proceed by case analysis on $\mathbf{isBF}(e)$:

- if $\mathbf{isBF}(e) = \top$, we show that for any $a \in \mathbb{A}$ we have $\mathcal{F}_a(e) \subseteq \mathbf{bindings}_a(e)$, thus ensuring that $\mathcal{F}_a(e)$ is finite;
- if $\mathbf{isBF}(e) = \bot$ we prove that there exists $a \in \mathbb{A}$ such that for any $N \in \mathbb{N}$, there is $u \in [\![e]\!]$ such that $|\mathcal{F}_a(u)| \geqslant N$. □

[4] In the next section we will refine this notion further for languages and have a weak and strong variant of binding-finiteness.

5 Language Classification

In this section, we compare various natural notions of tractability for languages over Σ. Some of those are classic nominal notions, having to do with the support of the language, and others are new notions, related to binding power and memory, two of which we already characterised in detail in the previous section. We will completely map the Boolean algebra generated by these predicates: we prove which containments hold, and provide examples to inhabit every non-empty boolean combination of these predicates.

It is worth noting that we do not discuss in this paper the automata corresponding to these classes of languages. In a separate companion paper [1], we present a Kleene theorem for non-deterministic nominal automata.

5.1 Definitions

We denote by **reg** the set of languages L such that $L = [\![e]\!]^{\alpha}$ for some expression $e \in \mathrm{Reg}\langle\Sigma\rangle$. We say that a permutation $\pi \in \mathfrak{S}_{\mathbb{A}}$ fixes a set of atoms $A \subseteq \mathbb{A}$ when $\forall a \in A, \pi(a) = a$.

Finite support We will consider four classes of languages based on their support.

bs : L has bounded support if there exists a finite set $A \subseteq \mathbb{A}$ such that

$$\bigcup_{u \in L} \mathbf{supp}(u) \subseteq A.$$

αbs : L has bounded α-support if there exists a finite set $A \subseteq \mathbb{A}$ such that

$$\bigcup_{u \in L} \mathbf{supp}_{\alpha}(u) \subseteq A.$$

fs : L is finitely supported if there is a finite set A such that if π fixes A then $\pi \cdot L = L$, meaning:

$$\forall u \in \Sigma^{\star}, \pi \cdot u \in L \Leftrightarrow u \in L.$$

αfs : L is finitely supported up-to α if there is a finite set A such that if π fixes A then $\pi \cdot L^{\alpha} = L^{\alpha}$, meaning:

$$\forall u \in \Sigma^{\star}, \pi \cdot u \in L^{\alpha} \Leftrightarrow u \in L^{\alpha}.$$

Binding classes We now consider five classes of languages based on properties of their binding power, memory, and weight.

ubf : L is uniformly binding finite if the set $\mathbb{F}L$ is finite, where

$$\mathbb{F}L := \{[a \mapsto \mathcal{F}_a(u)] \mid u \in L\}.$$

mf : L is memory-finite if there is a number $N \in \mathbb{N}$ such that $u \in L \Rightarrow \mathbf{m}(u) \leqslant N$.
bw : L has bounded weight if there is a number $N \in \mathbb{N}$ such that $u \in L \Rightarrow \|u\| \leqslant N$.
bf' : L is binding finite in the strong sense if the set $\mathcal{F}L$ is finite, where

$$\mathcal{F}L := \{\mathcal{F}_a(u) \mid \langle u, a\rangle \in L \times \mathbb{A}\}.$$

bf : L is binding finite in the weak sense if for any name $a \in \mathbb{A}$, the set $\mathcal{F}_a(L)$ is finite.

Next, we first study in detail how all these predicates and classes of languages relate to each other (Section 5.2) and then provide a long list of concrete examples of languages that show how strict some of these relations are (Section 5.3).

5.2 Inclusions

The results of this section, together with what we showed in Section 4.2, are summarised in Figure 2. The dashed arrows indicate implications that hold in the presence of **bf** : for instance the arrow from **reg** to **mf** represents on direction of Proposition 4.3, that is **reg** $\wedge$ **bf** $\Rightarrow$ **mf**.

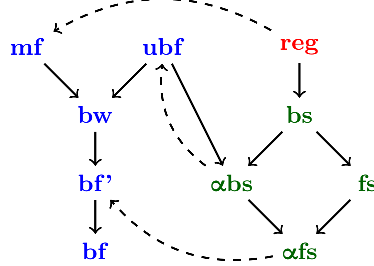


Fig. 2: Summary of the inclusions of tractability predicates

We may notice from this picture that for a regular language $L := [\![e]\!]^{\alpha}$, there are only two possibilities: either L is binding finite, in which case it enjoys all of these predicates, or it is not, in which case it only satisfies **bs** , **fs** , **αbs** , and **αfs** .

Lemma 5.1. ***ubf*** $\Rightarrow$ ***αbs***.

Proof. We show that $\mathbf{supp}_{\alpha}(L)$ is finite. First, notice that

$$\mathbf{supp}_{\alpha}(L) = \bigcup_{u \in L} \{a \in \mathbb{A} \mid \mathcal{F}_a(u) \neq \varepsilon\} = \bigcup_{f \in \mathbb{F}L} \{a \in \mathbb{A} \mid f(a) \neq \varepsilon\}.$$

For all $f \in \mathbb{F}L$ the set $\{a \in \mathbb{A} \mid f(a) \neq \varepsilon\}$ is finite since there exists $u \in L$ such that we have $f = [a \mapsto \mathcal{F}_a(u)]$ and so $\{a \in \mathbb{A} \mid f(a) \neq \varepsilon\} = \mathbf{supp}_{\alpha}(u)$ which is always finite. □

Lemma 5.2. ***ubf*** $\Rightarrow$ ***bw***.

Proof. Let $L \in \mathbf{ubf}$, we have:

$$\|L\| = \sup_{u \in L} \|u\| = \sup_{u \in L} \sum_{a \in \mathbb{A}} |\mathcal{F}_a(u)| = \sup_{u \in L} \sum_{a \in \mathbf{supp}_{\alpha}(L)} |\mathcal{F}_a(u)| = \max_{f \in \mathbb{F}L} \sum_{a \in \mathbf{supp}_{\alpha}(L)} |fu|.$$

We know $\mathbb{F}L$ to be finite, so by Lemma 5.1 $\mathbf{supp}_{\alpha}(L)$ is as well, thus $\|L\|$ is bounded. □

Lemma 5.3. ***bw*** $\Rightarrow$ ***bf'***.

Proof. Since $\|L\| = \sup_{u \in L} \|u\|$ and $\|u\| = \sum_{a \in \mathbb{A}} |\mathcal{F}_a(u)|$, we know that $|\mathcal{F}_a(u)| \leqslant \|u\| \leqslant \|L\|$. Therefore, $\mathcal{F}L \subseteq \mathcal{B}^{\leqslant \|L\|}$ which we know to be finite from Section 3.6. □

Lemma 5.4. ***bf'*** $\Rightarrow$ ***bf***.

Proof. For any $a \in \mathbb{A}$, we have $\mathcal{F}_a(L) \subseteq \mathcal{F}L$ so if the latter is finite, so is the former. □

Lemma 5.5. ***mf*** $\Rightarrow$ ***bw***.

Proof. By definition we have $\mathbf{m}(L) = \|\mathbf{prefix}(L)\| = \sup\{\|u\| \mid u \in \mathbf{prefix}(L)\}$. Since $L \subseteq \mathbf{prefix}(L)$ and sup is monotone, we get $\mathbf{m}(L) \geqslant \|L\|$. Therefore if $\mathbf{m}(L)$ is finite so is $\|L\|$. □

Lemma 5.6. ***bs*** $\Rightarrow$ ***abs***.

Proof. Assume we have a finite set $A \subseteq \mathbb{A}$ such that $\bigcup_{u \in L} \mathbf{supp}(u) \subseteq A$. Since for all words u we have $\mathbf{supp}_\alpha(u) \subseteq \mathbf{supp}(u)$, we get:

$$\bigcup_{u \in L} \mathbf{supp}_\alpha(u) \subseteq \bigcup_{u \in L} \mathbf{supp}(u) \subseteq A. \qquad \square$$

Lemma 5.7. ***bs*** $\Rightarrow$ ***fs***.

Proof. Assume we have a finite set $A \subseteq \mathbb{A}$ such that $\bigcup_{u \in L} \mathbf{supp}(u) \subseteq A$. If π fixes A, then for every word in $u \in L$ π fixes $\mathbf{supp}(u)$ so we have $\pi \cdot u = u$. This implies that we have $\pi \cdot L = \{\pi \cdot u \mid u \in L\} = L$. $\square$

Lemma 5.8. ***abs*** $\Rightarrow$ ***afs***.

Proof. Assume we have a finite set $A \subseteq \mathbb{A}$ such that $\bigcup_{u \in L} \mathbf{supp}_\alpha(u) \subseteq A$. If π fixes A, then for every word in $u \in L$, π fixes $\mathbf{supp}_\alpha(u)$ so we have $\pi \cdot u =_\alpha u$. This implies that

$$\begin{aligned} v \in \pi \cdot L^\alpha \Leftrightarrow \exists \langle v', u \rangle \in \Sigma^\star \times L : v = \pi \cdot v' \wedge v' =_\alpha u &\Leftrightarrow \exists u \in L : \pi^{-1} \cdot v =_\alpha u \\ &\Leftrightarrow \exists u \in L : v =_\alpha \pi \cdot u \\ &\Leftrightarrow \exists u \in L : v =_\alpha u \\ &\Leftrightarrow v \in L^\alpha. \end{aligned} \qquad \square$$

Lemma 5.9. ***fs*** $\Rightarrow$ ***afs***.

Proof. Assume we have a finite set A such that if π fixes A then $\pi \cdot L = L$, meaning: $\forall u \in \Sigma^\star$, $\pi \cdot u \in L \Leftrightarrow u \in L$. If π fixes A then $\pi \cdot L^\alpha = (\pi \cdot L)^\alpha = L^\alpha$. $\square$

Lemma 5.10. ***bf*** $\wedge$ ***afs*** $\Rightarrow$ ***bf'***.

Proof. Assume that we have a finite A such that for any $\pi \in \mathfrak{S}_\mathbb{A}$, if π fixes A then $\pi \cdot L^\alpha = L^\alpha$. First, consider $a, b \notin A$. Clearly $(a\ b)$ fixes A, so $(a\ b) \cdot L^\alpha = L^\alpha$. Therefore we have:

$$\begin{aligned} \mathcal{F}_a(L) = \mathcal{F}_a(L^\alpha) = \mathcal{F}_a((a\ b) \cdot L^\alpha) &= \mathcal{F}_{(a\ b)^{-1}(a)}(L^\alpha) \\ &= \mathcal{F}_b(L^\alpha) = \mathcal{F}_b(L). \end{aligned}$$

This proves that the binding power associated with every name outside A are equal, so let us choose $a_0 \notin A$, we get that $\mathcal{F}L$ is equal to $\mathcal{F}_{a_0}(L) \cup \bigcup_{a \in A} \mathcal{F}_a(L)$. Since A is finite and every $\mathcal{F}_a(F)$ is finite so is $\mathcal{F}L$. $\square$

Lemma 5.11. ***bf*** $\wedge$ ***abs*** $\Leftrightarrow$ ***ubf***.

Proof. The right-to-left implication is a consequence of Lemmas 5.1, 5.2, 5.3 and 5.4. Assume $\mathbf{supp}_\alpha(L)$ is finite and L is **bf** . According to Lemma 5.10, $\mathcal{F}L$ is finite. This means that $\mathbb{F}L$ is composed of function that have values in the finite set $\mathbb{F}L \cup \{\varepsilon\}$, and such that $f(a) \neq \varepsilon$ entails $a \in \mathbf{supp}_\alpha(L)$. Therefore there are at most $(|\mathbb{F}L| + 1)^{|\mathbf{supp}_\alpha(L)|}$ of them. $\square$

5.3 Examples

Definition of the languages We assume we have a family of atoms $(\alpha_i)_{i\in\mathbb{N}}$ such that if $i \neq j$ then $\alpha_i \neq \alpha_j$. We define the set $X := \{\alpha_{2k} \mid k \in \mathbb{N}\}$. We now present a list of 24 languages which we will use to distinguish the above classes.

$$
\begin{aligned}
L_1 &:= \left\{ {\langle_{\alpha_i}}^{i} \;\middle|\; i \in \mathbb{N} \right\}.\\
L_2 &:= \left\{ \langle_{a_1} \langle_{a_2} \langle_{a_3} \cdots \langle_{a_n} \;\middle|\; n \in \mathbb{N} \wedge (i \neq j \Rightarrow a_i \neq a_j) \wedge (a_i \in X) \right\}.\\
L_3 &:= \left\{ \langle_{a_1} \langle_{a_2} \langle_{a_3} \cdots \langle_{a_n} \;\middle|\; n \in \mathbb{N} \wedge (i \neq j \Rightarrow a_i \neq a_j) \wedge (a_i \in \mathbb{A}) \right\}.\\
L_4 &:= \{ \langle_a \mid a \in X \}.\\
L_5 &:= \{ \langle_a \mid a \in \mathbb{A} \}.\\
L_6 &:= \{ {\langle_a}^{n} \mid a \in X \wedge n \in \mathbb{N} \}\,(= {L_4}^{\star}).\\
L_7 &:= \{ {\langle_a}^{n} \mid a \in \mathbb{A} \wedge n \in \mathbb{N} \}\,(= {L_5}^{\star}).\\
L_8 &:= \left\{ {\langle_{\alpha_0}}^{n} \;\middle|\; n \in \mathbb{N} \right\} \left(= {\langle_{\alpha_0}}^{\star}\right).\\
L_9 &:= \{ {\langle_a}^{n}\, {{}_a\rangle}^{n} \mid a \in X \wedge n \in \mathbb{N} \}.\\
L_{10} &:= \{ {\langle_a}^{n}\, {{}_a\rangle}^{n} \mid a \in \mathbb{A} \wedge n \in \mathbb{N} \}\,(= {L_9}^{\alpha} = {L_{11}}^{\alpha}).\\
L_{11} &:= \left\{ {\langle_{\alpha_0}}^{n}\, {{}_{\alpha_0}\rangle}^{n} \;\middle|\; n \in \mathbb{N} \right\}.\\
L_{12} &:= \{ \langle_a\ {}_a\rangle \mid a \in X \}.\\
L_{13} &:= \{ \langle_a\ {}_a\rangle \mid a \in \mathbb{A} \}\,(= {L_{12}}^{\alpha}).\\
L_{14} &:= \left\{ {\langle_a}^{n+1}\, {{}_a\rangle}^{n} \;\middle|\; a \in X \wedge n \in \mathbb{N} \right\}.\\
L_{15} &:= \{ \langle_a \langle_b\ {}_b\rangle \mid a \in \mathbb{A} \wedge b \in X \}\,(= L_5 \cdot L_{12}).\\
L_{16} &:= \{ \langle_a {\langle_b}^{n}\, {{}_b\rangle}^{n} \mid a \in \mathbb{A} \wedge b \in X \wedge n \in \mathbb{N} \}\,(= L_5 \cdot L_9).\\
L_{17} &:= \{ \langle_a {\langle_b}^{n}\, {{}_b\rangle}^{n} \mid a \in \mathbb{A} \wedge b \in \mathbb{A} \wedge n \in \mathbb{N} \}\,(= L_5 \cdot L_{10}).\\
L_{18} &:= \left\{ \left(\langle_{\alpha_0}\ {}_{\alpha_0}\rangle\right)^{n} \mid n \in \mathbb{N} \right\} \left(= \left(\langle_{\alpha_0}\ {}_{\alpha_0}\rangle\right)^{\star}\right).\\
L_{19} &:= \left\{ {\langle_{\alpha_0}}^{n} {\langle_{\alpha_1}}^{n} \;\middle|\; n \in \mathbb{N} \right\}.\\
L_{20} &:= \left\{ \left(\langle_{\alpha_0}\ {}_{\alpha_0}\rangle\right)^{n} \left(\langle_{\alpha_1}\ {}_{\alpha_1}\rangle\right)^{n} \;\middle|\; n \in \mathbb{N} \right\}.\\
L_{21} &:= L_3 \cup L_{12}. \quad L_8 \cup L_{13}.\\
L_{22} &:= L_8 \cup L_{13}.\\
L_{23} &:= L_8 \cup L_9.\\
L_{24} &:= L_8 \cup L_{16}.
\end{aligned}
$$

Properties of the example languages We now list a series of remarks, allowing us to precisely classify each of the example languages. These are then used in Table 1 to say which predicates are satisfied by which languages.

- $\mathbf{R}_0$ From Sections 5.2 and 4.2 we know the implications from Figure 2.
- $\mathbf{R}_1$ for $L \in \{L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8, L_{19}\}$, $\forall u \in L, \{u\}^{\alpha} = \{u\}$, therefore $L^{\alpha} = L$. This means that for these languages $\mathbf{fs} = \boldsymbol{\alpha}\mathbf{fs}$.
- $\mathbf{R}_2$ for $L \in \{L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8, L_{14}, L_{19}\}$, $\forall u \in L, \mathbf{supp}_{\alpha}(u) = \mathbf{supp}(u)$. This means that for these languages $\mathbf{bs} = \boldsymbol{\alpha}\mathbf{bs}$.
- $\mathbf{R}_3$ for $L \in \{L_3, L_5, L_7, L_{10}, L_{13}, L_{17}\}$, $\forall \pi \in \mathfrak{S}_{\mathbb{A}}, \forall u \in L$, we have $\pi \cdot u \in L$, so $L \in \mathbf{fs}$.
- $\mathbf{R}_4$ for $L \in \{L_9, L_{10}, L_{11}, L_{12}, L_{13}, L_{18}, L_{20}\}$, $\forall u \in L, \forall a \in \mathbb{A}$, we have $\mathcal{F}_a(u) = \varepsilon$, so $L \in \mathbf{ubf}$.
- $\mathbf{R}_5$ L_8 and L_{18} are regular, L_{11}, L_{19} and L_{20} are not.
- $\mathbf{R}_6$ for $L \in \{L_3, L_5, L_7, L_{10}, L_{13}, L_{15}, L_{16}, L_{17}\}$, we have that $\bigcup_{u\in L} \mathbf{supp}(u) = \mathbb{A}$, so $L \notin \mathbf{bs}$.
- $\mathbf{R}_7$ for $L \in \{L_1, L_2, L_4, L_6, L_9, L_{12}, L_{14}\}$, we have $X \subseteq \bigcup_{u\in L} \mathbf{supp}(u)$, so $L \notin \mathbf{bs}$.
- $\mathbf{R}_8$ for $L \in \{L_8, L_{11}, L_{18}, L_{19}, L_{20}\}$, we have have if $u \in L$ then $\mathbf{supp}(u) \subseteq \{\alpha_0, \alpha_1\}$, so $L \in \mathbf{bs}$.
- $\mathbf{R}_9$ for $L \in \{L_1, L_2, L_4, L_6, L_9, L_{12}, L_{14}, L_{15}, L_{16}, L_{21}, L_{23}, L_{24}\}$, for any pair $\langle a, b\rangle \in X \times (\mathbb{A} \setminus X)$, there exists a word $u \in L$ such that $(a\ b) \cdot u \notin L$. Therefore, for any finite set B, we can find a, b such that $(a\ b)$ fixes B but $(a\ b) \cdot L \neq L$. This implies that $L \notin \mathbf{fs}$.

$\mathbf{R}_{10}$ for $C \in \{\mathbf{ubf}, \mathbf{mf}, \mathbf{bw}, \mathbf{bf'}, \mathbf{bf}, \mathbf{bs}, \boldsymbol{\alpha}\mathbf{bs}, \mathbf{fs}, \boldsymbol{\alpha}\mathbf{fs}, \mathbf{reg}\}$, if two languages L, M belong to the class C, $L \cup M$ belongs to C.

$\mathbf{R}_{11}$ for $C \in \{\mathbf{ubf}, \mathbf{mf}, \mathbf{bw}, \mathbf{bf'}, \mathbf{bf}, \mathbf{bs}, \boldsymbol{\alpha}\mathbf{bs}\}$, if $L \cup M$ belongs to C, then both L and M belong to C.

$\mathbf{R}_{12}$ for $L \in \{L_4, L_5, L_{12}, L_{13}, L_{15}, L_{18}, L_{20}\}$, for any word $u \in \mathbf{prefix}(L)$ we have $\|u\| \leqslant 2$, so $L \in \mathbf{mf}$.

$\mathbf{R}_{13}$ for $L \in \{L_9, L_{10}, L_{11}, L_{14}, L_{16}, L_{17}\}$, for any word $u \in L$ we have $\|u\| \leqslant 1$, so $L \in \mathbf{bw}$. However, for any $n \in \mathbb{N}$, there is a word $u \in \mathbf{prefix}(L)$ such that $\|u\| = 1$, so $L \notin \mathbf{mf}$.

$\mathbf{R}_{14}$ for $L \in \{L_6, L_7, L_8, L_{19}\}$, there are atoms $a \in \mathbb{A}$ such that $\mathcal{F}_a(L) = \mathbf{c}^\star$, so $L \notin \mathbf{bf}$.

$\mathbf{R}_{15}$ for $L \in \{L_{15}, L_{16}, L_{17}\}$, for $a \in \mathbb{A}$, there is $u \in L$ such that $\mathcal{F}_a(u) = \mathbf{c}$, so $L \notin \boldsymbol{\alpha}\mathbf{bs}$. However, for any $\pi \in \mathfrak{S}_{\mathbb{A}}$, $\pi \cdot u =_\alpha u$, so $\pi \cdot u \in L^\alpha$, hence the empty set supports L^α, meaning that $L \in \boldsymbol{\alpha}\mathbf{fs}$.

$\mathbf{R}_{16}$ since $L_{17} \in \mathbf{fs}$ and ${L_{16}}^\alpha = L_{17}$, $L_{16} \in \boldsymbol{\alpha}\mathbf{fs}$.

$\mathbf{R}_{17}$ since $\mathcal{F}_{\alpha_i}(L_1) = \{\varepsilon, \mathbf{c}^i\}$ and for every a outside the set $\{\alpha_i \mid i \in \mathbb{N}\}$ we have $\mathcal{F}_a(L_1) = \{\varepsilon\}$, we know that $L_1 \in \mathbf{bf}$ however, $\mathcal{F}L_1 = \mathbf{c}^\star$, so $L_1 \notin \mathbf{bf'}$.

$\mathbf{R}_{18}$ for $L \in \{L_1, L_2\}$, since in every word each atom appears at most once, $\mathcal{F}L = \{\mathbf{c}, \varepsilon\}$, so $L \in \mathbf{bf'}$. However, for any n the word $\langle_{\alpha_0} \langle_{\alpha_2} \ldots \langle_{\alpha_{2n}}$ is in L and has weight $n+1$, so $L \notin \mathbf{bw}$.

$\mathbf{R}_{19}$ For any pair of names $\langle a, b\rangle \in X \times (\mathbb{A} \setminus X)$, the word $\langle_a \in {L_{14}}^\alpha$ but $\langle_b \notin {L_{14}}^\alpha$. Therefore, for any finite set of atoms A, pick $b \in \mathbb{A} \setminus (X \cup A)$ and $a \in X \setminus A$. The permutation $(a\ b)$ fixes A but does not leave ${L_{14}}^\alpha$ unchanged: $L_{14} \notin \boldsymbol{\alpha}\mathbf{fs}$.

	L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}	L_{11}	L_{12}
ubf	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_4$	✓ $\mathbf{R}_4$	✓ $\mathbf{R}_4$	✓ $\mathbf{R}_4$
mf	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_{12}$	✓ $\mathbf{R}_{12}$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_{13}$	✗ $\mathbf{R}_{13}$	✗ $\mathbf{R}_{13}$	✓ $\mathbf{R}_{12}$
bw	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_{18}$	✗ $\mathbf{R}_{18}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_{13}$	✓ $\mathbf{R}_0$
bf'	✗ $\mathbf{R}_{17}$	✓ $\mathbf{R}_{18}$	✓ $\mathbf{R}_{18}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$
bf	✓ $\mathbf{R}_{17}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_{14}$	✗ $\mathbf{R}_{14}$	✗ $\mathbf{R}_{14}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$
bs	✗ $\mathbf{R}_7$	✗ $\mathbf{R}_7$	✗ $\mathbf{R}_6$	✗ $\mathbf{R}_7$	✗ $\mathbf{R}_6$	✗ $\mathbf{R}_7$	✗ $\mathbf{R}_6$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_7$	✗ $\mathbf{R}_6$	✓ $\mathbf{R}_8$	✗ $\mathbf{R}_7$
αbs	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_2$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_2$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_2$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$
fs	✗ $\mathbf{R}_9$	✗ $\mathbf{R}_9$	✓ $\mathbf{R}_3$	✗ $\mathbf{R}_9$	✓ $\mathbf{R}_3$	✗ $\mathbf{R}_9$	✓ $\mathbf{R}_3$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_9$	✓ $\mathbf{R}_3$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_9$
αfs	✗ $\mathbf{R}_1$	✗ $\mathbf{R}_1$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_1$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_1$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$
reg	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_5$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$

	L_{13}	L_{14}	L_{15}	L_{16}	L_{17}	L_{18}	L_{19}	L_{20}	L_{21}	L_{22}	L_{23}	L_{24}
ubf	✓ $\mathbf{R}_4$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_4$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_4$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$
mf	✓ $\mathbf{R}_{12}$	✗ $\mathbf{R}_{13}$	✓ $\mathbf{R}_{12}$	✗ $\mathbf{R}_{13}$	✗ $\mathbf{R}_{13}$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_{12}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$
bw	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_{13}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_{13}$	✓ $\mathbf{R}_{13}$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$
bf'	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_{10}$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_{11}$
bf	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_{14}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_{11}$
bs	✗ $\mathbf{R}_6$	✗ $\mathbf{R}_7$	✗ $\mathbf{R}_6$	✗ $\mathbf{R}_6$	✗ $\mathbf{R}_6$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_8$	✓ $\mathbf{R}_8$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$	✗ $\mathbf{R}_{11}$
αbs	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_2$	✗ $\mathbf{R}_{15}$	✗ $\mathbf{R}_{15}$	✗ $\mathbf{R}_{15}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_{11}$	✓ $\mathbf{R}_{10}$	✓ $\mathbf{R}_{10}$	✗ $\mathbf{R}_{11}$
fs	✓ $\mathbf{R}_3$	✗ $\mathbf{R}_9$	✗ $\mathbf{R}_9$	✗ $\mathbf{R}_9$	✓ $\mathbf{R}_3$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_9$	✓ $\mathbf{R}_{10}$	✗ $\mathbf{R}_9$	✗ $\mathbf{R}_9$
αfs	✓ $\mathbf{R}_0$	✗ $\mathbf{R}_{19}$	✓ $\mathbf{R}_{15}$	✓ $\mathbf{R}_{16}$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_0$	✓ $\mathbf{R}_{10}$	✓ $\mathbf{R}_{10}$	✓ $\mathbf{R}_{10}$	✓ $\mathbf{R}_{10}$
reg	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✓ $\mathbf{R}_5$	✗ $\mathbf{R}_5$	✗ $\mathbf{R}_5$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$	✗ $\mathbf{R}_0$

Table 1: Classification of the example languages

This table shows how different each class is and the example languages pinpoint exactly that the inclusions between the classes are strict. We depict the inclusions together with the languages that witness their difference in Figure 3.

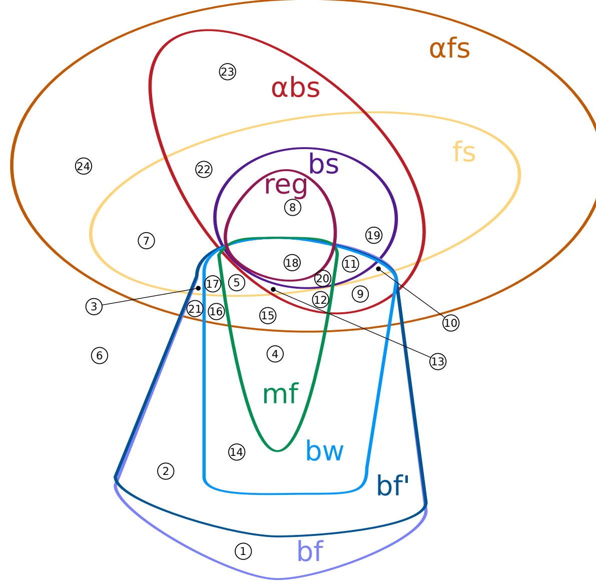


Fig. 3: Graphical representation of the tractability conditions.

6 Automata for equivalence checking

In this section, we study the following decision problems: given two regular expressions $e, f \in \mathrm{Reg}\langle\Sigma\rangle$, is it the case that $[\![e]\!]^\alpha = [\![f]\!]^\alpha$? Is it the case that $[\![e]\!]^\alpha \subseteq [\![f]\!]^\alpha$? To do so we will try and reduce these problems to comparing finite state automata. However, we do not know how to solve these in general, and only answer those questions when either e or f is binding finite.

Consider a finite state automaton $\mathscr{A} = \langle Q, \Sigma_\mathscr{A}, \rightarrow_\mathscr{A}, I, F\rangle$, where $\Sigma_\mathscr{A}$ is a finite subset of Σ. We write $[\![\mathscr{A}]\!]$ for the regular language accepted by $\mathscr{A}$. We will combine $\mathscr{A}$ with $\mathcal{T}$ to define a labelled transition system $\mathscr{A}^\alpha$ recognising the language $[\![\mathscr{A}]\!]^\alpha$. The states of $\mathscr{A}^\alpha$, called configurations in the following, are pairs $\langle q, s\rangle$, where $q \in Q$ and s is a stack. We now describe how to update a configuration by reading a letter:

$$\frac{\exists l' \in \Sigma_\mathscr{A} : q \xrightarrow{l'}_\mathscr{A} q' \quad s -[l'/l]\rightarrow_\mathcal{T} s'}{\langle q, s\rangle \xrightarrow{l}_{\mathscr{A}^\alpha} \langle q', s'\rangle}$$

The set of initial configurations is $I^\alpha := I \times \{\bot\}$, the set final configurations is $F^\alpha := F \times \mathbb{S}^{acc}$. The language of $\mathscr{A}^\alpha$ is the set of words $w \in \Sigma^\star$ such that there is path $c_i \xrightarrow{w}_{\mathscr{A}^\alpha} c_f$ between an initial configuration c_i and a final configuration c_f. In this sense, $\mathscr{A}^\alpha$ may be seen as an automaton over an infinite alphabet and with an infinite state space: $\mathscr{A}^\alpha = \left\langle Q \times \mathbb{S}, \Sigma, \rightarrow_{\mathscr{A}^\alpha}, I^\alpha, F^\alpha\right\rangle$.

Lemma 6.1. *The language of $\mathscr{A}^\alpha$ is the α-closure of $[\![\mathscr{A}]\!]$.*

Proof. From the definition of $\mathscr{A}^\alpha$ we can show that $\langle p, s\rangle \xrightarrow{w}_{\mathscr{A}^\alpha} \langle q, s'\rangle$ if and only if there exists a word v such that $p \xrightarrow{v}_\mathscr{A} q$ and $s -[v/w]\rightarrow_\mathcal{T} s'$, hence we can conclude using Theorem 3.1.

$$\begin{aligned} w \in [\![\mathscr{A}^\alpha]\!] &\Leftrightarrow \exists \langle p, q, s\rangle \in I \times F \times \mathbb{S}^{acc} : \langle p, \bot\rangle \xrightarrow{w}_{\mathscr{A}^\alpha} \langle q, s\rangle \\ &\Leftrightarrow \exists \langle p, q, s, v\rangle \in I \times F \times \mathbb{S}^{acc} \times \Sigma_\mathscr{A}^\star : p \xrightarrow{v}_\mathscr{A} q \wedge \bot -[v/w]\rightarrow_\mathcal{T} s \\ &\Leftrightarrow \exists v \in \Sigma_\mathscr{A}^\star : v \in [\![\mathscr{A}]\!] \wedge v =_\alpha w \Leftrightarrow w \in [\![\mathscr{A}]\!]^\alpha . \end{aligned}$$

□

Therefore, this transition system may be used to decide the membership problem: given u and $\mathscr{A}$, is it the case that $u \in [\![\mathscr{A}]\!]^{\alpha}$? We do not develop this further, as this result will be subsumed by Theorem 6.4.

Although this automaton is always infinite, there are cases where language equivalence/containment of these automata is decidable. In the following, we will only consider co-accessible automata, that is to say that for any state $p \in Q$ there exists a pair $\langle u, q \rangle \in \Sigma_{\mathscr{A}}^{\star} \times F$ such that $p \xrightarrow{u}_{\mathscr{A}} q$. This condition is easy to enforce: most algorithms producing automata from regular expressions either produce co-accessible automata or only need simple modifications to do so.

We now show the main technical lemma of this section:

Lemma 6.2. *Given a co-accessible automaton $\mathscr{A}$, an alphabet $\Sigma \in \mathcal{P}_f(\mathbb{\Sigma})$ and a bound $N \in \mathbb{N}$, there exists a finite state automaton $\mathscr{A}\lfloor_{\Sigma,N}$ such that $[\![\mathscr{A}\lfloor_{\Sigma,N}]\!] = [\![\mathscr{A}]\!]^{\alpha} \cap \mathcal{C}_{\Sigma}^{\leqslant N}$.*

Proof. We write $A = \mathbf{supp}(\mathscr{A}) = \bigcup_{l \in \Sigma_{\mathscr{A}}} \mathbf{supp}(l)$ and $B = \mathbf{supp}(\Sigma)$. Since $\Sigma_{\mathscr{A}}$ and Σ are finite, both A and B are finite sets. The automaton $\mathscr{A}\lfloor_{\Sigma,N}$ is simply built as

$$\mathscr{A}\lfloor_{\Sigma,N} = \left\langle Q', \Sigma, \rightarrow_{\mathscr{A}^{\alpha}}, I^{\alpha}, F^{\alpha} \cap Q' \right\rangle,$$

where $Q' = Q \times (A \times B)^{\leqslant N}$. For simplicity, we will denote $\mathscr{A}'$ for $\mathscr{A}\lfloor_{\Sigma,N}$ in the rest of the proof.

Let us check that this language accepted by this automaton is indeed $[\![\mathscr{A}]\!]^{\alpha} \cap \mathcal{C}_{\Sigma}^{\leqslant N}$.

1. Assume $w \in [\![\mathscr{A}']\!]$. Clearly $w \in [\![\mathscr{A}^{\alpha}]\!]$, since the states and transitions of $\mathscr{A}'$ are a subset of those of $\mathscr{A}^{\alpha}$. Since we restricted the alphabet to Σ, we also know that $w \in \Sigma^{\star}$. Therefore, we only need to check that $\forall u \in \mathbf{prefix}(w)$, $\sum_a \mathcal{c}_a(u) \leqslant N$. Let u be a prefix of w. Since $w \in [\![\mathscr{A}']\!]$, there is a run in $\mathscr{A}'$ labelled with w from some initial state $\langle q_0, [] \rangle$. We may extract the prefix of that run that corresponds to u: $\langle q_0, [] \rangle \xrightarrow{u}_{\mathscr{A}'} \langle q, s \rangle$. From the definition of $\mathscr{A}'$, we get the following facts: $\langle q_0, [] \rangle \xrightarrow{u}_{\mathscr{A}^{\alpha}} \langle q, s \rangle$ and $|s| \leqslant N$. From the definition of $\mathscr{A}^{\alpha}$, we know that there exists a word v such that $[] -[v/u]\rightarrow_{\mathcal{T}} s$. We conclude using Lemma 3.3:
$$\sum_a \mathcal{c}_a(u) = \sum_a \left(|\mathfrak{p}_2([])|_a \dot{-} \mathcal{d}_a(u) \right) + \mathcal{c}_a(u) = \sum_a |\mathfrak{p}_2(s)|_a = |s| \leqslant N.$$
2. Now, suppose $w \in [\![\mathscr{A}]\!]^{\alpha} \cap \mathcal{C}_{\Sigma}^{\leqslant N}$. We know that there is a path $\langle q_0, [] \rangle \xrightarrow{w}_{\mathscr{A}^{\alpha}} \langle q_f, s_f \rangle$ with $q_0 \in I$, $q_f \in F$ and $s_f \in \mathbb{S}^{acc}$. To show that $w \in [\![\mathscr{A}']\!]$, we need to be able to reproduce this run in $\mathscr{A}'$, so we want to check that every state visited along this path belongs to Q', i.e. every stack visited belongs to the set $(A \times B)^{\leqslant N}$. Let us inspect some intermediary state $\langle q, s \rangle$: w can be splitted as uv such that $\langle q_0, [] \rangle \xrightarrow{u}_{\mathscr{A}^{\alpha}} \langle q, s \rangle \xrightarrow{v}_{\mathscr{A}^{\alpha}} \langle q_f, s_f \rangle$. By construction of $\mathscr{A}^{\alpha}$, there must be some word u' such that $q_0 \xrightarrow{u'}_{\mathscr{A}} q$ and $[] -[u'/u]\rightarrow_{\mathcal{T}} s$. Since $w \in \mathcal{C}_{\Sigma}^{\leqslant N}$ and u is a prefix of w, we have $\sum_a \mathcal{c}_a(u) \leqslant N$. Therefore using the same argument as in the previous case, we immediately obtain that:
$$|s| = \sum_a \mathcal{c}_a(u) \leqslant N.$$
Furthermore, since $q_0 \xrightarrow{u'}_{\mathscr{A}} q$, we know that $u' \in \Sigma_{\mathscr{A}}^{\star}$, so $\mathbf{supp}(u) \subseteq A$, meaning that $\mathfrak{p}_1(s) \subseteq \mathbf{supp}(u) \subseteq A$. Similarly, since $u \in \mathcal{C}_{\Sigma}^{\leqslant N}$, we have $\mathfrak{p}_2(s) \subseteq \mathbf{supp}(u) \subseteq B$. This ensures that $s \in (A \times B)^{\leqslant N}$. □

6.1 Containment

We now focus on checking whether $[\![e]\!]^\alpha \subseteq [\![f]\!]^\alpha$ when either e or f is memory-finite.

We will rely on the following lemma:

Lemma 6.3. *Let $L, M, K \in \Sigma^\star$ such that $L \subseteq K$, then: $L^\alpha \subseteq M^\alpha \Leftrightarrow L \subseteq M^\alpha \cap K$.*

Proof. For the left-to-right direction, assume $L^\alpha \subseteq M^\alpha$. Since $L \subseteq L^\alpha$, we have $L \subseteq M^\alpha$. We conclude by monotonicity of $\cap$: $L = L \cap K \subseteq M^\alpha \cap K$.

For the converse direction, assume $L \subseteq M^\alpha \cap K$. Since $M^\alpha \cap K \subseteq M^\alpha$ we get $L \subseteq M^\alpha$, hence by monotonicity of the α-closure operator we get $L^\alpha \subseteq M^{\alpha\alpha}$. We now conclude, since idempotency of $-^\alpha$ ensures that $M^{\alpha\alpha} = M^\alpha$. ◻

We may now prove the following statement:

Theorem 6.4. *If either e or f is memory-finite, then $[\![e]\!]^\alpha \subseteq [\![f]\!]^\alpha$ is decidable.*

Proof. We may easily dismiss the case where f is memory-finite but e is not. Indeed in this case is not possible that $[\![e]\!]^\alpha \subseteq [\![f]\!]^\alpha$, as this would entail $\forall a \in \mathbb{A}, \mathcal{F}_a(e) \subseteq \mathcal{F}_a(f)$. Using **isBF** (e) and Lemma 4.7 we can easily detect this situation and provide the correct (negative) answer.

We now assume that e is memory-finite. Let Σ be the alphabet of e, i.e. the set of letters that appear in e. Thanks to Lemma 4.5, we know that $[\![e]\!] \subseteq \mathcal{M}_\Sigma^{\leqslant 2|e|} \subseteq \mathcal{C}_\Sigma^{\leqslant 2|e|}$. Using Lemma 6.2, we build the automaton $\mathscr{A}_f \downharpoonleft_{\Sigma,2|e|}$. We may now test the automata $\mathscr{A}_e$ and $\mathscr{A}_f \downharpoonleft_{\Sigma,2|e|}$ for language inclusion, which will lead to the correct answer thanks to Lemma 6.3:

$$\begin{aligned} [\![e]\!]^\alpha \subseteq [\![f]\!]^\alpha &\Leftrightarrow [\![e]\!] \subseteq [\![f]\!]^\alpha \cap \mathcal{C}_\Sigma^{\leqslant 2|e|} && \text{(Lemma 6.3)} \\ &\Leftrightarrow [\![\mathscr{A}_e]\!] \subseteq [\![\mathscr{A}_f \downharpoonleft_{\Sigma,2|e|}]\!] && \text{(Lemma 6.2)} \end{aligned}$$

◻

6.2 Equality

Since we have shown containment to be decidable, we automatically get a decision procedure for equality of memory-finite regular languages up-to α-equivalence, by simply proceeding by double-inclusion. There is however a more direct way, thanks to the following observation:

Lemma 6.5. *Let $L, M, K \subseteq \Sigma^\star$ such that $L, M \subseteq K$. Then $L^\alpha = M^\alpha$ if and only if $L^\alpha \cap K = M^\alpha \cap K$.*

Proof. The left-to-right implication being trivial, we focus on the converse. Assume $L^\alpha \cap K = M^\alpha \cap K$. Since $L \subseteq L^\alpha$ and $L \subseteq K$, we have $L \subseteq L^\alpha \cap K$. By hypothesis this means $L \subseteq M^\alpha \cap K$. Since $L \subseteq K$, Lemma 6.3 applies, so we get $L^\alpha \subseteq M^\alpha$. A symmetric argument shows the other containment. ◻

We may now prove the following statement:

Theorem 6.6. *If either e or f is memory-finite, then $[\![e]\!]^\alpha = [\![f]\!]^\alpha$ is decidable.*

Proof. In this case we may assume that both e and f are memory finite: if this is not the case (which we can detect using **isBF** ()) then the equality does not hold.

Let Σ be the alphabet of $e \cup f$, i.e. the set of letters that appear in either e or f, and $N = 2 \times \max(|e|, |f|)$. Thanks to Lemma 4.5, we know that $[\![e]\!], [\![f]\!] \subseteq \mathcal{M}_\Sigma^{\leqslant N} \subseteq \mathcal{C}_\Sigma^{\leqslant N}$. We now conclude using Lemmas 6.5 and 6.2:

$$\begin{aligned} [\![e]\!]^\alpha = [\![f]\!]^\alpha &\Leftrightarrow [\![e]\!]^\alpha \cap \mathcal{C}_\Sigma^{\leqslant N} = [\![f]\!]^\alpha \cap \mathcal{C}_\Sigma^{\leqslant N} && \text{(Lemma 6.3)} \\ &\Leftrightarrow [\![\mathscr{A}_e \downharpoonleft_{\Sigma,N}]\!] = [\![\mathscr{A}_f \downharpoonleft_{\Sigma,N}]\!] && \text{(Lemma 6.2)} \end{aligned}$$

◻

Remark 6.7. In a distinct paper [1] we showed that for any memory-finite expression e the language $[\![e]\!]^\alpha$ can be recognised by a deterministic nominal automaton. Since equivalence of these automata is decidable [3,5], this yields another proof of decidability. However since the reduction from expressions to deterministic automata is slightly involved, we expect that the approach presented in this section yields more efficient algorithms.

7 Discussion and further work

We have presented *bracket algebra*, a new algebraic framework to reason about programs with interleaved scopes. We built the framework modularly: first to reason on words (or program traces) to then reason on languages (that is, sets of traces). We presented a novel transducer model to capture equivalence of traces and showed it provides us with a sound and complete method for deciding equivalence. For languages, we developed a systematic study of their expressive power classified in terms of different properties related to their nominal characteristics and to their binding and memory power. The hierarchy depicted in Figure 3 is to the best of our knowledge the first language hierarchy for nominal languages. We have also provided an automaton model that enabled us to devise decision procedures for containment – $[\![e]\!]^\alpha \subseteq [\![f]\!]^\alpha$ – and equality – $[\![e]\!]^\alpha = [\![f]\!]^\alpha$ – of a sub-class of languages, namely memory-finite languages. Interestingly, only one of the expressions e or f is required to be memory-finite.

The results in sections 3, 4 and 6 have been fully proved in Coq, as part of an ongoing study of bracket algebra. The library is available on GitHub[5], and currently has over 30k lines.

We see several directions for future work. From a theoretical perspective, there are two obvious directions. First, we would like to investigate the companion hierarchy of automata models to the hierarchy of languages in Figure 3 and explore whether we can provide further decidability or hardness results for containment and equivalence. Second, we would like to be able to have a sound and complete axiomatizations to reason about the various classes of languages – it is not clear which sub-classes we will be able to axiomatize. From a more applied side, we would like to explore applications to software verification, by leveraging the ability of our framework to use **any** nominal set as alphabet (note we do not require orbit-finiteness, as our expressions can only mention a finite number of those).

References

1. Brunet, P., Silva, A.: A Kleene theorem for nominal automata (2018), draft available at `http://paul.brunet-zamansky.fr/Brackets/kleene.pdf`
2. Gabbay, J., Ghica, D.R., Petrişan, D.: Leaving the Nest: Nominal Techniques for Variables with Interleaving Scopes. In: CSL. vol. 41 (2015). https://doi.org/10.4230/LIPIcs.CSL.2015.374
3. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science **134**(2), 329 – 363 (1994). https://doi.org/10.1016/0304-3975(94)90242-9
4. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991. pp. 214–225. IEEE Computer Society (1991). https://doi.org/10.1109/LICS.1991.151646, `https://doi.org/10.1109/LICS.1991.151646`
5. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Polynomial-Time Equivalence Testing for Deterministic Fresh-Register Automata. In: MFCS (2018). https://doi.org/10.4230/LIPIcs.MFCS.2018.72
6. Pitts, A.M.: Nominal Sets: Names and Symmetry in Computer Science. Cambridge University Press, New York, NY, USA (2013)
7. Strichman, O.: Special issue: program equivalence. Formal Methods in System Design **52**(3), 227–228 (2018). https://doi.org/10.1007/s10703-018-0318-y, `https://doi.org/10.1007/s10703-018-0318-y`

[5] `https://github.com/monstrencage/BracketAlgebra`