Multi-tape Computing with Synchronous Relations

Christian Wurm and **Simon Petitjean**
{cwurm,petitjean}@phil.hhu.de

Abstract

We sketch an approach to encode relations of arbitrary arity as simple languages. Our main focus will be faithfulness of the encoding: we prove that with normal finite-state methods, it is impossible to properly encode the full class of rational (i.e. transducer recognizable) relations; however, there is a simple encoding for the synchronous rational relations. We present this encoding and show how standard finite-state methods can be used with this encoding, that is, arbitrary operations on relations can be encoded as operations on the code. Finally we sketch an implementation using an existing library (FOMA).

1 Introduction

There is no need to list the merits and advantages of finite-state methods. However, there are some drawbacks, which are partly due to intrinsic properties of rational (i.e. transducer recognizable) relations in general, partly due to restrictions in current libraries:

1. In general, the rational relations are not closed under intersection and complement. Some libraries offer an approximate version of intersection, but this is not guaranteed to provide accurate results.

2. The question whether given two transducers, the relation recognized by one is a subset of (or equal to) the other is generally undecidable.

3. Standard libraries only support binary relations; we will in fact prove there is no way to faithfully (we will make this precise below) encode rational relations (binary or more than binary) as regular languages by finite-state means.

The goal of this article is to solve all these problems with one step: restrict our attention to synchronous rational relations. This solves problem 1. because this class actually forms a Boolean algebra; it follows that 2. is also solved, as this way, it is easy to decide the inclusion/equivalence problem. We can also solve problem 3., because we can faithfully encode arbitrary n-ary synchronous relations as simple languages. This encoding preserves recognizability (by finite-state automata) and allows to faithfully encode standard operations on relations via (different) standard operations on the code (we will define and explain this in more detail below). As the only drawback, we have to renounce to the additional expressive power of rational relations over synchronous relations; but as it was argued in [9], there might be little actual need for this in linguistic applications.

While from a theoretical point of view, we think there is little to object to our approach, there are large practical obstacles: firstly, synchronous relations are not as comfortable to specify as rational relations: for the latter, the representation of rational expressions is particularly practical and compelling (it is used i.e. by FOMA/XFST). We solve this problem here by defining a subclass of rational expressions which allows us to represent all and only synchronous relations, and we provide a program checking whether an expression is in this class. Secondly, we do not want to build a new program or library from scratch, but rather stick to existing ones. Therefore, we implement an interface which allows to define arbitrary n-ary synchronous relations, which are then encoded as languages and can be passed on to FOMA, a standard automaton library (see [5]). We have to add, however, that the implementation is still work in progress.

Hence we show that we can make use of synchronous relations in finite-state processing without really big obstacles, and this solves the three problems we mentioned above. The paper is structured as follows: next, we lay out the motivations we have sketched here more explicitly. Then we provide the basic definitions of classes of relations and operations on them. Then we sketch our approach and procedure to solve the problems. Finally, we discuss the implementation.

2 Motivation and previous work

We here outline the motivations of this paper more explicitly. It is well-known that rational relations are not closed under intersection (for proof, see [1]), and consequently not under complement. This is problematic for some applications: we might want to intersect two relations, each one implementing a certain set of constraints, where grammatical forms have to satisfy both. Also complementation might be useful, because this way we might specify ill-formed transformations and go from there to well-formed transformations. In fact, libraries as FOMA have a pseudo-intersection operation, but it is not guaranteed to yield a mathematically correct result.[1]

What is more problematic about the lack of intersection and complement is that without them, we cannot decide whether two transducers compute the same relation. This is problematic, because there are many different ways and methodologies to define the same relation. It would be nice to be able to see whether two relations are actually equivalent or one is more permissive than another (e.g. in comparing grammatical frameworks), but this is generally impossible with rational relations: proving equivalence of transducers is equivalent to giving a solution to the Post correspondence problem, which is undecidable (see [4]).

Finally, to our knowledge all existing libraries for transducers and rational relations only support binary relations. This is sufficient for most purposes in grammar engineering: we are interested in input/output relations, for example `word form/morphological analysis`; intermediate steps can and should be discarded. This is no longer true if we are interested in reconstruction of old languages from reported sound changes and shifts: here we often are interested in intermediate forms, and would need a form of "lossless composition" which preserves the intermediate steps. An alternative motivation is for example the desire to include phonology, prosody, semantics etc. into relations. For a nice motivation see [6, 7]. Once we have intersection and arbitrary relations, we also have the possibility to generalize composition by "matching" an arbitrary number of components instead of just one. As this allows to encode intersection, rational relations are obviously not closed under these operations, whereas synchronous rational relations are (for a general definition see below).

The most important previous work on this topic is [7], which also gives an encoding of string tuples of arbitrary size into strings. However, this encoding is somewhat problematic: it either does not preserve recognizability, that is, if an n-ary rational relation is encoded in this way, the code language is not a regular language, or it cannot be computed by a finite automaton (this follows from lemma 4). Moreover, it is not faithful for some operations (see below). This is one main motivation for our approach. Synchronous relations actually have already been used in some applications (see [8]), but in a different library (Vaucanson) and unrelated to the issue of introducing relations of arbitrary arity.

3 Preliminary definitions

There are two equivalent ways to define and represent rational relations, namely firstly as the relations recognized by finite-state transducers, and secondly as the relations denoted by rational expressions. A transducer computes an n-ary relation, if its transitions are labelled by $(\Sigma \cup \{\epsilon\})^n$. For reasons of space, we only formally define rational expressions, and presuppose knowledge of transducers. Fix an arbitrary alphabet Σ and an arbitrary arity n. Then for $a_1, ..., a_n \in \Sigma \cup \{\epsilon\}$, $(a_1, ..., a_n)$ is a rational expression (denoting $\{(a_1, ..., a_n)\}$); moreover if e, f rational expressions, then so are $e \cdot f$ (denoting componentwise concatenation of tuples), $e + f$ (denoting union) and e^* (denoting $1 + e + (e \cdot e) + ...$, where $1 = \{(\epsilon, ..., \epsilon)\}$. A relation is rational if it is denoted by some rational expression; we denote the class by **R**. The class of synchronous ratio-

[1]This intersection is based on identical automaton transitions, but for transducers, these are no longer unique, see [7].

nal relations (**SR**) is more complicated to define: put $\Sigma_\perp := \Sigma \cup \{\perp\}$, for $\perp \notin \Sigma$. The **convolution** of a tuple of strings $(w_1, ..., w_i) \in (\Sigma^*)^i$, written as $\otimes(w_1, ..., w_i)$, is in $((\Sigma_\perp)^*)^i$ and of length $max(\{|w_j| : 1 \leq j \leq i\})$, is defined as follows: the kth letter-tuple of $\otimes(w_1, ..., w_i)$ is $\langle \sigma_1, ..., \sigma_i \rangle$, where σ_j is the k-th letter of w_j provided that $k \leq |w_j|$, and $\perp$ otherwise. The convolution of a relation $R \subseteq (\Sigma^*)^i$ is defined by $\otimes R := \{\otimes(w_1, ..., w_i)$: $(w_1, ..., w_i) \in R\}$. A relation $R \in (\Sigma^*)^i$ is **synchronous regular**, if there is an ϵ-free finite-state automaton with transitions labelled by $(\Sigma_\perp)^i$ recognizing $\otimes R$.

Informally, **SR** are the relations computed by finite-state transducers which allow ϵ-transitions in a component only if no other letter is to follow in this component. There is also a characterization of **SR** in terms of first-order logic (see [2]), which is very convenient as it entails a lot of closure properties, in particular Boolean closure and closure under projection and cylindrification.

To have a more perspicuous notation, we sometimes use the **identity expansion** on languages: for $L \subseteq \Sigma^*$, we have $\mathsf{id}(L) \subseteq (\Sigma^*)^2$ with $\mathsf{id}(L) = \{(w, w) : w \in L\}$. We thus expand languages to their identity relations. For example FOMA represents every language L as $\mathsf{id}(L)$, hence the concept is well-known.

We use **projection** in a slightly different sense than usual, namely in the sense of *projecting away*. We define for $i \leq n$, $R \subseteq (\Sigma^*)^n$, $\pi_i(R) = \{(w_1, ..., w_{i-1}, w_{i+1}, ..., w_n) : (w_1, ..., w_n) \in R\}$. We urge the reader to keep in mind that for us, the ith projection is not *choosing* the ith component of tuples, but rather *discarding* it. On binary relations, these two notions of projection are obviously notational variants, but on relations of higher arity, our notion is much more powerful and convenient. The inverse of projections are **cylinders**: for $i \leq n + 1$, $R \subseteq (\Sigma^*)^n$, $C_i(R) = \{(w_1, ..., w_{i-1}, v, w_i, ..., w_n) : v \in \Sigma^*, (w_1, ..., w_n) \in R\}$. We can also define C_i as follows: for $R \in (\Sigma^*)^n$, $C_i(R) = \{(w_1, ..., w_{n+1}) : \pi_i(\{(w_1, ..., w_{n+1}\}) \subseteq R\}$; hence cylinders are maximal pre-images of projections. To avoid undefined terms, we introduce the following convention: if $R \subseteq (\Sigma^*)^n$, $i > n$, then $\pi_i(R) = R$, and if $i > n + 1$, then $C_i(R) = R$ (these two conventions are again necessarily parallel, otherwise C_i is no longer an inverse for π_i).

When we encode tuples as strings, relations as languages, the two notions are closely connected to homomorphisms and inverse homomorphisms. We define homomorphisms as follows: $h : (\Sigma^*)^n \to (T^*)^n$ is a homomorphism, if $h(w_1, ..., w_n) = (h(w_1), ..., h(w_n))$, and $h(aw) = h(a)h(w)$. h is a **relabelling**, if $a \in \Sigma$ implies $h(a) \in T$. For a homomorphism $h : \Sigma^* \to T^*$, $L \subseteq T^*$, we write $h^{-1}(L) = \{v \in \Sigma^* : h(v) \in L\}$ for the maximal pre-image. For clarity, we distinguish between a function f and its **graph**, where $graph(f) = \{(x, f(x)) : x \in dom(x)\}$. For every relabelling h, $graph(h)$ is a relation computed by a one-state synchronous transducer. For an arbitrary function $f : M \to N$, $X \subseteq M$, we write $f[X] = \{f(x) : x \in X\}$. Note that if $L \subseteq \Sigma^*$, then $h[L] = \pi_1(\mathsf{id}(L) \circ graph(h))$, where $\circ$ denotes relation composition.

A less well-known notion we need to explain is (generalized) **composition** with relations of arbitrary arity. We put, for $R \subseteq (\Sigma^*)^n$, $S \subseteq (T^*)^m$, $o \leq m, n$,

$$R \circ_o S = \{(w_1, ..., w_{n-o}, w_{n+1}, ..., w_{(n+m)-o}) : (w_1, ..., w_n) \in R, (w_{n-o}, ..., w_{(n+m)-o}) \in S\}.$$

Hence if $o = 0$, this becomes Cartesian product. Standard composition is $\circ_1$ (usually applied to binary relations), and we will usually write $\circ$ for $\circ_1$. We also define a relative we call **lossless composition**:

$$R \oplus_o S = \{(w_1, ..., w_{n-o}, ..., w_n, ..., w_{(n+m)-o}) : (w_1, ..., w_n) \in R, (w_{n-o}, ..., w_{(n+m)-o}) \in S\}.$$

Hence in this case, the "matching components" are not discarded, but rather kept, this way increasing the arity of the relation; if $m = n = o$, then this becomes intersection. As before, we usually write $\oplus$ instead of $\oplus_1$, which is the standard lossless composition as in [7].

A simple operation is inversion: $R^{-1} = \{(w_n, ..., w_1) : (w_1, ..., w_n) \in R\}$; hence we invert the order of tuples.

Two more standard operations for relations over strings are concatenation and Kleene star: assume $R, S \subseteq (\Sigma^*)^n$. Then $R \cdot S = \{(w_1 v_1, ..., w_n v_n) : (w_1, ..., w_n) \in R, (v_1, ..., v_n) \in S\}$; and $R^* = \{(\epsilon, ..., \epsilon)\} \cup R \cup R \cdot R \cup ....$

The following lemma summarizes the most important closure properties:

Lemma 1 *(**SR** closure properties)*

1. *If $R, S \subseteq (\Sigma^*)^n$, $R, S \in$ **SR**, then $(\Sigma^*)^n - R, S \cup R, S \cap R \in$ **SR**.*

2. *If* $R \subseteq (\Sigma^*)^n$, $R \in \mathbf{SR}$, *then* $\pi_i(R), C_i(R) \in \mathbf{SR}$.

3. *If* $R \subseteq (\Sigma^*)^m$, $S \subseteq (\Sigma^*)^n$, $o \leq m, n$, *then if* $R, S \in \mathbf{SR}$, *then* $R \circ_o S, R \oplus_o S \in \mathbf{SR}$.

4. *If* $R \in \mathbf{SR}$, *h a relabelling, then* $h[R] \in \mathbf{SR}$. *If h a homomorphism, then* $h[L] \in \mathbf{R}$.

These results are all well-known and easy to verify, given the logical characterization of **SR** (see [2],[9]): operations in 1. correspond to logical connectives; regarding 2., projection corresponds to existential quantification, cylindrification to introducing new (free) variables. Generalized (lossless) composition in 3. can be easily defined with operations in 1. and 2. (see section 4.2), and claim 4. is obvious.

What is problematic for **SR** is the lack of closure under concatenation and Kleene star: if $R, S \subseteq (\Sigma^*)^n$, $R, S \in \mathbf{SR}$, then $R \cdot S$ and R^* need not be in **SR**.

4 Encodings: faithfulness and completeness

4.1 Coding preliminaries

We say a map $\psi : (T^*)^n \to \Sigma^*$ encodes tuples in strings, if there are maps $\phi_1, ..., \phi_n$ such that for all $i : 1 \leq i \leq n$,

(1) $\phi_i(\psi(w_1, ..., w_n)) = w_i$

We write $\vec{x}$ to refer to tuples of strings. This means $\psi(\vec{x})$ contains all relevant information of all components, as they can be uniquely reconstructed. Note that this already entails a number of things, like: for $\vec{x} = (w_1, ..., w_n)$, $(\phi_1(\psi(\vec{x})), ..., \phi_n(\psi(\vec{x})) = \vec{x}$; this in turn shows that ψ is injective. We then say a language $L \subseteq \Sigma^*$ encodes a relation $R \subseteq (T^*)^n$, if $L = \psi[R]$. We define the complexity of the encoding via the complexity of the maps ϕ_i, as these are string-to-string and hence more convenient to handle (but in most natural cases, the complexity of ϕ_i determines that of ψ). An encoding is **simple**, if the maps ϕ_i are relabellings; it is **rational**, if the ϕ_i are finite-state computable. In general, we say a function f is rational if $graph(f)$ is a rational relation. Given an encoding $\psi : (T^*)^n \to \Sigma^*$, $code_\psi := \psi[(T^*)^n]$ denotes the set of code words.

The crucial question for encodings is whether we can faithfully transform operations on relations to operations on the code. Let $R_1, ..., R_n$ be relations, τ be an n-ary operation, ψ be an encoding. Then we say that the operation τ_ψ faithfully encodes τ, if

(2) $\psi(\tau(R_1, ..., R_n)) = \tau_\psi(\psi(R_1), ..., \psi(R_n))$

Hence we can simulate operations on relations by operations on codes. ψ being an injective function already entails that it is faithful for union, intersection and composition; complement easily follows, provided $code_\psi$ is a regular language. For us, the most relevant property of encodings is the following: we want that all standard finite-state methods (FSM) can be faithfully encoded as standard finite-state methods, so if τ is a standard FSM, then τ_ψ should be a standard FSM (though possibly a different one).

We now come to our simple encoding. It is based on tuple concatenation, but not componentwise: we defined $\cdot$ by $(a, b) \cdot (c, d) = (ac, bd)$, which results in a tuple of strings. To encode tuples as strings, we form $(a, b)(c, d)$ (without $\cdot$), which is not a tuple of strings, but rather a string of two tuples. We say that a string of tuples $\vec{x}_1 ... \vec{x}_i$ is a **factorization** of $\vec{y} \in (\Sigma^*)^n$, if 1. $\vec{x}_1, ..., \vec{x}_i \in (\Sigma \cup \epsilon)^n$, and $\vec{x}_1 \cdot ... \cdot \vec{x}_i = \vec{y}$. Factorizations are thus maximal decompositions, but importantly, factorizations are not unique, because there are factorizations such as $(a, \epsilon)(\epsilon, b)$ of (a, b). What is however unique are **synchronous factorizations**: a factorization $\vec{x}_1 .... \vec{x}_n$ is synchronous, if the following holds: if the jth letter of $\vec{x}_i$ is ϵ, then for all $m : i \leq m \leq n$, the jth letter of $\vec{x}_m$ is ϵ. Hence synchronous factorizations simply embody the synchronicity condition of synchronous relations, and synchronous factorizations correspond to convolutions, except that the latter are relations, the former are words – which is a crucial difference!

Note: all relations over strings have a unique synchronous factorization, regardless of whether they are synchronous, rational, or even computable. Hence we can conceive of

$$synfact : (\Sigma^*)^n \to ((\Sigma \cup \{\epsilon\})^n)^*$$

as a function, which for every tuple gives its synchronous factorization. This is actually a simple encoding in our sense, where ϕ_i is defined by the relabelling $h(a_1, ..., a_n) = a_i$. Keep in mind however that in factorizations, $(a_1, ..., a_n)$ is treated as an arbitrary letter, just as b. The following result is most important for $synfact$:

Lemma 2 *Assume $R \subseteq (\Sigma^*)^n$. Then R is synchronous rational if and only if synfact$[R]$ is a regular language.*

Proof. The proof is actually straightforward: just interpret a synchronous transducer recognizing a relation as recognizing a language, then you recognize the factorization of the relation, and vice versa. So the transducer remains the same, just its interpretation changes. □

This lemma is also very useful in order to prove that a relation is not synchronous. For example, take the relation $R_1 = (b,\epsilon)^* \cdot (a,a)^*$. In its synchronous factorization, we have three letters, namely (a,a), (b,a) and (a,ϵ), which for simplicity we write c,d,e. Then the synchronous factorization has the form $\mathit{synfact}(R_1) = \{d^m c^n e^o : m+n = n+o\}$. If we intersect this with d^*e^*, then we obtain $\{d^m e^m : m \in \mathbb{N}\}$, which is well-known to be not regular, and hence R_1 is not synchronous. We now provide a useful lemma showing how the maps ϕ_i relate to ψ. We define

(3) $\langle\phi_1,...,\phi_n\rangle(w) = (\phi_1(w),...,\phi_n(w))$

We extend this to sets in the pointwise fashion.

Lemma 3 *Assume L is regular, and let $\phi_1,...,\phi_n$ be rational functions. Then $\langle\phi_1,...,\phi_n\rangle[L]$ is a rational relation.*

Proof. By adding an additional component to *graph*(ϕ_i), it is easy to see that $\{(w,\phi_i(w),w) : w \in \Sigma^*\}$ is a rational relation. Call this (ternary) relation R_{ϕ_i}. By closure under (simple) composition, we can construct $\mathsf{id}(L) \circ R_{\phi_1} \circ ... \circ R_{\phi_n}$, which is an $(n+2)$-ary relation of the form $\{(w,\phi_1(w),...,\phi_n(w),w) : w \in L\}$. By using projections π_1,π_{n+2} we obtain $\langle\phi_1,...,\phi_n\rangle[L]$, which is then still rational (by closure under projection). □

By closure under inversion, this means that if $\psi(R)$ is finite-state decodable, it is also finite-state encodable: there is an $(n+1)$-ary rational relation $\mathit{graph}(\psi) = \{(w_1,...,w_n,v) : \psi(w_1,...,w_n) = v\}$. Now we can state that there does not exist a rational encoding for rational relations:

Lemma 4 *There is no rational encoding $\psi : (\Sigma^*)^n \to T^*$ such that for all rational relations R, $\psi[R]$ is regular.*

Proof. Assume there is such an encoding ψ. An encoding is an injection, hence we have $\psi(R) \cap \psi(S) = \psi(R \cap S)$. Since the encodings $\psi(R)$, $\psi(S)$ are regular, so is $\psi(R) \cap \psi(S) = \psi(R \cap S)$. As ψ can be computed by a finite-state transducer, by closure under transduction and projection, $\langle\phi_1,...,\phi_n\rangle[\psi(R \cap S)] = \psi^{-1}[\psi(R \cap S)]$ is a rational relation. However, for injective ψ, $\psi^{-1}[\psi(R \cap S)] = R \cap S$, which is the intersection of two rational relations and in general $R \cap S$ might be not rational – contradiction. □

4.2 Faithfulness for standard operations

After fixing the encoding for relations, we here present the encoding of operations τ_ψ for some standard operations τ. Here it is important to keep in mind: whereas for defining relations, the "tuple structure" of $(a_1,...,a_n)$ is essential, if we consider languages/factorizations, then $(a_1,...,a_n)$ is just an arbitrary letter no different from b. For reasons of space, in the following table we write ψ for *synfact*; on the left, we present the operation on (synchronous) relations, on the right the corresponding operation on the code.

	τ (on relation)	τ_ψ (on language)
1.	$\psi(R \cup S)$	$\psi(R) \cup \psi(S)$
2.	$\psi(R \cap S)$	$\psi(R) \cap \psi(S)$
3.	$\psi(\overline{R})$	$\overline{\psi(R)} \cap \mathit{code}_\psi$
4.	$\psi(\pi_i(R))$	$h_i[\psi(R)]$, h_i a relabelling
5.	$\psi(C_i(R))$	$h_i^{-1}(\psi(R))$, h_i a relabelling
6.	$\psi(R \circ_1 S)$	$\pi_2(C_3(\psi(R)) \cap C_1(\psi(S)))$
7.	$\psi(R \oplus_1 S)$	$C_3(\psi(R)) \cap C_1(\psi(S))$
8.	$R \circ_i S$	generalize 6.
9.	$R \oplus_i S$	generalize 7.
10.	$\psi(R^{-1})$	$h[\psi(R)]$, h a relabelling.

These are the set-theoretic operations (in a wide sense). As regards π and C (projection and cylindrification), these are usually not considered standard operations on relations, but this is only because on binary relations, they do not make too much effect. For our codes/factorizations, they become homomorphisms/inverse homomorphisms: as our letters have the form $(a_1,...,a_n)$, π_i is simply the relabelling $h_i : ((\Sigma \cup \{\epsilon\})^n)^* \to ((\Sigma \cup \{\epsilon\})^{n-1})^*$, defined by $h_i(a_1,...,a_n) = h(a_1,...,a_{i-1},a_{i+1},...,a_n)$; cylindrification C_i becomes the corresponding inverse relabelling h_i^{-1}. We usually do not have homomorphisms directly in our libraries, but we can easily define the one-state synchronous transducers computing them, and obtain the results by composition:

(4) $h[L] = \pi_1(\mathsf{id}(L) \circ \mathit{graph}(h))$

(5) $h^{-1}[L] = \pi_2(\mathit{graph}(h) \circ \mathsf{id}(L))$

So in some way or other, we can easily encode these operations, and hence all these operations can be used without any restriction. What is more serious and problematic are concatenation and Kleene star.

4.3 Concatenation and Kleene star

Our encoding is *not* faithful for concatenation and Kleene star. This is because the concatenation of two synchronous factorizations is not necessarily a synchronous factorization, such as $(a, \epsilon)(\epsilon, a)$. This is related to the fact that synchronous relations are not closed under concatenation, and correspondingly not under Kleene star, so there is no remedy to this problem. Lack of closure under these operations is probably the biggest problem with synchronous relations. We will circumvent this problem by introducing a **category system** of expressions. We can conceive of rational expressions as grammars with just one category, where every combination of categories yields that same category as result. We distinguish three categories of rational expressions:

1. el, the equal-length expressions (all components have equal length)
2. ed, the ϵ-difference expressions, where shorter components are ϵ
3. bd, the bounded difference expressions
4. gd, where difference can be unbounded and shorter components need not be ϵ
5. $\perp$, the expressions which are no longer guaranteed to be synchronous

As we have said, these categories concern the syntactic form of expressions, not their denotation, for which it is just a heuristic. Fix an arbitrary arity; then we have the following syntactic rules:

- $(a_1, ..., a_n) \in el$, if $a_1 \neq \epsilon, ..., a_n \neq \epsilon$
- $(a, ..., a_n) \in bd$ and $(a, ..., a_n) \in ed$ if for some $i \in \{1, ..., n\}$, $a_i = \epsilon$

Note that the assignment is polymorphic, where type polymorphism is handled in the standard way (it can also be avoided by adding a type $ed \wedge bd$). The combinatorics are as follows (we use x as variable for arbitrary categories)

- $el \cdot el = el$
- $el \cdot ed = bd \cdot ed = gd$
- $el \cdot bd = bd \cdot el = bd \cdot bd = bd$
- $el \cdot gd = bd \cdot gd = gd$
- $gd \cdot x = ed \cdot x = \perp$
- $el^* = el$
- $ed^* = ed$
- $bd^* = gd^* = \perp$

Moreover, we define an order $el \leq bd, ed \leq gd \leq \perp$, and for $+$ denoting union, for expressions e, e' of type x, x', $e + e'$ has type $x \vee y$, that is, the join with respect to the order. We call the expressions of category el, bd, ed, gd the **synchronous rational expressions** (SR-expressions); this consequently forms a (proper) subset of the rational expressions We could also devise a more fine-grained and permissive system, but we do not present it for reasons of space. The important thing is the following:

Lemma 5 *Every synchronous rational expression denotes a synchronous rational relation.*

This is easy to prove; for the few critical cases, use the synchronization lemma from [3]. So with SR-expressions, we are on the safe side, though many expressions which do denote synchronous relations are excluded. Note that the problem whether an arbitrary rational expression denotes a synchronous relation is undecidable (see [3], proposition 5.5).

4.4 Completeness of the constructions

Here we prove that with extended SR-expressions (for definition see below), we can construct all and only the synchronous relations. Importantly, this is not to say that only extended synchronous rational expressions denote synchronous relations, but for every synchronous rational relation we can construct an extended SR-expression. Take an alphabet Σ with $|\Sigma| \geq 2$. Let $EL \subseteq (\Sigma^*)^2$ be the set of equal-length string pairs $\{(w, v) : |w| = |v|\}$, $pref \subseteq (\Sigma^*)^2$ the set of pairs $\{(w, wv) : w, v \in \Sigma^*\}$, and for $a \in \Sigma$, $R_a \subseteq \Sigma^*$ the set of strings $\{wa : w \in \Sigma^*\}$. It is easy to see that EL can be constructed as synchronous expression with category el, $pref$ with category $el \cdot ed = gd$, $R_a : a \in \Sigma$ with category el (as it is unary). The completeness of our construction follows from the crucial direction of Eilenberg's result in [2]:

Theorem 6 *(Eilenberg, Elgot, Shepherdson) Assume $|\Sigma| > 1$. Then every synchronous rational relation of arbitrary arity over Σ can be constructed from EL, $\underline{pref}$, $R_a : a \in \Sigma$ by the operations C_i, π_i, $\cup, \cap, \overline{[-]}$.*

Actually, this is slightly different from the original formulation, as we leave out the logic part and only consider the semantics; still our formulation easily follows from the main theorem of [2]. The proof of this statement is long and complicated, so we omit a sketch. We define **extended synchronous expressions** as follows:

- If e is a synchronous rational expression, then it is an extended synchronous rational expression.
- If e, f are extended synchronous rational expressions, then so are $\pi_i(e)$, $C_i(e)$, $e \cap f$, $\overline{e}$.

The interpretation of these expressions is straightforward, as constructors are interpreted as themselves. Now the previous lemmas 1,5 and theorem 6 have the following consequence:

Corollary 7 *A relation R is synchronous regular if and only if it is denoted by some extended synchronous regular expression.*

Of course, our approach so far is rather terse, and there is lots of syntactic sugar we can add; in particular, the operations of (generalized) composition and lossless composition can be added, as they are straightforwardly definable by generalized synchronous expressions. For reasons of space, we do not present this here, but make use of this in our implementation.

5 The procedure and implementation

Of course, it would be possible to construct a library for synchronous relations of arbitrary arity from scratch. This is however not necessary, as our results indicate: we can use a library which is able to handle regular languages and binary relations, and all we have to do is mediate the input:

$$\text{user} \Longleftrightarrow \text{interface} \Longleftrightarrow \text{existing FS-library}$$

Hence the user can interact with our interface, writing relations of arbitrary arity with synchronous rational expressions. The interface encodes them as terms which denote languages (or rather their identity expansion), which are then passed on to an existing library, in our case FOMA. Furthermore, every request by the user is again mediated. These requests can be of different nature:

1. assign the relation in question to a variable, and use it to construct a larger relation
2. check equivalence of two expressions (with or without variables), or emptiness of an expression.
3. check whether a word w is denoted by an expression, or give the output for a certain input, or print some set of tuples which are recognized

These three types of requests can be easily handled, and we quickly sketch the procedure. 1. is easily taken care of, as this is just a variable assignment. 2. emptiness is routinely checked in FOMA; in order to check inclusion $R \subseteq S$, we just have to construct $R \cap \overline{S}$ and check its emptiness. As regards 3., we can just use the standard method constructing the relation $\mathsf{id}(\{w\})$ and compose/intersect, and print the output or check emptiness.

The processing chain of the interface starts with the parsing of the user input. The language that we propose for describing relations is comparable to the one used for regular expressions in FOMA, except that the elementary units are tuples instead of atoms. Before being encoded, the abstract syntax tree resulting of the parsing must be checked, as only synchronous rational expressions will be encoded by the interface.

The checking uses a color system, where colors represent the level of threat to the synchronicity of the relation. Each node in the abstract syntax tree of the expression will be colored either in black, green, orange or red. We start by coloring all the leaves of the tree (the tuples), and then all the colors of the internal nodes of the tree (the operations) are determined depending on the color of their daughter nodes (the arguments of the operations). The checking process explores the tree bottom up until the root, so the whole relation, is given a color. After checking, we will consider that a relation is not synchronous only if its color is red. A tuple will be given the green color if it features an empty word. When a node represents the application of a Kleene star, the resulting color will be orange if its daughter node (its argument) is green, so if the Kleene star is applied to a term containing an empty word. The relation until now is still synchronous, but will not be anymore if anything is concatenated on the right. For this reason, a node is given the red color if it represents a concatenation and if its left daughter node is orange.

One problem still needs to be taken care of after the checking: empty words should not appear in a tuple, except if for all the other tuples concatenated to its right also feature the empty word at the same index. For example, $(a, \epsilon)(a, b)$ should be forbidden. However, $(a, b)(a, \epsilon)$ denotes the same relation, and can be obtained by a simple transformation of the first expression. Our implementation realizes this type of transformation, which we call ϵ-shifting. Whenever two tuples are concatenated, for each empty word in the left tuple, if the word at the corresponding index in the right tuple is not empty, then we swap them. We repeat this process until no ϵ can be shifted anymore.

After checking the expression and performing ϵ-shifting, the encoding can be done. The encoding that we target is actually very close to the string that we had before parsing, due to the similarity between our language and the one of FOMA. The essential difference is that in the target string, we need to make sure that tuples will be interpreted by FOMA as atoms.

Let us now go through the steps of the processing chain while looking at a concrete example. We consider the following input:

```
( (a, epsilon, b) (a, c, a) ) | (a, c, b)*
```

The abstract syntax tree produced by the parser is be as follows:

```
['union',['concat',[('a','epsilon','b')],
                   [('a','c','a')]  ],
         ['star',[('a','c','b')]  ] ]
```

The checking of this tree starts by giving colors to the tuples: (a,epsilon,b) is green because of the ϵ, the two others are black. The concatenation does not produce a red color here, as the left daughter node ([('a', 'epsilon', 'b')]) is not orange but green. It would be the case for example if the input was:

```
( (a, epsilon, b)* (a, c, a) ) | (a,c,b)*
```

In this case, the process would have been stopped after unsuccessful checking. The next step is the ϵ-shifting, which explores the tree looking for concatenations. Only one shift is performed, on:

```
['concat', [('a', 'epsilon', 'b')],
           [('a', 'c', 'a')] ]
```

The tree which we obtain after the ϵ-shifting is as follows:

```
['union',['concat',[('a','c','b')],
                   [('a','epsilon','a')]  ],
         ['star',[('a','c','b')]  ] ]
```

The encoding part basically does the opposite of what the parser did, unfolding the abstract syntax tree into a string:

```
((%['a'%,'c'%,'b'%] %['a'%,'epsilon'%,'a'%])
|( %['a'%,'c'%,'b'%] )*)
```

where % is the escape character allowing us to let FOMA consider tuples (here represented as lists) as a single atoms. Now that the output string was produced, the user can compute with FOMA an automaton for the relation. Our interface supports all the operations mentioned in this article, except for composition. This is due to the fact that our interface does not implement its own operations, but uses the ones provided by FOMA, following the translations given in section 4.2. Projection and cylindrification, which are necessary to compute composition, are not supported by FOMA, as they depend on our specific way to encode the tuples. Even though there is no trivial solution for their implementation because of this reason, possible workarounds would include the use of homomorphisms or the development of an extension to FOMA which provides ways to access and modify the elements in our tuples.

6 Conclusion

We have presented an approach to allow users to work with (synchronous) rational relations of arbitrary arity, with full Boolean closure properties and a decidable inclusion problem. Our approach is based on the desire to work with existing libraries, and we have done this by encoding arbitrary relations as simple languages. Our two main results are the following: firstly, an approach as ours cannot work with the full class of rational relations, because it is impossible to encode arbitrary rational relations as regular languages by finite-state means. On the other hand, we have sketched that it works very well with synchronous rational relations, for which only concatenation and star are problematic. The second main result is that we have presented a class of expressions which denotes all and only the synchronous rational expressions, which is not trivial, as the problem whether a rational expression denotes a synchronous relation is undecidable. From a practical point of view, we have provided a type checker for expressions and implemented the encoding. However, to provide a full user-interface, some work still needs to be done.

References

[1] Jean Berstel. *Transductions and Context-free Languages*. Teubner, Stuttgart, 1979.

[2] S. Eilenberg, C. C. Elgot, and J. C. Shepherdson. Sets recognized by n-tape automata. *Journal of Algebra*, 13:447–464, 1969.

[3] Christiane Frougny and Jacques Sakarovitch. Synchronized rational relations of finite and infinite words. *Theor. Comput. Sci.*, 108(1):45–82, 1993.

[4] T. V. Griffiths. The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *J. ACM*, 15(3):409–413, July 1968.

[5] Mans Hulden. Foma: a finite-state compiler and library. In Alex Lascarides, Claire Gardent, and Joakim Nivre, editors, *EACL 2009, 12th Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, Athens, Greece, March 30 - April 3, 2009*, pages 29–32. The Association for Computer Linguistics, 2009.

[6] Mans Hulden. Grammar design with multi-tape automata and composition. In Thomas Hanneforth and Christian Wurm, editors, *Proceedings of the 12th International Conference on Finite-State Methods and Natural Language Processing, FSMNLP 2015, Düsseldorf, Germany, June 22-24, 2015*. The Association for Computer Linguistics, 2015.

[7] Mans Hulden. Rewrite rule grammars with multitape automata. *Journal of Language Modelling*, To appear.

[8] Florian Lesaint. Synchronous relations in Vaucanson. Technical Report 0833, Laboratoire de Recherche et Développement de L'Epita, 2008.

[9] Christian Wurm and Younes Samih. Synchronous regular relations and morphological analysis. In Mark-Jan Nederhof, editor, *Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing, FSMNLP 2013, St. Andrews, Scotland, UK, July 15-17, 2013*, pages 35–38. The Association for Computer Linguistics, 2013.