Innovations in Systems and Software Engineering (2019) 15:253–265
https://doi.org/10.1007/s11334-019-00347-1

S.I.: NFM2018

Certified normalization of generalized traces

Hendrik Maarand[2] · **Tarmo Uustalu**[1,2]

Received: 1 October 2018 / Accepted: 21 May 2019 / Published online: 18 June 2019

Abstract
Mazurkiewicz traces are a generalization of strings where an independence relation on the alphabet for commutability of letters induces an equivalence relation on strings. The equivalence relation can be made more expressive by allowing the commutability of two adjacent letters in a string to depend on their left context. We generalize two classical normal forms and the corresponding normalization algorithms for Mazurkiewicz traces for Sassone et al.'s context-dependent generalization of traces, formalize this development in the dependently typed programming language Agda, and show generalized traces in action on an example from relaxed shared-memory concurrency (local reads in TSO).

Keywords Concurrency · Mazurkiewicz traces · Normal forms · Relaxed memory

1 Introduction

A string over an alphabet can be used to represent a sequence of observations that we have made or a sequence of actions that we have carried out. It may be that, for some letters in this string, there is no semantic difference between one letter appearing immediately before the other and the other way around. This idea is made precise in Mazurkiewicz traces [16] using the notion of independent letters.

A Mazurkiewicz trace corresponds to a set of strings that can be considered equivalent—it is an equivalence class of strings. Two strings are taken to be equivalent if they can be transformed into each other by a finite number of commutations of adjacent letters. Commutation is only allowed for letters in a given irreflexive and symmetric binary relation, called the independence relation. Two letters that are not independent are called dependent. The ordering of occurrences of dependent letters is fixed in an equivalence class.

While traces are sets of strings, it may be desirable in practice to work with particular single strings representing these sets canonically. Ideally, these representatives should be defined by a decidable predicate. In every trace, there should be exactly one such representative, in which case these representatives can be called normal forms. Given a string, it should be possible to compute its normal form, i.e. the normal form in its equivalence class. One such normal form is the Foata normal form, corresponding to a maximally parallel representation of a trace. Another is the lexicographic normal form, which is the minimal string in its equivalence class according to the lexicographic ordering induced by a strict total order on the alphabet. Yet another is dependence graphs. The possibility to work only with representatives is important in practice. It is often referred to as partial-order reduction.

In some applications, it may be necessary to depart from standard trace theory by making commutability of two adjacent letters in a string depend on their position in it, specifically their left context. The idea is that this context or history can be seen as a kind of state, affecting commutability. For this generalization, the independence relation is made dependent on a string parameter for the context. To behave reasonably, it has to meet some well-behavedness conditions. Above all, it must be consistent, i.e. stable under equivalence of contexts, but usually more is required. The exact necessary conditions depend on the application at hand; different sets of conditions have been considered by different authors.

It is natural to ask whether the Foata and lexicographic normalization of Mazurkiewicz traces can work also for generalized traces. In this paper, we study and answer this question. On the first look, the prospects for a positive answer are unclear, as the situation is subtler than for standard traces.

✉ Hendrik Maarand
hendrik@cs.ioc.ee

Tarmo Uustalu
tarmo@ru.is

[1] School of Computer Science, Reykjavik University, Menntavegi 1, 101 Reykjavík, Iceland

[2] Department of Software Science, Tallinn University of Technology, Akadeemia tee 21B, 12618 Tallinn, Estonia

It is not immediate that the concept of a Foata normal form is reasonable at all—the order of letters in a step should not matter, but their contexts depend on it—or that the normalization function can be defined as in the standard case with one traversal of the given string—a priori, independence or dependence between letters in a string might not remain invariant under inserting an additional letter into the context. But, as it turns out, everything works out well, if one assumes the consistency and coherence conditions introduced by Sassone et al. [20]. Still, the definitions and proofs require considerably more care than in the standard case. Especially the proofs become quite subtle, as it is easy to make mistakes when the independence of letters may be altered when manipulating the context. This makes context-dependent Foata and lexicographic normalization a good exercise in certified programs and proofs. We conducted this exercise in the dependently typed programming language Agda [17].

The contribution of this paper thus consists in developing the theory of generalized Foata and lexicographic normalization and in formalizing it. Furthermore, we also demonstrate the usefulness of this generalization on an example from relaxed shared-memory concurrency (local reads in the TSO model). The reporting is organized as follows. We first introduce standard Mazurkiewicz traces informally in Sect. 2 and then move on to generalized traces in Sect. 3, where our formal descriptions begin. We then continue with generalized Foata and lexicographic normalization in Sects. 4 and 5. In Sect. 6, we demonstrate how context-dependent independence arises in relaxed shared-memory concurrency. Then we briefly discuss related work and conclude.

Our Agda formalization is available from http://cs.ioc.ee/~hendrik/code/generalized-traces/isse.zip. It has been developed with Agda version 2.5.4 and Agda standard library version 0.16. It consists of approximately 3500 lines of source code, roughly half of which is about traces and normalization, and the other half consists of the example from Sect. 6 and utility code for cons-/snoc-list manipulation and similar purposes.

We assume no knowledge of trace theory from the reader and introduce all relevant concepts and facts. When describing our theory development, we try to give a relatively high level view, yet be close to the formalization in Agda. We use syntax that is close to Agda's and explain it minimally as we go, primarily just through commenting what we are doing. Compared to the actual Agda code, in this pseudo-code, we often leave out implicit arguments of functions (in their types and defining equations), refrain from showing the symbol _ in binary operation identifiers, etc., for improved readability. The electronic version of this article links the descriptions of the theory development to the corresponding pieces in the HTML listing of the Agda source.

This is a substantially revised and extended version of the paper [15]. The main addition is the treatment of lexicographic normalization of generalized traces in addition to Foata normalization. The concurrency alphabet used in the example in Sect. 6 has also been fully formalized.

2 Mazurkiewicz traces

An *alphabet* A is a (non-empty) set whose elements are called letters. A *string* is a list of letters, i.e. an element of A^*, the free monoid on A. An *independence relation* $\mathsf{I} \subseteq \mathsf{A} \times \mathsf{A}$ is an irreflexive and symmetric binary relation. Its complement $\mathsf{D} = (\mathsf{A} \times \mathsf{A}) \setminus \mathsf{I}$, which is reflexive and symmetric, is called the *dependence relation*. Intuitively, if $\mathsf{a}\ \mathsf{I}\ \mathsf{b}$, then the strings uabv and ubav are considered the same. We define $\sim \subseteq \mathsf{A}^* \times \mathsf{A}^*$ to be the least relation such that $\mathsf{a}\ \mathsf{I}\ \mathsf{b}$ implies $\mathsf{uabv} \sim \mathsf{ubav}$ and define *(Mazurkiewicz) equivalence* $\sim^*$ to be its reflexive–transitive closure. A *(Mazurkiewicz) trace* is an equivalence class of strings wrt. $\sim^*$, i.e. an element of the quotient set $\mathsf{A}^*/\sim^*$, which is the free partially commutative monoid.

For example, if $\mathsf{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and I is the least symmetric relation satisfying $\mathsf{a}\ \mathsf{I}\ \mathsf{b}$, $\mathsf{a}\ \mathsf{I}\ \mathsf{d}$, $\mathsf{b}\ \mathsf{I}\ \mathsf{d}$, $\mathsf{c}\ \mathsf{I}\ \mathsf{d}$, then the strings abcd and bdac are equivalent, since $\mathsf{abcd} \sim \mathsf{bacd} \sim \mathsf{badc} \sim \mathsf{bdac}$, but acbd is not equivalent to them. The strings abcd, abdc, adbc, bacd, badc, bdac, dabc, dbac form one equivalence class of strings or trace. Another is $\{\mathsf{acbd}, \mathsf{acdb}, \mathsf{adcb}, \mathsf{dacb}\}$. Altogether, there are four traces containing each letter from A exactly once.

The definitions of Foata and lexicographic normal forms assume that we have a strict total order (i.e. a transitive and asymmetric relation) $\prec$ on the alphabet. We take the strict total order on the alphabet A to be $\mathsf{a} \prec \mathsf{b} \prec \mathsf{c} \prec \mathsf{d}$.

The Foata normal form is a sequence of steps where a step is a $\prec$-sorted string. All letters in a step must be pairwise independent, and every letter in a step must have a dependent letter in the previous step (the first step is excepted). This leads to the maximally parallel representation of the trace, with every letter appearing in the earliest possible step.

The Foata normal form of abcd in our example is $(\mathsf{abd})(\mathsf{c})$. It is a normal form since the letters are pairwise independent in both of the steps and c has a dependent letter in the previous step. It is the normal form of abcd since, if we turn the normal form $(\mathsf{abd})(\mathsf{c})$ into a string by flattening the steps, we get abdc, which is equivalent to abcd. Similarly, the normal form of acbd is $(\mathsf{ad})(\mathsf{c})(\mathsf{b})$.

A string is said to be in lexicographic normal form if it is the least element in its equivalence class according to the lexicographic ordering induced by $\prec$. An equivalent characterization in terms of a forbidden pattern was given by Anisimov and Knuth [3]: a string s is in lexicographic normal form if and only if, for every factorization tbuav of s

where b I a and a $\prec$ b, there is a letter d in u such that d D a (in other words, we should not be able to commute a to the left past b).

The string abcd is the lexicographic normal form in its equivalence class. Similarly, the string acbd is the lexicographic normal form in its equivalence class. The only potentially forbidden pattern in this string could be formed by c and b as they are in the wrong order, but the two letters are dependent and hence there are no forbidden patterns. The equivalent string acdb has a forbidden pattern: the letters d and b are in the wrong order, independent, and the substring between them (the empty string) does not contain a dependent letter.

3 Generalized Mazurkiewicz traces

There are several generalizations of trace theory. We are considering the generalization given by Sassone et al. [20]. In this setting, the essential difference is that independence is no longer a binary relation but an assignment of an irreflexive and symmetric independence relation to every string u. More precisely, we assume that we have an alphabet A and a context-dependent independence relation

I : A → List> A → A → Set

The second parameter to I is for the context. We use cons-lists over A (elements of List A) to represent strings (String = List A) and snoc-lists (List> A) to represent contexts of strings and also (steps of) normal forms. In the formal development, A and I together with their properties are module parameters.

We use both cons- and snoc-lists in the development, as this allows function definition by structural recursion and proof by structural induction from the correct end of the list which can be on the left or on the right depending on what is being done in a given situation. Typically, we want to work somewhere in the middle of a cons-list. We then split it into two parts, the left half (prefix) being a snoc-list and the right half (suffix) being a cons-list. Such pairs of snoc- and cons-lists are zippers for the cons-list type.

We seek to follow the following lexical convention where reasonable:

- a, b, c and d are letters
- s, t, u and v are snoc- or cons-lists
- ss and tt are snoc-lists of snoc-lists.

We now define when two strings differ only by the ordering of two adjacent independent letters.

Definition 1 $\sim$: List A → List> A → List A → Set

$$\frac{a\ I_{u+>s}\ b}{s >+< a <: b <: t\ \sim_u\ s >+< b <: a <: t}\ \text{swap}$$

This says that the strings sabt and sbat are equivalent in the context u when the letters a and b are independent in the context u +> s. The context is represented as a snoc-list as we usually need to access it from the right while strings are represented as cons-lists as we usually need to access them from the left. We use <: for cons, <+ for cons-append, :> for snoc and +> for snoc-append. We also use a mixed append operation >+< that takes a snoc- and a cons-list and produces a cons-list. When we need to translate between the two representations, we use s2c for snoc-to-cons and c2s for cons-to-snoc translation.

Mazurkiewicz equivalence is the reflexive-transitive closure of the above relation.

Definition 2 $\sim^*$: List A → List> A → List A → Set

$$\frac{s = t}{s \sim^*_u t}\ \text{refl*} \qquad \frac{s \sim_u v \quad v \sim^*_u t}{s \sim^*_u t}\ \text{swap-trans*}$$

A proof of $s \sim^*_u t$ can be thought of as a sequence of instructions for transforming s into t by swapping adjacent independent letters. No letters from u can be involved in these swaps. When the context u is empty, then we may omit the index.

In generalized traces the family of independence relations is required to be *consistent*, i.e. stable under equivalence:

I-cons : u $\sim^*$ v → a I_u b → a I_v b

It is also required to be *coherent*:

I-co1 : a I_u b → b I_{ua} c → a I_{ub} c → a I_u c

I-co2-e : a I_u b → b I_u c → a I_u c → a I_{ub} c

I-co2-r : a I_u b → b I_u c → a I_{ub} c → a I_u c

The e and r suffixes in the name I-co2 refer to extending and reducing the context. (We have suppressed the formal notation for snoc-lists in the contexts here; we follow this approach also in examples where the usual "silent" notation for strings is more readable.)

The original motivation for context-dependent independence relation in [20] was to make the independence relation more expressive or finer. However, without any restrictions, the context-dependent independence relation could become too fine. Consider an alphabet with a $I_{[]}$ b, c I_{ab} d and c D_{ba} d. Although we say that a and b are independent in the empty context, we could argue that they are not. If c and d are independent in the context ab, but dependent in the context ba, then the ordering of a and b (in the empty context) matters. The consistency and coherence conditions ensure that a context-dependent independence relation is sufficiently coarse. This is necessary for our generalizations of the Foata and lexicographic normal forms and the corresponding normalization algorithms.

Consistency is a very basic hygiene condition. It just states that equivalent strings must give the same independence relation. This rules out the undesirable behaviour described in

the previous paragraph. The coherence conditions are more interesting since they involve different contexts. Looking at their shape, we could say that the essence of the coherence conditions is that, if we have a letter b so that both a and b as well as b and c are independent (in some context), then b is "independent enough" (wrt. a and c), so that the independence or dependence of a and c is not affected by adding b to the context or removing it. The important details in the rules are the contexts. There is no need for a version of I-co1 for extending the context as that is derivable from the two I-co2 conditions.

One way to see the coherence conditions is to say that they are the smallest set of conditions guaranteeing that any choice of three conditions, one from each of the following three pairs, implies the other three conditions: (a I_u b, a I_{uc} b), (b I_u c, b I_{ua} c), (a I_u c, a I_{ub} c). This is with the exception of the choice of the second condition from each pair; from these three conditions one cannot conclude anything. For example, a I_u b and b I_{ua} c and a I_{ub} c imply not only a I_u c and b I_u c (both by I-co-1), but also a I_{uc} b (follows from those by I-co2-e).

We illustrate generalized traces with a modification of our previous example. We now take I to be the least consistent, coherent family of symmetric relations such that a $I_{[]}$ b, a $I_{[]}$ d, b I_a d, b I_{ac} d, c I_{ab} d. Explicitly, this means that we also have b $I_{[]}$ d (by I-co2-r), a I_d b, a I_b d (by I-co2-e) and c I_{ba} d (by I-cons). Now abcd has the same equivalence class as before, but acbd is only equivalent to acdb, leaving adcb and dacb in a different equivalence class.

4 Generalized Foata normalization

In this section we describe Foata normal forms for generalized traces and the corresponding normalization algorithm. We conclude with the correctness proof of the algorithm.

4.1 Normal forms

We represent a Foata normal form as a snoc-list of steps which in turn are snoc-lists of letters. We define Step as a synonym for List> A and Foata as a synonym for List> Step. These are the types of "raw" steps and normal forms. In order to define well-formed normal forms, we introduce some auxiliary notation. We define s ■I_u a to be All (λ b → b I_u a) s, expressing that, for every letter b in s, we have that b and a are independent in the context u. Similarly, we define s ♦D_u a to be Any (λ b → b D_u a) s, expressing that, for some letter b in s, we have that b and a are dependent in the context u. Generally, the proposition All P xs holds when the predicate P holds on every element of the list xs. A proof of Any P xs points to some element in the list xs that satisfies the predicate P.

A step (in a context u) is considered well-formed if it satisfies the following predicate.

Definition 3 StepOk : List> A → Step → Set

$$\frac{}{\text{StepOk u [a]}}\ \text{sngl}$$

$$\frac{\text{StepOk u (s :> b)} \quad \text{b} \prec \text{a} \quad \text{(s :> b)} \blacksquare I_u \text{ a}}{\text{StepOk u (s :> b :> a)}}\ \text{snoc}$$

A well-formed step (in a context u) is either a singleton or it consists of a well-formed step to which a new letter is added on the right, which has to be greater than the previous rightmost letter. The added letter and the step must be independent.

We now turn to well-formed normal forms. A letter in a step of a Foata normal form must have a dependent letter supporting it in the previous step. We formalize this by saying that the preceding normal form ss has to support the letter a, written as Sup ss a. This is defined as P-ne (λ tt t → t ♦$D_{\text{concat> tt}}$ a) ss. We use here a small helper P-ne P xs which holds trivially when xs is empty, and when xs is non-empty, it requires that P ys y holds where ys and y are the tail and head of the snoc-list xs. A "raw" Foata normal form (a list of steps) is well-formed if it satisfies the following predicate.

Definition 4 FoataOk : Foata → Set

$$\frac{}{\text{FoataOk []}}\ \text{empty}$$

$$\frac{\text{FoataOk ss} \quad \text{StepOk (concat> ss) s} \quad \text{All (Sup ss) s}}{\text{FoataOk (ss :> s)}}\ \text{step}$$

Thus a well-formed Foata normal form can either be the empty list of steps or consist of a well-formed normal form with an added step. This step must be well-formed in the context of the normal form and every letter in the added step must be supported by the normal form.

The function emb that embeds a normal form into strings is defined as emb ss = s2c (concat> ss).

In the example from the end of the last section, we have that (abd)(c) is a Foata normal form since we have a $I_{[]}$ b, a $I_{[]}$ d and b $I_{[]}$ d making (abd) a valid step and a $D_{[]}$ c ensuring that the sole letter in the step (c) is supported. We also have that (a)(c)(bd) is a normal form since b I_{ac} d ensures that the step (bd) is well-formed and a $D_{[]}$ c, c D_a b and c D_a d provide the requisite support for the letters in the steps (c) and (bd).

4.2 Normalization

The main ingredient in the normalization algorithm is a function that takes a normal form and a letter and inserts the letter into its right place in the normal form.

We define a function find> parameterized by a decider P? of a predicate P on a context (a snoc-list) and an element. It splits a given snoc-list xs into two parts, ls and rs, so that all of the elements in rs satisfy the predicate and the rightmost element in ls violates the predicate.

Algorithm 1 (find>)

```
find> : (∀ xs x → Dec (P xs x)) → List> X
      → List> X × List> X
find> P? [] = [] , []
find> P? (xs :> x) with P? xs x
find> P? (xs :> x) | yes _ =
  let ls , rs = find> P? xs in ls , rs :> x
find> P? (xs :> x) | no _ = xs :> x , []
```

Given a step and a letter, we can use find> to find the right position for the letter in the step.

Algorithm 2 (insert-s)

```
insert-s : Step → A → Step
insert-s s a =
  let ls , rs = find> (λ _ b → a ≺? b) s in
  ls :> a +> rs
```

Here we use find> with a predicate that ignores the context. The step s is split into ls and rs so that everything in rs is greater than a and the rightmost letter in ls is not. We assume that the ordering relation $\prec$ is decidable, with $\prec?$ as the decider. Hence a $\prec?$ b is either yes (together with a proof of a $\prec$ b) or no (together with a proof of $\neg$ a $\prec$ b).

Given a normal form and a letter, we use find> to find the correct step for the letter.

Algorithm 3 (insert)

```
insert : Foata → A → Foata
insert ss a with find> (λ tt t → t ■I?concat> tt a) ss
insert ss a | ls , []       = ls :> ([] :> a)
insert ss a | ls , rs :> r = let s , rs′ = first rs r in
  ls :> insert-s s a +> rs′
```

Here find> splits the normal form into two parts, ls and rs, so that all the steps in rs are independent of a and the rightmost step in ls is dependent (or ls is empty). If rs is empty, then we add a new step to the normal form. Otherwise, we insert a into the leftmost step in rs (the function first extracts leftmost element in a non-empty snoc-list). We assume that the independence relation I is decidable, with a decider I?. Here we use a derived decider ■I? for deciding whether a step and a letter are independent in a context.

The normalization function just traverses the input string from the left to the right and inserts each letter into the correct position in the accumulated normal form.

Algorithm 4 (norm)

```
norm′ : Foata → String → Foata
norm′ ss []       = ss
norm′ ss (a <: t) = norm′ (insert ss a) t

norm : String → Foata
norm t = norm′ [] t
```

We continue with our example and look at the evolution of the accumulator as the string bacd is normalized. First, the letter b is inserted into the empty normal form, resulting in the normal form (b). Next, the letter a is inserted into this normal form, which results in (ab) because of a $I_{[]}$ b. Next, the letter c is inserted into the result. We have a $D_{[]}$ c, which means that a new step must be added and the result is (ab)(c). We now need to insert d into the normal form. We have c I_{ab} d and in addition we also have a $I_{[]}$ d and b $I_{[]}$ d. This makes the first step the earliest possible step for d and the result is (abd)(c).

We have defined the Foata normalization function, but we have no assurance yet that it produces well-formed Foata normal forms (i.e. elements of Foata satisfying the FoataOk predicate). We will now proceed to show that the function norm constructs a well-formed normal form from the input string.

We start by showing that inserting a letter into a well-formed step gives a well-formed step.

Lemma 1 (insert-sOk) ∀ u s a →
StepOk u s → s ■I_u a → StepOk u (insert-s s a)

Proof Since s is a well-formed step, the letters in it are sorted wrt. $\prec$ and unique (by irreflexivity of the independence relation). insert-s splits s into ls and rs so that a is less than every letter in rs and a is not less than the rightmost letter in ls. We also have that s and a are independent, which implies independence of ls and rs of a. This allows us to construct StepOk u (ls :> a +> rs). □

To outline what we need to do next, let us look at a small example. Suppose we have a normal form stuv consisting of steps s, t, u, and v, and we wish to insert the letter a into this normal form. It so happens that a will go into the step t. This means that, instead of the old context st, the letters in u must now be independent in the new context s(insert-s t a). Likewise, the letters in v must now be independent in the context s(insert-s t a)u. Furthermore, every letter in v must now be supported by a letter in u in the context s(insert-s t a).

To show that the independence of letters in a step is preserved during an insert that inserts a letter into the context, we have the following lemma.

Lemma 2 (step-ext) ∀ u s a →
StepOk u s → s ■I_u a → StepOk (u :> a) s

Proof Since s is a well-formed step, we know that for any two distinct letters b and c from s we have b I_u c. We also have b I_u a and c I_u a. Using I-co2-e, we can derive b $\mathsf{I}_{u:>a}$ c. This means that pairwise independence of letters in s has been preserved in the extended context and the step is still well-formed. □

Next, we are considering the situation where we are inserting the letter a into the normal form ss :> s :> t and we have determined that a must go into a step in ss. We wish to show that the letters in t are still supported after the insert. We use PW I_u s to express that the predicate I_u holds between any two letters in s, i.e. the letters in s are pairwise independent in the context u. The normal form ss is considered here as the context u, and b is a letter from the step t.

Lemma 3 (♦D-ext-lem) ∀ u s a b →
b $\mathsf{I}_{u+>s}$ a → PW I_u s → s ■I_u a →
s ♦D_u b → s ■$\mathsf{I}_{u:>a}$ b → ⊥

Proof From our assumptions, we have that there is a letter d in s such that d D_u b and this d must also satisfy d I_u a and d $\mathsf{I}_{u:>a}$ b. We have derived a version of I-co1 (named ■I-co1) that decreases the context not by a letter but by a step. We use this on s ■I_u a, a $\mathsf{I}_{u+>s}$ b, s ■$\mathsf{I}_{u:>a}$ b and PW I_u s to derive a I_u b. We use I-co2-r on d I_u a, a I_u b, and d $\mathsf{I}_{u:>a}$ b to get d I_u b. This contradicts d D_u b. □

This shows that, under suitable conditions, we can add a letter to the end of the context and still have a supporting letter in the previous step. From I-cons, we know that this support is then preserved for any equivalent context. To show that insert preserves the equivalence class, we first show that we can "slide" an independent letter past a step without changing the equivalence class.

Lemma 4 (slide-step) ∀ u s a →
s ■I_u a → PW I_u s →
s2c (u +> s :> a) ∼* s2c (u :> a +> s)

Proof The proof is by induction on s. In the case where s = s′ :> b, we have s′ ■I_u a, b I_u a, PW I_u s′, s′ ■I_u b and we get b $\mathsf{I}_{u+>s'}$ a by a derived version of I-co2-e that allows us to extend the context by a step. This allows us to swap b and a in u +> s′ :> b :> a to get u +> s′ :> a :> b and then apply induction hypothesis. □

Lemma 5 (insert-lem) ∀ ss a →
FoataOk ss → emb (insert ss a) ∼* emb ss <+ [a]

Proof The proof follows the analysis of ss done by insert. When a new singleton step is added (ss is empty or ends with a step that supports a), then the two sides are equal and we are done. When a is inserted into the last step s, then s and a must be independent and s is split into ls and rs. According to slide-step, we can slide a past rs to the end of the normal form without changing the equivalence class. When a is inserted into an earlier step, then we use induction hypothesis to slide the letter a from its inserted position to the beginning of the last step and, since a and the last step must have been independent to begin with, we can slide it past that step to the end of the normal form. □

Lemma 6 (sup-insert-lem) ∀ ss s a b →
FoataOk (ss :> s) → a $\mathsf{I}_{\mathsf{concat}>\ (\mathsf{ss}:>\mathsf{s})}$ b →
¬ Sup (ss :> s) b → ¬ Sup ss b → Sup (ss :> s) a →
Sup (insert (ss :> s) b) a

Proof Since b is supported by neither ss :> s nor ss, we know that insert (ss :> s) b is the same as insert ss b :> s. Thus we need to show Sup (insert ss b :> s) a. Since Sup (ss :> s) a just means the existence of a dependent letter in s, we use ♦D-ext-lem to show that it cannot be the case that s ■$\mathsf{I}_{\mathsf{concat}>\ \mathsf{ss}:>\mathsf{b}}$ a. From insert-lem, we know that this context is equivalent to concat> (insert ss b) and so there must still be a supporting letter in s after the insert, thus Sup (insert ss b :> s) a. □

Lemma 7 (insertOk) ∀ ss a →
FoataOk ss → FoataOk (insert ss a)

Proof The proof follows the analysis of ss done by insert. When a letter a is inserted into a particular step s, then insertsOk ensures that the resulting step is valid. When insert goes past a step s with the letter a, then step-ext ensures that the step s is still valid in the context extended with a and insert-lem ensures that s is valid after the insert. When insert goes past two steps, s and t, with the letter a, then sup-insert-lem ensures that all the letters in t are still supported by s in the context resulting from the insert. □

Lemma 8 (norm′Ok) ∀ ss t →
FoataOk ss → FoataOk (norm′ ss t)

Proof The proof is by induction on t and just applies insertOk in the step case. □

Proposition 1 (normOk) ∀ t → FoataOk (norm t) □

4.3 Correctness

The correctness proof of the normalization algorithm consists of the proofs of the soundness and completeness properties. By soundness we mean that equivalent strings must get assigned the same normal form. By completeness we mean that any two strings that get assigned the same normal form must be equivalent.

The key lemma for completeness is that the result of normalizing a string (and then embedding it) is equivalent to that string.

Lemma 9 (nf-exists′) ∀ ss t →
FoataOk ss → emb (norm′ ss t) ∼* emb ss <+ t

Proof The proof is by induction on t. In the step case, we use insertOk to show that inserting the first letter of t into ss is a normal form and then apply induction hypothesis. The equivalence follows from insert-lem. □

Proposition 2 (nf-exists) ∀ t → emb (norm t) ∼* t □

Corollary 1 (completeness) ∀ t t′ →
norm t = norm t′ → t ∼* t′

Proof Apply nf-exists to both t and t′. □

To prove soundness of the normalization algorithm, we first show the commutativity of the normalization algorithm for independent letters. We start by showing that the order in which we insert two independent letters into the same step does not matter.

Lemma 10 (insert-s-commutes) ∀ u s a b →
StepOk u s → a $\mathsf{I}_{u +> s}$ b → s ■I_u a → s ■I_u b →
insert-s (insert-s s a) b = insert-s (insert-s s b) a

Proof By insert-sOk, we know that insert-s produces well-formed steps. The letters in a well-formed step are sorted wrt. ≺ and unique. This means that the two ways to insert the two letters must result in the same step. Hence the order of the inserts does not matter. □

Lemma 11 (insert-commutes) ∀ ss a b →
FoataOk ss → a $\mathsf{I}_{\text{concat}> ss}$ b →
insert (insert ss a) b = insert (insert ss b) a

Proof There are three cases to consider: both letters are supported by ss, only one of them is, or neither of them is.

In the first case, a new step is added no matter whether we insert a first or b first. Since a and b are independent and supported by ss, both of them end up in the new step. The order in which the letters are inserted into the new step does not matter as the result must agree with the ordering on the alphabet.

In the second case, say that a is the letter supported by ss. When we first insert a and then b, then a singleton step for a is added. Since a and b are independent and b is not supported by ss, it must be that b is inserted to some step in ss, i.e. insert (ss :> [a]>) b = insert ss b :> [a]>. This step is the same where b would be inserted if it were inserted first. Since a is supported by ss, then insert ss b must also support a (this follows from sup-insert-lem) and thus a new singleton step must be added also in this case.

In the third case, both letters are inserted into ss. Since ss cannot be empty, we take ss = ss′ :> s. Here we perform another case analysis on into which steps the letters go. If both go into s, then we apply insert-s-commutes. If one of the letters, say b, goes into ss′ and the other into s, then we argue similarly to the second case: if a is supported by ss′, then it is also supported by insert ss′ b, and, if s and b are independent, then so are insert-s s a and b. Thus the order of inserts does not matter. If both a and b go into ss′, then we apply induction hypothesis. □

Lemma 12 (norm′-commutes) ∀ ss a b →
FoataOk ss → a $\mathsf{I}_{\text{concat}> ss}$ b →
norm′ ss (a <: b <: []) = norm′ ss (b <: a <: [])

Proof This follows from insert-commutes. □

Lemma 13 (norm′-append) ∀ ss s t →
norm′ ss (s <+ t) = norm′ (norm′ ss s) t

Proof By induction on s. □

Lemma 14 (sound∼) ∀ ss t t′ →
FoataOk ss → t $\sim_{\text{concat}> ss}$ t′ → norm′ ss t = norm′ ss t′

Proof We have that t and t′ differ only by the ordering of two adjacent independent letters, i.e. t = uabv and t′ = ubav for some u, v, a and b. The result follows from norm′-commutes (norm′ ss u) a b. We use norm′-append twice on both sides to get the result. □

Lemma 15 (soundness′) ∀ ss t t′ →
FoataOk ss → t $\sim^*_{\text{concat}> ss}$ t′ → norm′ ss t = norm′ ss t′

Proof By induction on t $\sim^*_{\text{concat}> ss}$ t′. □

Proposition 3 (soundness) ∀ t t′ →
t ∼* t′ → norm t = norm t′ □

The soundness and completeness proofs give us a certified decision procedure for checking whether two strings are equivalent: normalize the two strings and check whether the normal forms are the same.

Algorithm 5 (equivalent?)
equivalent? : (t t′ : String) → Dec (t ∼* t′)
equivalent? t t′ with foata-eq? (norm t) (norm t′)
equivalent? t t′ | yes feq = yes (completeness feq)
equivalent? t t′ | no ¬feq =
  no (λ eqv → ¬feq (soundness eqv))

This procedure will either return yes, together with instructions how to turn u into v (which letters need to be exchanged), or no, together with a proof that it is not possible to turn u into v. Here foata-eq? uses the decidable equality on the alphabet to decide whether the two normal forms are the same.

Proposition 4 (stability) ∀ ss →
FoataOk ss → norm (emb ss) = ss

Proof This is by induction on the normal form. In the non-empty case, we have to show that renormalizing a step s in the context of the preceding normal form ss′ results in the same step. This is the case since we started with a normal form and thus the step is well-formed and every letter in the step is supported by the preceding normal form. By induction hypothesis, we know that renormalizing ss′ results in ss′. Since letters from s are supported by ss′, this means that no letter from s can be inserted into ss′. Similarly, all the letters from s fit into the same step. Hence the result is s. □

Corollary 2 (nf-unique) $\forall$ ss tt $\to$
FoataOk ss $\to$ FoataOk ss′ $\to$
emb ss $\sim^*$ emb ss′ $\to$ ss = ss′

Proof This follows from stability and soundness. □

5 Generalized lexicographic normalization

In this section, we describe the lexicographic normal form for generalized traces and the corresponding normalization algorithm. The section concludes with the correctness proof of the normalization algorithm.

5.1 Normal forms

We represent a "raw" lexicographic normal form as a snoc-list of letters (List> A). The embedding function emb of normal forms into strings is s2c.

We consider a list of letters to be a well-formed lexicographic normal form when each letter in it is in a valid position. Similarly to the previous section, a letter is in a valid position in a normal form if it is supported by the preceding normal form.

Definition 5 Sup : List> A $\to$ A $\to$ Set

$$\frac{\text{P-ne}\ (\lambda\ s'\ b \to b\ D_{s'}\ a)\ s}{\text{Sup}\ s\ a}\ \text{D-sup}$$

$$\frac{\text{Sup}\ s\ a \quad b\ I_s\ a \quad b \prec a}{\text{Sup}\ (s :> b)\ a}\ \text{I-sup}$$

Hence a letter is supported by a (snoc-)list if either the list is empty or ends with a dependent letter or the tail of the list supports the letter and the head is independent of and smaller than the letter. A list of letters is a well-formed lexicographic normal form when every letter in the list is supported.

Definition 6 LexOk : List> A $\to$ Set

$$\frac{}{\text{LexOk}\ []}\ \text{nil} \qquad \frac{\text{LexOk}\ s \quad \text{Sup}\ s\ a}{\text{LexOk}\ (s :> a)}\ \text{snoc}$$

We continue with our example and show that abcd is a lexicographic normal form. From LexOk [] and Sup [] a, we get LexOk a. Next, we get Sup a b from Sup [] b using I-sup and thus get LexOk ab. We have Sup ab c by D-sup and b D_a c resulting in LexOk abc. Finally, we get Sup abc d from Sup [] d by applying I-sup three times. Thus the list abcd is a well-formed normal form.

We now define what a "chain" of independent letters wrt. a letter is.

Definition 7 Cl : List> A $\to$ List> A $\to$ A $\to$ Set

$$\frac{}{[]\ \text{Cl}_u\ a} \qquad \frac{b\ I_{u\ +>\ s}\ a \quad s\ \text{Cl}_u\ a}{(s :> b)\ \text{Cl}_u\ a}$$

When s is a chain of independent letters wrt. a, then we can "slide" a past s: emb (s :> a) $\sim_u^*$ a <: emb s.

Anisimov and Knuth's characterization of lexicographic normal forms forbids the "bua" pattern. Here we show that our definition also forbids this pattern.

Proposition 5 (LexOk-bua) $\forall$ t u v a b $\to$
LexOk (t :> b +> u :> a +> v) $\to$
a I_t b $\to$ a $\prec$ b $\to$ u $\text{Cl}_{t\ :>\ b}$ a $\to \bot$

Proof The proof is by induction on v. In the empty case, we have that a is supported by something since it is the last letter in a normal form. The actual support must come from t since a is independent with both b and u (in the relevant contexts). This however means that b $\prec$ a since the support for a must have been constructed by I-sup. This contradicts our assumption. In the case where v is non-empty, we apply induction hypothesis. □

We use the strict total order $\prec$ on A to define the corresponding lexicographic order relation on strings, both in the non-strict and strict versions, and prove that it is a total order.

Definition 8 $\preceq_{\text{Lex}}$: String $\to$ String $\to$ Set

$$\frac{}{[]\ \preceq_{\text{Lex}}\ t}\ \text{nil}$$

$$\frac{a \prec b}{a <: s \preceq_{\text{Lex}} b <: t}\ \text{lt} \qquad \frac{a = b \quad s \preceq_{\text{Lex}} t}{a <: s \preceq_{\text{Lex}} b <: t}\ \text{eq}$$

Lemma 16 (antisym-$\preceq_{\text{Lex}}$) Antisymmetric $\preceq_{\text{Lex}}$

The lexicographic normal form has to be the least element in its equivalence class wrt. the lexicographic order $\preceq_{\text{Lex}}$. Here we show that the normal forms we have defined are indeed lexicographically smaller than any other string in their equivalence class.

Lemma 17 (LexOk-lex′) $\forall$ u s t $\to$
LexOk (u +> s) $\to$ emb s $\sim_u^*$ t $\to$ emb s $\preceq_{\text{Lex}}$ t

Proof The proof is by induction on the strings s and t. If both are empty, then we are done since [] $\preceq_{\mathsf{Lex}}$ []. In the cases where one is empty and the other is not, we have a contradiction since equivalent strings must have the same length. In the case where both are non-empty (a <: emb s and b <: t), we perform case analysis on the head elements. If a $\prec$ b, then we are done. If b $\prec$ a, then we have a contradiction since b must be somewhere in emb s (by the equivalence) and the letters before b (including a) must be independent with it. This creates a forbidden pattern in u +> s. In the case of a = b, we get by induction hypothesis that emb s $\preceq_{\mathsf{Lex}}$ t (after moving a to the context u). □

Proposition 6 (LexOk-lex) ∀ s t →
LexOk s → emb s $\sim^*$ t → emb s $\preceq_{\mathsf{Lex}}$ t

5.2 Normalization

The main ingredient in the normalization algorithm is a function that inserts a letter into its correct position in a list (which is assumed to be a well-formed normal form). Given a string s and a letter a, the idea is to split the string s into three parts: sd, sp, and si so that sd ends with a letter dependent on a, all letters in sp are independent of and smaller than a, and letters in si are independent of a and the first letter of si is greater than a.

Algorithm 6 (findPos)
```
findPos : List> A → A → List> A × List> A × List> A
findPos []        a = [] , [] , []
findPos (s :> b) a with b I?ₛ a
findPos (s :> b) a | no _ = s :> b , [] , []
findPos (s :> b) a | yes _ with findPos s a
findPos (s :> b) a | yes _ | sd , sp , si :> i =
  sd , sp , si :> i :> b
findPos (s :> b) a | yes _ | sd , sp , [] with b ≺? a
findPos (s :> b) a | yes _ | sd , sp , [] | no _ =
  sd , sp , [] :> b
findPos (s :> b) a | yes _ | sd , sp , [] | yes _ =
  sd , sp :> b , []
```

The function findPos implements the described functionality. Like before, we assume that the independence relation I and the order relation $\prec$ are decidable, with deciders I? and $\prec$?. The insert function now just plugs the letter between sp and si in the result of findPos.

Algorithm 7 (insert)
```
insert : List> A → A → List> A
insert s a =
  let sd , sp , si = findPos s a in
  sd +> sp :> a +> si
```

The normalization algorithm just traverses the input string letter by letter and inserts the letters into the accumulating normal form, as in Foata normalization.

Algorithm 8 (norm)
```
norm' : List> A → String → List> A
norm' s []       = s
norm' s (a <: t) = norm' (insert s a) t

norm : String → List> A
norm t = norm' [] t
```

We continue with our example and look at what are the intermediate steps when normalizing bacd. First, when inserting b into the empty normal form, insert splits it into [], [], [] and the result is b. Next, when inserting a, the normal form b is split into [], [], b since a $\prec$ b and b $I_{[]}$ a. The result is ab. When inserting c into ab, the split is ab, [], [] and the result is abc. Finally, when inserting d into abc, the split is [], abc, [] and the result is abcd.

We have defined the lexicographic normalization algorithm. This produces "raw" normal forms, i.e. just snoc-lists. We will now show that these snoc-lists are well-formed normal forms. We begin with a couple of lemmas exhibiting that findPos behaves as expected.

The first lemma says that findPos just splits the input string.

Lemma 18 (findPos-split) ∀ s a →
let sd , sp , si = findPos s a in
sd +> sp +> si = s

Proof From the definition of findPos, it is clear that it does not rearrange the letters in s (b always stays to the right of the result of the recursive call to findPos). The proof just follows the analysis of s done by findPos. □

The next lemma ensures that the si component in the result of findPos consists of a "chain" of independent letters.

Lemma 19 (findPos-I) ∀ s a →
let sd , sp , si = findPos s a in
si $\mathsf{Cl}_{\mathsf{sd} +> \mathsf{sp}}$ a

Proof Here the proof also follows the analysis of s done by findPos. When b is added to the si component in the result, then we know that b and a must be independent. Induction hypothesis is used when the si component of the result of the recursive call is non-empty. □

The next lemma ensures that the leftmost letter of si in the result of findPos is greater than the letter a. The proposition a $\prec$first si holds when a is less than the first letter of si.

Lemma 20 (findPos-$\prec$first) ∀ s a →
let _ , _ , si = findPos s a in
a $\prec$first si

Proof From the definition of findPos, we see that when the first letter is added to the si component, then it must be greater than a since it is not smaller than a and cannot be equal by irreflexivity of independence. □

We now show that insert preserves the equivalence class in the following sense: the normalization algorithm uses insert in the situation where a prefix s has been normalized to nf and the suffix a <: t is yet to be normalized. Then insert will find the right place for a in nf and the result of that should be equivalent to nf :> a.

Lemma 21 (insert-lem) $\forall\ s\ a \to$
$\mathsf{emb}\ (\mathsf{insert}\ s\ a) \sim^* \mathsf{emb}\ s <\!\!+ [a]$

Proof By definition, insert plugs a between sp and si. From findPos-l, we get that si is a chain of independent letters and thus we can move a past it without changing the equivalence class. □

The next lemma ensures that under certain conditions the support of a letter is preserved when another letter is inserted into the supporting string.

Lemma 22 (slideSup) $\forall\ s\ ii\ i\ b\ a \to$
$\mathsf{Sup}\ (s +\!\!> ii :\!\!> i)\ a \to (ii :\!\!> i)\ \mathsf{Cl}_s\ b \to$
$b\ \mathsf{I}_{s +\!\!> ii :\!\!> i}\ a \to b \prec \mathsf{first}\ (ii :\!\!> i) \to$
$\mathsf{Sup}\ (s :\!\!> b +\!\!> ii :\!\!> i)\ a$

Proof The proof is by induction on Sup (s +> ii :> i) a. The base case is D-sup, which means that we have $i\ \mathsf{D}_{s +\!\!> ii}\ a$ and we need to show that $i\ \mathsf{D}_{s :\!\!> b +\!\!> ii}\ a$. This follows from I-co1 and I-cons. In the I-sup case, we have $i\ \mathsf{I}_{s +\!\!> ii}\ a$, but we do not know which is the dependent letter that supports a. If ii is empty, then b is inserted immediately before i and we construct I-sup using I-co2-r and I-co2-e. In the non-empty case, we construct the support from induction hypothesis and use I-co2-r and I-co2-e to show that i and a are still independent when the head of ii is added to the support. □

Lemma 23 (insertOk) $\forall\ s\ a \to$
$\mathsf{LexOk}\ s \to \mathsf{LexOk}\ (\mathsf{insert}\ s\ a)$

Proof The proof follows the analysis of s done by findPos. The result follows from slideSup and the preceding lemmas about findPos. □

Lemma 24 (norm'Ok) $\forall\ s\ t \to$
$\mathsf{LexOk}\ s \to \mathsf{LexOk}\ (\mathsf{norm}'\ s\ t)$

Proof The proof is by induction on t and applies insertOk in the step case. □

Proposition 7 (normOk) $\forall\ t \to \mathsf{LexOk}\ (\mathsf{norm}\ t)$ □

5.3 Correctness

The key lemma for the completeness proof is that the result of normalizing a string is equivalent to that string.

Lemma 25 (nf-exists') $\forall\ s\ t \to$
$\mathsf{emb}\ (\mathsf{norm}'\ s\ t) \sim^* \mathsf{emb}\ s <\!\!+ t$

Proof The proof is by induction on t. In the step case, we use induction hypothesis together with insert-lem to show that inserting the first letter of t into s does not change the equivalence class. □

Proposition 8 (nf-exists) $\forall\ t \to \mathsf{emb}\ (\mathsf{norm}\ t) \sim^* t$ □

Corollary 3 (completeness) $\forall\ t\ t' \to$
$\mathsf{norm}\ t = \mathsf{norm}\ t' \to t \sim^* t'$

Proof Apply nf-exists to both sides of the equation. □

Continuing towards soundness, we first prove the uniqueness of normal forms.

Proposition 9 (nf-unique) $\forall\ s\ s' \to$
$\mathsf{LexOk}\ s \to \mathsf{LexOk}\ s' \to \mathsf{emb}\ s \sim^* \mathsf{emb}\ s' \to s = s'$

Proof From the assumptions, by LexOk-lex we get both $\mathsf{emb}\ s \preceq_{\mathsf{Lex}} \mathsf{emb}\ s'$ and $\mathsf{emb}\ s' \preceq_{\mathsf{Lex}} \mathsf{emb}\ s$, from where by antisymmetry of $\preceq_{\mathsf{Lex}}$, we get emb s = emb s'. Since emb = s2c is injective (inverted by c2s), s = s' follows. □

Corollary 4 (soundness) $\forall\ t\ t' \to$
$t \sim^* t' \to \mathsf{norm}\ t = \mathsf{norm}\ t'$

Proof Applying nf-exists to both t and t', we get emb (norm t) $\sim^*$ emb (norm t'). The result follows from this by nf-unique and normOk. □

From uniqueness of normal forms, we also get the stability of the normalization algorithm.

Corollary 5 (stability) $\forall\ s \to$
$\mathsf{LexOk}\ s \to \mathsf{norm}\ (\mathsf{emb}\ s) = s$

Proof This follows from nf-unique, normOk and nf-exists. □

An alternative approach to soundness would have been to prove the following lemma.

Lemma 26 (insert-commutes) $\forall\ s\ a\ b \to$
$\mathsf{LexOk}\ s \to a\ \mathsf{I}_s\ b \to$
$\mathsf{insert}\ (\mathsf{insert}\ s\ a)\ b = \mathsf{insert}\ (\mathsf{insert}\ s\ b)\ a$ □

This leads to soundness, stability and uniqueness as in the previous section.

Finally, we can now prove the converses of LexOk-lex and LexOk-bua showing that the least string in its equivalence class is the lexicographic normal form and that a string with no forbidden patterns is a lexicographic normal form.

Proposition 10 (lex-LexOk) $\forall\ s \rightarrow$
$(\forall\ t \rightarrow \mathsf{emb}\ s \sim^* t \rightarrow \mathsf{emb}\ s \preceq_{\mathsf{Lex}} t) \rightarrow \mathsf{LexOk}\ s$

Proof By nf-exists, $\mathsf{emb}\ s \sim^* \mathsf{emb}\ (\mathsf{norm}\ (\mathsf{emb}\ s))$. Thus, by the assumption, $\mathsf{emb}\ s \preceq_{\mathsf{Lex}} \mathsf{emb}\ (\mathsf{norm}\ (\mathsf{emb}\ s))$. At the same time, normOk gives us $\mathsf{LexOk}\ (\mathsf{norm}\ (\mathsf{emb}\ s))$, from where $\mathsf{emb}\ (\mathsf{norm}\ (\mathsf{emb}\ s)) \preceq_{\mathsf{Lex}} \mathsf{emb}\ s$ follows by LexOk-lex. By antisymmetry of $\preceq_{\mathsf{Lex}}$, we therefore have $\mathsf{emb}\ s = \mathsf{emb}\ (\mathsf{norm}\ (\mathsf{emb}\ s))$. Since emb is injective, this entails $s = \mathsf{norm}\ (\mathsf{emb}\ s)$. But $\mathsf{LexOk}\ (\mathsf{norm}\ (\mathsf{emb}\ s))$, so $\mathsf{LexOk}\ s$. □

Proposition 11 (bua-LexOk) $\forall\ s \rightarrow$
$(\forall\ t\ u\ v\ a\ b \rightarrow t :> b +> u :> a +> v = s \rightarrow$
$a\ \mathsf{I}_t\ b \rightarrow a \prec b \rightarrow u\ \mathsf{CI}_{t :> b}\ a \rightarrow \bot) \rightarrow$
$\mathsf{LexOk}\ s$

Proof If $s = \mathsf{norm}\ (\mathsf{emb}\ s)$, then we are done by normOk. If not, then we can factor the two as $s = tbuav$ and $\mathsf{norm}\ (\mathsf{emb}\ s) = tau'bv'$. By nf-exists, we have $\mathsf{emb}\ tbuav \sim^* \mathsf{emb}\ tau'bv'$. The letter after t is the first position where the two differ ($a \neq b$). We have $a\ \mathsf{I}_t\ b$, $u\ \mathsf{CI}_{t :> b}\ a$ and $u'\ \mathsf{CI}_{t :> a}\ b$ since norm has moved them past each other. If $a \prec b$, then we contradict the assumption that there are no forbidden patterns in s. If $b \prec a$, then there is a forbidden pattern in the normal form $\mathsf{norm}\ (\mathsf{emb}\ s)$. □

6 Example

Here we will give a small example where generalized traces are needed to describe the behaviour of a concurrent system reasonably precisely. The example is from shared-memory concurrency with write buffers. It corresponds to the Total Store Order (TSO) relaxed memory model of the SPARC family [21].

The machine that we are going to model consists of processors and shared memory where each processor has a single write buffer. A program consists of lists of read and write instructions. The execution of a write instruction proceeds in two stages: the write is first enqueued in the processor's write buffer, and some time later it is dequeued and written (committed) to memory. The execution of a read instruction reads the memory "through" the local write buffer: if there is a pending write to the location of the read, then the read operation reads its result from the latest pending write to that location, otherwise it reads the value from memory.

We think of program executions on this machine as strings over an alphabet A of events. The letters in this alphabet are tuples $\mathsf{Proc} \times \mathsf{Id} \times \mathsf{Action}$ where Proc (the processor identifier) and Id (the event identifier) are both natural numbers and Action is a pair $\mathsf{Op} \times \mathsf{Loc}$ where Op is either R (read), W (write) or C (commit) and Loc is the memory location, which is also represented as a natural number. In this simple example we ignore the values read and written by the events because the independence relation we define does not consider them.

Two events from different processors are dependent when they access the same memory location and at least one of them is a write. We do not consider a W event to access the memory as it only affects the local write buffer. Similarly, an R event only accesses the memory when it reads its value from memory (not from the write buffer). Two events from the same processor are dependent if they are W or R events (we respect the program order) or both are C events (the buffer is *first-in-first-out*) or they are a corresponding pair of a W and C event (this C is the commit of this W event).

We have defined this context-dependent independence relation in our formalization and shown that it satisfies the consistency and coherence conditions. This allows us to use the normalization algorithms for deciding whether two executions (strings over the alphabet) on this machine are equivalent.

Let us consider the following program where two processors write to variable x and one of them also reads from x.

P_1		P_2	
(a,a')	[x] := 1	(c,c')	[x] := 2
(b)	r1 := [x]		

As mentioned before, the execution of write instructions proceeds in two stages. Thus, the events representing [x] := 1 are $a = 1, 1, W, 1$ and $a' = 1, 1, C, 1$; they share the event identifier as they correspond to the write and commit stages of the same instruction. The event for the read r1 := [x] is $b = 1, 2, R, 1$. The events representing [x] := 2 are $c = 2, 1, W, 1$ and $c' = 2, 1, C, 1$.

The possible executions of the first processor are $aa'b$ and aba'. This is a consequence of the independence relation: after performing a, the event a' is pending; since a' and b are independent, the processor has a choice which one to perform next. The second processor can only execute as cc'. The possible executions of the whole program are all the possible interleavings of an execution from the first processor with an execution from the second processor. This program has 20 possible executions in total and these are partitioned into three equivalence classes. This is summarized in the following table.

A consequence of this independence relation is that the executions $acc'ba'$ and $acbc'a'$ are equivalent. Interestingly, the two only differ by the ordering of b and c'. The equivalence of the two, justified by b and c' being independent in a context containing a but not a', may seem counterintuitive as b is supposed to read x and c' is supposed to write to x. It may look as if the value read by b could be affected by c'. But this is not so. The two executions are semantically

Execution	Foata	Lexicographic
aa′bcc′ aba′cc′ aa′cbc′ abca′c′ aca′bc′ acba′c′ caa′bc′ caba′c′	(ac)(a′b)(c′)	aa′bcc′
aa′cc′b aca′c′b caa′c′b	(ac)(a′)(c′)(b)	aa′cc′b
abcc′a′ acbc′a′ cabc′a′ acc′a′b acc′ba′ cac′a′b cac′ba′ cc′aa′b cc′aba′	(ac)(bc′)(a′)	abcc′a′

equivalent because b appears before a′ (i.e. in the presence of a pending write to x) and thus b reads the value of x from the write buffer and not from memory.

The effect described in the previous paragraph is also visible from the Foata normal forms in the table. The second one, (ac)(a′)(c′)(b), implies that c′ and b are dependent in the context aca′. The third one, (ac)(bc′)(a′), implies that b and c′ are independent in the context ac.

If we were to model this using ordinary traces (without context-dependence), then we would have to choose whether to set b and c′ to be dependent or independent. The only reasonable choice is to let them be dependent as the equivalence relation induced by this is a safe approximation of the one above: equivalent executions according to the new relation are equivalent according to the previous relation. The downside of this is that there will be more equivalence classes. If we would set b and c′ to be independent, then we would have that aca′c′b and aca′bc′ are equivalent. This does not agree with the equivalence relation defined above, but more importantly, it does not make sense semantically as b reads the value of x from memory (instead of the write buffer) and is thus affected by c′.

7 Related work

Traces were introduced into concurrency theory by Mazurkiewicz [16], but they originate from the enumerative combinatorics work by Cartier and Foata [6]. In particular, Foata normalization is from that work. The lexicographic normalization was first investigated by Anisimov and Knuth [3]. These two normal forms and normalization algorithms are described in many of the standard expositions of trace theory, e.g. [1,8].

Generalizing traces for context-dependent independence has been considered by several authors, but with different well-behavedness conditions on independence. Sassone et al. [20] introduced context-dependent independence as we have considered it. Katz and Peled [12] introduced conditional independence, considering a coherence condition that in our setting would amount to $a\ I_u\ b$ and $b\ I_{ua}\ c$ implying ($a\ I_u\ c$ iff $a\ I_{ub}\ c$). This condition is equivalent to the conjunction of conditions I-co-1 and I-co2-e of Sassone et al. Droste [9], in a work on concurrent automata, again with state-dependent independence, required what would in our setting amount to $a\ I_u\ b$, $b\ I_u\ c$, $a\ I_{ub}\ c$ implying $a\ I_{uc}\ b$, $b\ I_{ua}\ c$, $a\ I_u\ c$, i.e. condition I-co2-r and a little more.

Hoogers et al. [11] developed local traces where independence relates lists of steps to steps. This is a different setup where coherence conditions like those of Sassone et al. do not arise, because one only works with contexts of steps, not contexts of individual letters.

Partial-order reduction (POR) and use of representatives in model-checking, originally proposed by Godefroid [10] and Peled [19], are in wide use. The literature is too extensive to review here, especially as this paper is not on POR as such but specifically on normal forms of traces. We mention that dynamic POR for stateless model-checking of relaxed memory concurrent programs in particular has been considered by Abdulla et al. [2] and Zhang et al. [23]. In our own previous work [14], we used Foata normal forms for generalized traces for generating representative executions of all four memory models of the SPARC hierarchy.

Chou and Peled [7] have formalized standard Mazurkiewicz traces in the context of formally verifying a partial-order reduction technique in HOL. Yang et al. [22], Aspinall and Sevčik [4] and Owens et al. [18] pioneered the formalization of semantics of relaxed memory models with proof assistants, using HOL, Isabelle/HOL, HOL4.

There are some parallels of Lipton's theory of reduction (movers) [13] to trace theory, or more precisely, semi-commutation, where independence is non-symmetric. Movers have been used in reasoning about relaxed memory concurrency by Bouajjani et al. [5].

8 Conclusion and future work

We believe it to be important to exercise care when choosing the semantic domain for behaviours for a class of concurrent systems. Descriptions of behaviours in terms of an apparently more involved abstraction can sometimes be more precise, yet still analysable with less effort. In this paper, we certified two normalization algorithms for generalized Mazurkiewicz traces. The example from Sect. 6 demonstrates that standard

Mazurkiewicz traces are not flexible enough in some circumstances and generalized traces can lead to fewer equivalence classes. This is good in any situation where one needs to exhaustively check an equivalence-invariant property on all equivalence classes.

Here we have looked at normalization of generalized Mazurkiewicz traces. We also wonder whether something similar is possible for yet more flexible notions such as various specializations of pomsets.

Acknowledgements This work was supported by the ERDF funded Estonian national centre of excellence Project EXCITE (2014-2020.4.01.15-0018) and the Estonian Ministry of Education and Research institutional research Grant IUT33-13.

References

1. Aalbersberg IJJ, Rozenberg G (1988) Theory of traces. Theor Comput Sci 60(1):1–82
2. Abdulla PA, Aronis S, Atig MF, Jonsson B, Leonardsson C, Sagonas K (2015) Stateless model checking for TSO and PSO. In: Baier C, Tinelli C (eds) TACAS 2015. LNCS, vol 9035. Springer, Berlin, pp 353–367
3. Anisimov AV, Knuth DE (1979) Inhomogeneous sorting. Int J Comput Inf Sci 8(4):255–260
4. Aspinall D, Sevčik J (2007) Formalising Java's data race free guarantee. In: Schneider K, Brandt J (eds) TPHOLs 2007. LNCS, vol 4732. Springer, Berlin, pp 22–37
5. Bouajjani A, Enea C, Mutluergil SO, Tasiran S (2018) Reasoning about TSO programs using reduction and abstraction. In: Chockler H, Weissenbacher G (eds) CAV 2018, part 2. LNCS, vol 10982. Springer, Berlin, pp 336–353
6. Cartier P, Foata D (1969) Problemes combinatoires de commutation et réarrangements. LNM, vol 85. Springer, Berlin
7. Chou C-T, Peled D (1996) Formal verification of a partial-order reduction technique for model checking. In: Margaria T, Steffen B (eds) TACAS'96. LNCS, vol 1055. Springer, Berlin, pp 241–257
8. Diekert V, Métivier Y (1997) Partial commutation and traces. In: Rozenberg G, Salomaa A (eds) Handbook of formal languages. Beyond words, vol 3. Springer, Berlin, pp 457–553
9. Droste M (1990) Concurrency, automata and domains. In: Paterson MS (ed) ICALP'90. LNCS, vol 443. Springer, Berlin, pp 185–208
10. Godefroid P (1990) Using partial orders to improve automatic verification methods. In: Clarke EM, Kurshan RP (eds) CAV '90. LNCS, vol 531. Springer, Berlin, pp 176–185
11. Hoogers PW, Kleijn HCM, Thiagarajan PS (1995) A trace semantics for Petri nets. Inf Comput 117(1):98–114
12. Katz S, Peled D (1995) Defining conditional independence using collapses. Theor Comput Sci 101(2):337–359
13. Lipton RJ (1975) Reduction: a method of proving properties of parallel programs. Commun ACM 18(12):717–721
14. Maarand H, Uustalu T (2017) Generating representative executions. In: Vasconcelos VT, Haller P (eds) Proceedings of 10th workshop on programming language approaches to concurrency and communication-centric software, PLACES 2017. Electronic Proceedings in Theoretical Computer Science, vol 246. Open Publishing Association, Sydney, pp 39–48
15. Maarand H, Uustalu T (2018) Certified Foata normalization for generalized traces. In: Dutle A, Muñoz C, Narkawicz A (eds) NFM 2018. LNCS, vol 10811. Springer, Berlin, pp 299–314
16. Mazurkiewicz A (1977) Concurrent program schemes and their interpretations. DAIMI report PB-78, Aarhus University
17. Norell U (2009) Dependently typed programming in Agda. In: Koopman P, Plasmeijer R, Swierstra D (eds) AFP 2008. LNCS, vol 5832. Springer, Berlin, pp 230–266
18. Owens S, Sarkar S, Sewell P (2009) A better x86 memory model: x86-TSO. In: Berghofer S, Nipkow T, Urban C, Wenzel M (eds) TPHOLs 2009. LNCS, vol 5674. Springer, Berlin, pp 391–407
19. Peled D (1993) All from one, one for all: on model checking using representatives. In: Courcourbetis C (ed) CAV'93. LNCS, vol 697. Springer, Berlin, pp 409–423
20. Sassone V, Nielsen M, Winskel G (1993) Deterministic behavioural models for concurrency. In: Borzyszkowski AM, Sokolowski S (eds) MFCS'93. LNCS, vol 711. Springer, Berlin, pp 682–692
21. SPARC International Inc., Weaver DL (1994) The SPARC architecture manual. Prentice Hall, Upper Saddle River, NJ
22. Yang Y, Gopalakrishnan G, Lindstrom G, Slind K (2004) Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: Proceedings of 18th international parallel and distributed processing symposium, IPDPS 2004. IEEE, Los Alamitos, CA, pp 433–441
23. Zhang N, Kusano M, Wang C (2015) Dynamic partial order reduction for relaxed memory models. In: Proceedings of 36th ACM SIGPLAN conference on principles of language design and implementation, PLDI 2015. ACM, New York, pp 250–259