Certified Foata Normalization for Generalized Traces

Hendrik Maarand[1(✉)] and Tarmo Uustalu[2,1]

[1] Department of Software Science, Tallinn University of Technology, Akadeemia tee 21B, 12618 Tallinn, Estonia
hendrik@cs.ioc.ee

[2] School of Computer Science, Reykjavik University, Menntavegi 1, 101 Reykjavik, Iceland
tarmo@ru.is

Abstract. Mazurkiewicz traces are a well-known model of concurrency with a notion of equivalence for interleaving executions. Interleaving executions of a concurrent system are represented as strings over an alphabet equipped with an independence relation, and two strings are taken to be equivalent if they can be transformed into each other by repeatedly commuting independent consecutive letters. Analyzing all behaviors of the system can be reduced to analyzing one canonical representative from each equivalence class; normal forms such as the Foata normal form can be used for this purpose. In some applications, it is useful to have commutability of two adjacent letters in a string depend on their left context. We develop Foata normal forms and normalization for Sassone et al.'s context-dependent generalization of traces, formalize this development in the dependently typed programming language Agda and show generalized Foata normalization in action on an example from relaxed shared-memory concurrency (local reads in TSO).

1 Introduction

Strings over an alphabet are a simple model of concurrent program behavior presuming that events from different threads are interleaved in an execution of a program. Mazurkiewicz traces [11] are an improvement over strings; a trace corresponds to a set of interleaving executions that can be considered to be equivalent. Traces are equivalence classes of strings. Two strings are taken to be equivalent if they can be transformed to each other by a finite number of commutations of adjacent letters. Commutation is allowed for letters in a given irreflexive and symmetric binary relation, called the independence relation. Traces therefore enable distinguishing concurrent events and causally related events.

While traces are sets of strings, in practice it is desirable to deal with single strings representing these sets canonically. Ideally, such representatives should be strings in some kind of normal form, with normality defined by a decidable predicate. In every trace, there should be exactly one normal form. In particular, given a string, it should be possible to compute its normal form, i.e., the normal

© Springer International Publishing AG, part of Springer Nature 2018
A. Dutle et al. (Eds.): NFM 2018, LNCS 10811, pp. 299–314, 2018.
https://doi.org/10.1007/978-3-319-77935-5_21

form in its equivalence class. One natural such normal form is the Foata normal form, corresponding to maximally parallel executions; the Foata normal form is well known and understood. (Another such normal form is the lexicographic normal form; yet another is dependence graphs.) The possibility to reduce exploring the full set of executions of a program to exploring the executions in normal form is important in practice. It is often referred to as partial-order reduction.

In some concurrency applications, one needs 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 functions 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 Foata normalization 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. 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 as 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. But, as it turns out, everything works out well, if one assumes consistency and the coherence conditions introduced by Sassone et al. [15]. Still the definitions and proofs require considerably more care than in the standard case. Especially the proofs become quite subtle, it is easy to make mistakes. This makes context-dependent Foata normalization a good exercise in certified programs and proofs. We conducted this exercise in the dependently typed programming language Agda [12].

The contribution of this paper thus consists in developing the theory of generalized Foata normalization and formalizing it. We also demonstrate the usefulness of this generalization. The reporting is organized as follows. We first introduce both standard Mazurkiewicz traces and Foata normal forms and normalization as well as the context-dependent generalization using mathematical notation on a high level in Sect. 2. Then, in Sect. 3, we describe the formalization of the generalization in Agda, disclosing a fair degree of detail of not only the definitions, but also the proofs. In Sect. 4, we demonstrate that context-dependent independence arises naturally in relaxed shared-memory concurrency. Then we briefly discuss related work and conclude.

Our Agda formalization is at http://cs.ioc.ee/~hendrik/code/generalized-traces/agda.zip. The code works with Agda version 2.5.2 and Agda standard library version 0.13. It consists of approx. 3800 lines of code whereof standard traces take approx. 1100 lines, generalized traces approx. 1300 lines and the rest is utility code for cons/snoc list manipulation and similar purposes. The formalization of generalized traces does not depend on the formalization of standard traces.

We assume no knowledge of trace theory from the reader, introducing all relevant concepts and facts. When describing our formal development, we show snippets of Agda code, but we also comment them.

2 Traces, Foata Normal Forms and Normalization

Traces are equivalence classes of strings with respect to a congruence relation that allows to commute certain pairs of letters. More precisely, an *alphabet* Σ is a (non-empty) set whose elements we call letters; letters model events. A *string* is a list of letters, i.e., an element of Σ^*, the free monoid on Σ. Strings are used to model executions of programs. An *independence relation* $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric binary relation. Its complement $D = (\Sigma \times \Sigma) \setminus I$, which is reflexive and symmetric, is called the *dependence relation*. Intuitively, if aIb, then the strings $uabv$ and $ubav$ represent the "same" execution. We define $\sim \subseteq \Sigma^* \times \Sigma^*$ to be the least relation such that aIb implies $uabv \sim ubav$ 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 $\Sigma^*/{\sim^*}$, which is the free partially commutative monoid.

For example, if $\Sigma = \{a, b, c, d\}$ and I is the least symmetric relation satisfying aIb, aId, bId, cId, then the strings $abcd$ and $bdac$ are equivalent, since $abcd \sim bacd \sim badc \sim bdac$, 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 $\{acbd, acdb, adcb, dacb\}$. Altogether, there are only four traces containing each letter of Σ exactly once.

A *(Foata) step* is a non-empty set s of pairwise independent letters, i.e., for any different $a, b \in s$, we require aIb. If we are given a strict total order, i.e., a transitive and asymmetric relation, $\prec \subseteq \Sigma \times \Sigma$ on letters, then we can equivalently define that a step is a $\prec$-sorted non-empty list of pairwise independent letters. A *(Foata) normal form* $n : \mathsf{Nf}$ is a list $s_0 \ldots s_{m-1}$ of steps such that, for any $i < m$, unless $i = 0$, for any $b \in s_i$, there is $a \in s_{i-1}$ such that aDb.

To continue the example above, suppose that $\prec$ is given by $a \prec b \prec c \prec d$. We then have that $(abd)(c)$ is a normal form, since a, b, d are pairwise independent and c is dependent with, for instance, b. However, $(a)(c)(bd)$ is not a normal form: we have bId, aDc, cDb, but cDd does not hold.

Viewing steps as non-empty lists, we have a straightforward embedding $\mathsf{emb} : \mathsf{Nf} \to \Sigma^*$ of normal forms into strings given by concatenation.

Assuming I and $\prec$ to be decidable, every trace has a unique normal form, i.e., we have a function $\mathsf{norm} : \Sigma^* \to \mathsf{Nf}$ such that $u \sim^* \mathsf{emb}\ (\mathsf{norm}\ u)$ (existence of a normal form), $u \sim^* v$ implies $\mathsf{norm}\ u = \mathsf{norm}\ v$ (soundness of norm) and $n = \mathsf{norm}\ (\mathsf{emb}\ n)$ (stability of norm). Existence straightforwardly gives that $\mathsf{norm}\ u = \mathsf{norm}\ v$ implies $u \sim^* v$ (completeness of norm). From soundness and stability, it follows that $u \sim^* \mathsf{emb}\ n$ implies $\mathsf{norm}\ u = n$ (uniqueness of a normal form).

The function norm is defined quite naturally. It takes a string u and traverses it from the left to the right, maintaining the normal form of the prefix already seen. If the normal form of this prefix is $s_0 \dots s_{m-1}$, then the function finds the greatest $i \leq m$ such that the next letter a is dependent with some letter in s_{i-1} unless $i = 0$, and inserts a into step s_i, if $i < m$ (commuting past all steps whose all letters it is independent with), or adds a new singleton step s_m consisting initially of a only, if $i = m$. Intuitively, the given string is thus rearranged into a maximally parallel form.

In our example, the string $acbd$ is normalized as follows. We first make a normal form consisting of a single step (a). Letter c is dependent with a, thus cannot be added to this step, so we start a new step: $(a)(c)$. Letter b is dependent with c, so we start a new step again: $(a)(c)(b)$. Letter d is independent with all of b, c, a, so we insert it into the first step: $(ad)(c)(b)$.

In generalized traces, the commutability of two adjacent letters depends on their left context, corresponding to the execution so far. The independence relation is parameterized by a string for this context. More precisely, independence is an assignment of an irreflexive and symmetric relation $I_u \subseteq \Sigma \times \Sigma$ to every string u. This family of relations must be *consistent*, i.e., stable under equivalence in the sense that $u \sim^* v$ and $aI_u b$ imply $aI_v b$. Sassone et al. [15] require also the following *coherence* conditions:

1. $aI_u b$ and $bI_{ua} c$ and $aI_{ub} c$ imply $aI_u c$,
2. $aI_u b$ and $bI_u c$ imply ($aI_u c$ iff $aI_{ub} c$).

Consistency is a very basic hygiene condition. The coherence conditions are more difficult to make sense of and memorize. They also have many similar-looking consequences. 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: ($aI_u b$, $aI_{uc} b$), ($bI_u c$, $bI_{ua} c$), ($aI_u c$, $aI_{ub} c$). 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, $aI_u b$ and $bI_{ua} c$ and $aI_{ub} c$ imply not only $aI_u c$ and $bI_u c$ (both by 1.), but also $aI_{uc} b$ (follows from those by 2.$\Rightarrow$).

To illustrate generalized traces, let us modify our example. We take I now to be the least consistent, coherent family of symmetric relations such that $aI_{[]} b$, $aI_{[]} d$, $bI_a d$, $bI_{ac} d$, $cI_{ab} d$. (We write [] for the empty string.) Explicitly, this means that we also have $bI_{[]} d$ (by 2.$\Leftarrow$), $aI_d b$, $aI_b d$ (by 2.$\Rightarrow$) and $cI_{ba} d$ (by consistency). 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.

We would like to scale Foata normalization to generalized traces. Our adjustment of the definition of a Foata normal form is as follows. A *(Foata) normal form* is a list $s_0 \dots s_{m-1}$ of non-empty lists of letters (*steps*) such that, for all $i < m$,

– for any $a, b \in s_i$, if $a \neq b$, then $aI_{s_0 \dots s_{i-1}} b$,
– unless $i = 0$, for any $b \in s_i$, there is $a \in s_{i-1}$ such that $aD_{s_0 \dots s_{i-2}} b$,
– s_i is $\prec$-sorted.

Note that in the above conditions dependence or independence is stated wrt. contexts of whole steps rather than contexts of individual letters in a step. This is motivated by the intuition that the letters in a step should be concurrent and their order of appearance in the step should be incidental (depending on the chosen total strict order on the alphabet, which should be immaterial).

We thus have a sensible-looking definition, but does it work? And can the normalization function be defined in the same way as for standard traces? In the next section, we will show this to be the case, by describing our formalized development that includes proofs. The coherence conditions turn out to be instrumental in ensuring that we are indeed entitled to check coherence for contexts of steps in normal forms rather than contexts of individual letters.

In our example, the equivalence class of $abcd$ consists of 8 strings, with $(abd)(c)$ the normal form, as before. The equivalence class of $acbd$ is $\{acbd, acdb\}$, with $(a)(c)(bd)$ the normal form. The equivalence class of $adcb$ is $\{adcb, dacb\}$, with $(ad)(c)(b)$ the normal form.

The significance of coherence can be demonstrated already on this small example. If we had $bD_{[]}d$ instead of $bI_{[]}d$ (violating 2.$\Leftarrow$), then $(abd)(c)$ would cease to be a normal form under our chosen definition of normal forms. Instead, both $(ab)(cd)$ and $(ad)(b)(c)$ would be normal forms, although $abcd \sim abdc \sim adbc$, so normal forms would not be unique. (Our chosen normalization function would return $(ab)(cd)$ for $abcd$ and $abdc$, $(ad)(b)(c)$ for $adbc$.) If we had aD_db and aD_bd instead of aI_db and aI_bd (violating 2.$\Rightarrow$), then the strings $bdac$ and $dbac$ would only be equivalent to each other and without a normal form under our chosen definition of normal forms. (Our chosen normalization function would return $(abd)(c)$ for $bdac$ and $dbac$, which is not a normal form of these strings.)

3 Formalization

We will now present an overview of our Agda formalization of generalized traces, Foata normal forms and normalization. Showing Agda code, we will use some unofficial shortcuts to enhance readability; above all, we will typically omit implicit arguments in type signatures. While we will describe the definition of the normalization function in detail, for lemmas of the correctness proof, we will only give the type (the statement) and omit the proof.

3.1 Traces

We start our formalization from an alphabet `A` that we assume to be given, therefore we have defined it as a module parameter in Agda.

```
A : Set
```

The next component is that of a string (or word) over the alphabet `A`.

```
String  = List  A
String> = List> A
```

We work with two versions of strings—cons-lists and snoc-lists of letters—since, in some situations, we prefer to access the letters from the left end and in some situations from the right end. In fact, we will see that our normalization function manipulates a zipper: it traverses the input string (a cons-list) from the left and inserts every letter into the accumulated normal form, which is a snoc-list (of steps). `List>` is the type of snoc-lists. We have marked the usual list operations on snoc-lists with a trailing `>` to emphasize that they are for lists where the head element is on the right. Conversions between cons-lists and snoc-lists are denoted `c2s` and `s2c`.

We also assume a context-dependent independence relation on `A` that is both irreflexive and symmetric.

```
_I[_]_ : A → String> → A → Set
I-irr  : ∀ c → Irreflexive _I[ c ]_
I-sym  : ∀ c → Symmetric   _I[ c ]_
```

The underscores in mixfix operator identifiers mark the places where the arguments will go, in the order given in the type signature. `a I[ c ] b` means that, in the context `c` (which is a `String>`), the letters `a` and `b` are independent. We model the context as a snoc-list since most of the time we need to access the right end of the context. We define the context-dependent dependence relation `a D[ c ] b` as the negation (complement) of independence.

We say that two strings are one-step convertible if they differ only by the ordering of a pair of adjacent independent letters.

```
data _∼[_]_ : String → String> → String → Set where
  swap : a I[ c ++> c2s u ] b → (u ++ a ∷ b ∷ v) ∼[ c ] (u ++ b ∷ a ∷ v)
```

Note that the letters `a` and `b` are independent in the context `c` plus `u`, which is the common prefix of the two strings. We now take the reflexive-transitive closure of this one-step convertibility relation.

```
data _∼[_]*_ : String → String> → String → Set where
  refl*       : u ≅ v → u ∼[ c ]* v
  swap-trans* : u ∼[ c ] t → t ∼[ c ]* v → u ∼[ c ]* v
```

Note that the context `c` is fixed in the case of `swap-trans*`, which means that, in `u ∼[ c ]* v`, the strings that are actually considered equivalent are `c` plus `u` and `c` plus `v`, but the context `c` stays the same and no exchanges can be done in the context.

We are often working with strings in the empty context (we are looking at whole strings). For this case, we define the following abbreviations.

```
u ∼  v = u ∼[ [] ]  v
u ∼* v = u ∼[ [] ]* v
```

3.2 Normal Forms

We represent a Foata normal form as a snoc-list of steps and a step as a snoc-list of letters.

```
Step  = List> A
Foata = List> Step
```

These are the datatypes for "raw" steps and normal forms, we have not yet imposed the relevant well-formedness conditions. The embedding function is defined by flattening the two-layer snoc-list and converting the result to a cons-list.

```
emb : Foata → String
emb ss = s2c (concat> ss)
```

To define the well-formedness predicate for `Step`, we assume that our given alphabet `A` has a strict total order `_<_` defined on it. We also lift the independence relation from a binary relation on letters to a relation between a step and a letter.

```
_<_  : A → A → Set
sto< : StrictTotalOrder _<_

_■I[_]_ : Step → String> → A → Set
s ■I[ c ] a = All> (λ b → b I[ c ] a) s
```

`All> P xs` means that the predicate `P` holds on every element of the snoc-list `xs`.

```
data StepOk : String> → Step → Set where
  sngl : (a : A) → StepOk c [ a ]>
  snoc : StepOk c (s :> a') → (a : A) → a' < a → (s :> a') ■I[ c ] a
      → StepOk c (s :> a' :> a)
```

A well-formed step is either a singleton (in a context `c`) 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. Also, the old step and the new letter must be independent.

A small remark here is that `StepOk c s` is not necessarily a proposition (there can be more than one inhabitant of this type). If we were in a situation where we have `p, q : StepOk c s` and we had to show `p ≅ q`, we would have two options. Either we would have to assume that `a < b` and `a I[ c ] b` are propositions (for any `a`, `b` and `c`) or we would have to change the definition of `StepOk` to use propositional truncation when using `_<_` and `_I[_]_` (to have a normalization function, we must assume that both of these predicates are decidable, and this gives us effective truncation as a byproduct).

To define the well-formedness predicate for `Foata`, we first lift the dependence relation to a relation between a step and a letter (to describe when a letter is "supported" by a step). We also extend this to normal forms.

```
_♦D[_]_ : Step → String> → A → Set
s ♦D[ c ] a = Any> (λ b → b D[ c ] a) s
```

```
_♦D'_ : Foata → A → Set
[]       ♦D' a = ⊤
(ss ::> s) ♦D' a = s ♦D[ concat> ss ] a
```

`Any> P xs` is the type of existence of an element `x` in `xs` such that `P x` holds. ⊤ is the unit type (the trivially true proposition). Note the context used in the non-empty case of `♦D'`.

```
data FoataOk : Foata → Set where
  empty : FoataOk []
  step  : FoataOk ss → StepOk (concat> ss) s → All> (λ a → ss ♦D' a) s
       → FoataOk (ss ::> s)
```

A well-formed normal form can either be empty or it can consist of a well-formed normal form to which a step is added that must be well-formed in the context of this normal form. An additional condition for the result to be well-formed is that every letter in the new step is supported by the preceding normal form.

Similarly to `StepOk`, `FoataOk` is not a proposition. Even if we redefine `StepOk` so that it becomes a proposition, we still have the proof that the letters of the new step are supported by the normal form in the `step` case. If there are multiple candidates to support a letter in the new step, then we should give a canonical way to pick one of them. For example, we could always pick the rightmost dependent letter from the previous step as the support. Another option is again to use propositional truncation.

3.3 Normalization

We define a function `find>`, which given a decidable predicate and a list, splits the input list into two lists so that all of the elements in the second list (on the right) satisfy the predicate and the rightmost element in the first list violates the predicate, or the first list is empty. Note that the decidable predicate `P` here is context-dependent (it looks at both the head and tail of the snoc-list).

```
find> : (∀ xs x → Dec (P xs x)) → List> X → List> X × List> X
find> d? [] = [] , []
find> d? (xs ::> x) with d? xs x
find> d? (xs ::> x) | yes p = let ys , zs = find> d? xs in ys , zs ::> x
find> d? (xs ::> x) | no ¬p = xs ::> x , []
```

If we have a step and a letter that should be added to that step, then we can accomplish this by finding the right place for the new letter according to the ordering `_<_`. We assume decidability of `_<_`.

```
_<?_ : (a b : A) → Dec (a < b)

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

The properties of find> give us that all letters in rs are greater than a and the rightmost letter in ls is less than a or ls is empty. If s happened to be a well-formed step, then so are also both ls and rs. Given a normal form and a letter, we need to figure out the correct step for that letter and push it into that step. We assume decidability of _I[_]_.

```
_I[_]?_ : (a : A) → (c : String>) → (b : A) → Dec (a I[ c ] b)
```

```
insert : Foata → A → Foata
insert ss a with find> (λ xs x → x ■I[ concat> xs ]? a) ss
insert ss a | ls , []      = ls ::> [ a ]>
insert ss a | ls , rs ::> r = let s , rs' = first rs r in ls ::> push s a ++> rs'
```

Here, we use find> to split the normal form ss into two parts ls and rs so that a and all of the steps in rs are independent and a and the last step in ls are dependent or ls is empty. Note that while a and the steps in rs are independent, the contexts for those steps (and independence relations) are different. We pattern match on rs to decide whether to add a new step or not. If rs is empty, then ss already supports a and we must add a new step, otherwise, we push a to the leftmost step in rs. Here, _■I[_]?_ lifts _I[_]?_ from deciding independence of letters to deciding independence of a step and a letter.

With insert in place, we can now define a normalization function that traverses the string from the left to the right and inserts each letter into the correct position in the accumulated normal form.

```
norm' : Foata → String → Foata
norm' ss []      = ss
norm' ss (a :: t) = norm' (insert ss a) t
```

```
norm : String → Foata
norm t = norm' [] t
```

This is our normalization function, but it produces "raw" normal forms. We will now show that the functions we defined are "good" in the sense that they produce good output from good input. We begin with push.

```
pushOk : StepOk c s → (a : A) → s ■I[ c ] a → StepOk c (push s a)
```

Given that s is a well-formed step and s and a are independent, the result of push s a is also a well-formed step. To construct StepOk c (push s a), we need to show that the letters in push s a are ordered and independent. They are ordered since the letters in s are ordered and push uses find> to find a position in the list which respects the ordering. The letters are independent since the letters in s are independent and a and s are independent, which means that a and any subset of s (including the results of find>) are independent. The context c stays fixed inside a step.

We now continue with insert and show that it produces a well-formed normal form when given a well-formed normal form and a letter.

```
insertOk : FoataOk ss → (a : A) → FoataOk (insert ss a)
```

Compared to `pushOk`, things get much more involved here. Namely, when we insert a letter into a normal form, we are modifying a step somewhere in the middle of the normal form. As a result, the contexts for the invariants of the steps to the right of the modified step have changed.

For example, if we have a normal form `stuv` consisting of the steps `s`, `t`, `u` and `v`, then inserting `a` into `stuv` that will go into the step `t` changes the following. The letters in `u` must now be independent in the context `s(push t a)` instead of `st` and the letters in `v` must now be independent in the context `s(push t a)u`. Also, every letter of `v` must now have support from a letter in `u` in the context `s(push t a)` instead of `st`. One consequence of the consistency and coherence axioms is that when we do an `insert`, then we do not need to renormalize the part of the normal form for which the context changed, we can prove that the invariants still hold.

```
I-cons   : c ∼* c' → a I[ c2s c ] b → a I[ c2s c' ] b
I-co1    : a I[ c ] b → a I[ c ::> b ] d → d I[ c ::> a ] b → a I[ c ] d
I-co2-e : a I[ c ] b → b I[ c ] d → a I[ c ] d       → a I[ c ::> b ] d
I-co2-r : a I[ c ] b → b I[ c ] d → a I[ c ::> b ] d → a I[ c ] d
```

During an `insert`, we need to make sure that every step that we overtake with the new letter still satisfies the invariants, pairwise independence and support.

For pairwise independence, we need to repeatedly apply the `I-co2-e` axiom. To move a letter `b` past a step `s`, we must have `s ■I[ c ] b`. Since `s` is a (valid) step, it must be that the letters in `s` are pairwise independent. This means that, for every `a` and `d` in `s`, we have that `a I[ c ] b`, `d I[ c ] b` and `a I[ c ] d`, which gives `a I[ c ::> b ] d`. Consequently, the letters in `s` are still pairwise independent after extending the context with `b`.

For support, we think of the situation where we have a context (normal form) `c` and steps `s` and `s'` such that, for every `d` from `s'`, we have an `a` from `s` so that `a D[ c ] d`. This stops `d` from being moved into the earlier step `s`. To move `b` past `s` and `s'` into the context `c`, it must be that `s ■I[ c ] b` and `s' ■I[ c ++> s ] b`, which also gives that `a I[ c ] b` and `d I[ c ++> s] b`. This starts to resemble the contraposition of `I-co1`:

```
D-co1 : a I[ c ] b → a D[ c ] d → d I[ c ::> a ] b → a D[ c ::> b ] d
```

What we are missing from `D-co1` is `d I[ c ::> a ] b`, but we have `d I[ c ++> s ] b` and we also know that `a` is from `s`. We can prove the following lemma about extending the context of support:

```
PW : (X → X → Set) → List> X → Set
PW P []        = ⊤
PW P (xs ::> x) = PW P xs × All> (λ x' → P x' x) xs

♦D-ext-lem : d I[ c ++> s ] b → PW _I[ c ]_ (s ::> b)
           → s ♦D[ c ] d → s ■I[ c ::> b ] d → ⊥
```

This expresses the fact that, under suitable conditions, it cannot be that `d` is supported by (the step) `s` in the context `c` but is not supported by `s` in the extended context `c ::> b`. The first condition is that the letters `b` (that we add to the context) and `d` are independent (in the context `c` plus `s`). We also require that `s` can be viewed as a step (its letters are pairwise independent) and that the letter `b` and the step `s` are independent. The last two conditions are expressed by `PW _I[ c ]_ (s ::> b)`, which says that the predicate `_I[ c ]_` holds pairwise between the letters in `s ::> b`.

`s ♦D[ c ] d` and `s ■I[ c ::> b ] d` give us that there is a letter `a` in `s` such that `a D[ c ] d` and `a I[ c ::> b ] d`. We can apply contraposition of `I-co2-r` to get that both of `a I[ c ] b` and `d I[ c ] b` cannot hold. From `PW _I[ c ]_ (s ::> b)`, we get `a I[ c ] b`, which means that `d D[ c ] b` must hold. We derive a version of `I-co1` where one of the letters has been replaced with a step.

```
■I-co1 : s ■I[ c ] b → d I[ c ++> s ] b → s ■I[ c ::> b ] d → PW _I[ c ]_ s
       → d I[ c ] b
```

All of the arguments of this rule match the `I`-arguments of `♦D-ext-lem`, so we can derive `d I[ c ] b`. This cannot be since we previously derived `d D[ c ] b`.

`♦D-ext-lem` allows us to extend the context with a suitable letter, but we also need to "slide" the new letter to the position where `insert` would take it.

```
slide-step : PW _I[ c ]_ (s ::> b) → s2c (c ++> s ::> b) ∼* s2c (c ::> b ++> s)
```

This says that, if we have a letter `b` and step `s` that are independent, then we can slide `b` past `s` and the resulting string is equivalent to the one we started with. By repeatedly applying `slide-step`, we can move a letter past multiple steps and still preserve equivalence. This allows us to prove that `insert` is good. It follows that `norm` is also good:

```
normOk' : FoataOk ss → (t : String) → FoataOk (norm' ss t)
normOk  : (t : String) → FoataOk (norm t)
```

3.4 Properties

We will now proceed to soundness and completeness of our normalization algorithm. The first lemma is about `insert` and it says that inserting a letter into a normal form and then embedding into a string is equivalent to first embedding the normal form into a string and adding the letter to the end.

```
insert-lem : FoataOk ss → (a : A) → emb (insert ss a) ∼* (emb ss ++ [ a ])
```

This lemma is essentially `slide-step` lifted to the case of multiple steps and defined in terms of `insert`. This gives us the necessary tools to prove the existence of normal forms, which will then lead to completeness.

```
nf-exists' : FoataOk ss → (t : String) → emb (norm' ss t) ∼* (emb ss ++ t)
nf-exists  : (t : String) → emb (norm t) ∼* t

completeness : norm u ≅ norm v → u ∼* v
```

To prove soundness, we first show that our normalization operation commutes for independent letters. We start with `push`.

```
push-commutes : StepOk c s → a I[ c ++> s ] b → s ■I[ c ] a → s ■I[ c ] b
              → push (push s a) b ≅ push (push s b) a
```

This says that, if we have a well-formed step s and two letters a and b that are independent and that fit into s (a, b and s are independent), then it does not matter in which order we push them into the step. Since we have a strict total order on the alphabet and the letters in a valid step must be ordered, there is only one way to put a and b into s such that the result is ordered. We continue with a similar lemma about insert.

```
insert-commutes : FoataOk ss → a I[ concat> ss ] b
                → insert (insert ss a) b ≅ insert (insert ss b) a
```

This says that, if we have a normal form ss and two letters that are independent in the context of that normal form, then it does not matter in which order we insert them. We prove this by case analysis on the steps that a and b go to.

If both of these letters are supported by ss, then we must add a new step and show that it does not matter whether we add a new step by a singleton a or b. It may be that one of the letters fits into the existing steps and the other does not, in which case we have to show that, even if we add a new step, the other letter will still go past it (as the letters are independent) and that we need to add a new step even if we insert the other letter before. The last option is that both of them fit into the existing steps. If both of them go to the same step, then we apply push-commutes. Otherwise, we need to show that a letter goes to the same step whether we insert the other letter or not. We make use of some lemmas to consider an insert operation as a push. For example:

```
insert-last : ss ♦D' a → s ■I[ concat> ss ] a
            → insert (ss ::> s) a ≅ ss ::> push s a
```

Here we have that a is supported by ss, which means that insert cannot take this letter any further. We also have that a can be pushed into s since they are independent. This gives us that, in this situation, insert (ss ::> s) a is the same as ss ::> push s a. These tools allow us to prove that insert commutes. The fact that norm' commutes follows immediately.

```
norm'-commutes : FoataOk ss → a I[ concat> ss ] b
               → norm' ss (a :: b :: []) ≅ norm' ss (b :: a :: [])
```

We are now ready to prove soundness of the normalization algorithm. We first show that norm' is sound for strings that are one-step convertible.

```
sound∼ : FoataOk ss → t ∼[ concat> ss ] t' → norm' ss t ≅ norm' ss t'
```

It is important to note that t and t' are convertible in the possibly non-empty context concat> ss. The proof basically splits t and t' into pieces, so that we can focus on the pair of letters that make the two strings different and apply norm'-commutes. A useful lemma at this point is norm'-append that exposes the compositional nature of our normalization algorithm.

```
norm'-append : (t t' : String) → norm' ss (t ++ t') ≅ norm' (norm' ss t) t'
```

We lift `sound∼` to its reflexive-transitive closure. This also gives that `norm` is sound.

```
sound* : FoataOk ss → t ∼[ concat> ss ]* t' → norm' ss t ≅ norm' ss t'
soundness : t ∼* t' → norm t ≅ norm t'
```

We can now decide equivalence of two strings by first normalizing the strings and then checking if the normal forms are equal. If they are, then `completeness` says that the strings are equivalent, and, if the normal forms are not equal, then `soundness` says that the strings cannot be equivalent. To do this, we assume decidable equality on the alphabet.

```
_≅?_ : (a b : A) → Dec (a ≅ b)
equivalent? : (u v : String) → Dec (u ∼* v)
```

We also have that normal forms are stable, meaning that, if we (re-)normalize the embedding of a normal form, then we get back the same normal form.

```
stability : FoataOk ss → norm (emb ss) ≅ ss
```

We also get uniqueness of normal forms, meaning that, if we have two normal forms whose embeddings are equivalent, and thus come from the same equivalence class, then they must be equal.

```
nf-unique : FoataOk ss → FoataOk ss' → emb ss ∼* emb ss' → ss ≅ ss'
```

4 Example: Local Reads in TSO

Here we will give a small example where generalized traces are needed to describe the behaviour of a concurrent system reasonably precisely. The example comes from shared-variable concurrency in a system with write buffers which corresponds to the Total Store Order (TSO) relaxed memory model from the SPARC hierarchy [16].

The machine that we are going to model consists of processors and shared memory where each processor has local registers and a single write buffer. A program is a list of read and write instructions. The execution of a write instruction generates two events. First, the write is added to the buffer (the main event). Later, it is flushed to memory (the shadow event). The processor sees the memory "through" the buffer: if there are any writes to the variable x in the buffer, then the processor sees the value of the last write to x as the value of x, otherwise, it sees the value currently in memory.

We can represent program executions on this machine as words over the alphabet of events. We can also consider a dependence relation on this alphabet. Two events from different processors are dependent when they access the same memory location and at least one of them is a write. Two events from the same processor are dependent, if they are both main events (we respect the program order) or both shadow events (the buffer is *first-in-first-out*) or they are a corresponding pair of a main and a shadow event (a shadow event cannot happen before its cause, the main event).

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]`	

Our first scenario considers the execution $acc'ba'$, which has the same meaning as $acbc'a'$. This is because, in both of these executions, b happens before a', so there is a write in the buffer and b reads its value from that. Whether c' happens before or after b, does not matter. It seems reasonable in this situation to say that b and c' are independent.

The second scenario considers the execution $aca'bc'$, which has a different meaning than $aca'c'b$. This is because b happens after a' and thus b reads its value from memory since the buffer is empty. In the first execution, b reads the value written by a', and, in the other execution, it reads the value written by c'. Here, we would like to say that b and c' are dependent.

If we were to model this program using standard traces, we would have to set b and c' to be dependent; otherwise, the two executions of the second scenario would be considered equivalent, but they have different behaviour in the sense of taking the same initial state to different final states. This forced choice, however, distinguishes the two executions of the first scenario that behave the same way. This is an imprecision, we have more equivalence classes than desirable. Generalized traces allow us to say that b and c' are dependent only if there are no writes to x by the first processor in the buffer. This is information that can be read off the given context. The context should not contain a without also a' somewhere to its right, so, for example, the empty context and aa' make b and c' dependent, but ac does not.

5 Related Work

Traces were introduced into concurrency theory by Mazurkiewicz [11], but they originate from the enumerative combinatorics work by Cartier and Foata [3]. In particular, Foata normalization is from that work. Foata normalization is described in many of the standard expositions of trace theory, e.g., [1,5].

Generalizing traces for context-dependent independence has been considered by several authors, but with different well-behavedness conditions on independence. Sassone et al. [15] introduced context-dependent independence as we have considered it in this paper. Katz and Peled [9] introduced state-dependent independence, considering a coherence condition that in our setting would amount to aI_ub and $bI_{ua}c$ implying (aI_uc iff $aI_{ub}c$). This condition is equivalent to the conjunction of conditions 1. and 2.$\Rightarrow$ of Sassone et al. Droste [6], in a work on concurrent automata, again with state-dependent independence, required what would in our setting amount to aI_ub, bI_uc, $aI_{ub}c$ implying $aI_{uc}b$, $bI_{ua}c$, aI_uc, i.e., condition 2.$\Leftarrow$ and a little more. Hoogers et al. [8] 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 and use of representatives in model-checking, originally proposed by Godefroid [7] and Peled [14], are in wide use. Dynamic partial-order reduction for stateless model-checking of relaxed memory concurrent programs in particular has been considered by Abdulla et al. [2] and Zhang et al. [17]. In our own previous work [10], we used Foata normal forms for generalized traces for generating representative executions of all four memory models of the SPARC hierarchy.

Chou and Peled [4] have formalized standard Mazurkiewicz traces in the context of formally verifying a partial-order reduction technique in HOL. Owens et al. [13], who formalized TSO in HOL4, pioneered formalization of semantic accounts of relaxed memory models with proof assistants.

6 Conclusion and Future Work

We believe it to be important to exercise care when choosing the semantic domain for behaviors for a class of concurrent systems. Descriptions of behaviors in terms of an apparently more involved abstraction can sometimes be more precise, yet still analyzable with less effort. In this paper, we presented certified Foata normalization of generalized Mazurkiewicz traces. The example from Sect. 4 demonstrates that standard Mazurkiewicz traces are not flexible enough and generalized traces can lead to fewer equivalence classes. That is good in any situation where one needs to exhaustively check an equivalence-invariant property on all equivalence classes.

In this paper, we looked at Foata normal forms. Another well-known normal form of traces is the lexicographic normal form. It would be interesting to see how this works with generalized traces. We also wonder whether something similar is possible for yet more flexible notions such as various specializations of pomsets. We are currently working on a generic framework for semantics and analysis for relaxed memory concurrency that has normal-forms based dynamic partial-order reduction built into its core and thus only deals with normal forms.

Acknowledgements. This work was supported by the ERDF funded Estonian national CoE project EXCITE and the Estonian Ministry of Education and Research institutional research grant IUT-3313.

References

1. Aalbersberg, I.J.J., Rozenberg, G.: Theory of traces. Theor. Comput. Sci. **60**(1), 1–82 (1988)
2. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28

3. Cartier, P., Foata, D.: Problemes combinatoires de commutation et réarrangements. LNM, vol. 85. Springer, Heidelberg (1969). https://doi.org/10.1007/BFb0079468
4. Chou, C.-T., Peled, D.: Formal verification of a partial-order reduction technique for model checking. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 241–257. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_48
5. Diekert, V., Métivier, T.: Partial commutation and traces. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages: Beyond Words, vol. 3, pp. 457–553. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6_8
6. Droste, M.: Concurrency, automata and domains. In: Paterson, M.S. (ed.) ICALP 1990. LNCS, vol. 443, pp. 195–208. Springer, Heidelberg (1990). https://doi.org/10.1007/BFb0032032
7. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 176–185. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0023731
8. Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: A trace semantics for Petri nets. Inf. Comput. **117**(1), 98–114 (1995)
9. Katz, S., Peled, D.: Defining conditional independence using collapses. Theoret. Comput. Sci. **101**(2), 337–359 (1995)
10. Maarand, H., Uustalu, T.: Generating representative executions. In: Vasconcelos, V.T., Haller, P. (eds.) Proceedings of 10th Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software, PLACES 2017. Electronic Processing Theoretical Computer Science, vol. 246, pp. 39–48. Open Publishing Association, Sydney (2017)
11. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI Report PB-78, Aarhus University (1977)
12. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04652-0_5
13. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
14. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_34
15. Sassone, V., Nielsen, M., Winskel, G.: Deterministic behavioural models for concurrency. In: Borzyszkowski, A.M., Sokołowski, S. (eds.) MFCS 1993. LNCS, vol. 711, pp. 682–692. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57182-5_59
16. SPARC International Inc.: The SPARC Architecture Manual, Version 9. Prentice Hall, Englewood Cliffs (1994). (Ed. by D.L. Weaver and T. Germond)
17. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Proceedings of 36th ACM SIGPLAN Conference on Principles of Language Design and Implementation, PLDI 2015, pp. 250–259. ACM, New York (2015)