Multi-tape Computing with Synchronous Relations

Christian Wurm & Simon Petitjean

University of Düsseldorf, Germany

Umea, 05.09.2017

Multi-tape Computing with Synchronous Relations

There are many motivations for using multi-tape transducers:

- We want to relate more than two aspects of a language: e.g. semantics, morphology, phonology, phonetics.
- We want to keep track of intermediate steps in composition of relations: e.g. in old language reconstruction.
- We want to relate more than two languages.
- ...

Multi-tape Computing with Synchronous Relations

However, multi-ary relations are not usually supported by standard libraries, and behave differently from binary relations in some ways.

Our solution

Our solution is to encode multi-ary relations as binary/unary relations.

However, in general, we cannot encode arbitrary rational (transducer recognizable) relations as unary relations (see below). But this is possible with the **synchronous rational relations** (**SR**).

Multi-tape Computing with Synchronous Relations

Synchronous rational relations are in a sense a largest subclass of the rational relation, which forms a Boolean algebra.
Hence:

- Closure under intersection, complement (contrary to rational relations)
- Consequently: decidability inclusion and equivalence of two relations (contrary to rational relations)
- Closure under generalized (lossless) composition, i.e. matching of one or more components, with and without cancelling out.

Multi-tape Computing with Synchronous Relations

Problem: Synchronous rational relations are inconvenient to use for the community:

- Rational expressions (as in FOMA [Hulden, 2009]) do not allow to characterize **SR**.
- Solution: we describe a class of expressions which exactly characterizes **SR**.

Multi-tape Computing with Synchronous Relations

Problem: Synchronous rational relations are inconvenient to use for the community:

- (Synchronous) multi-tape relations are not supported by standard libraries
- Solution: we implement an interface which translates **SR**-expressions to regular languages, faithfully encoding all operations. These can be handled by standard libraries.

Outline

Outline

Problems of rational relations

Rational relations

A relation is rational if it is denoted by some rational expression
Fix an arbitrary alphabet Σ and an arbitrary arity n

- for $a_1, \ldots, a_n \in \Sigma \cup \{\epsilon\}$, $(a_1, \ldots, a_n)$ is a rational expression (denoting $\{(a_1, \ldots, a_n)\}$)
- if e, f are rational expressions, then so is $e \cdot f$ (denoting componentwise concatenation of tuples),
- if e, f are rational expressions, then so is $e + f$ (denoting union)
- if e is a rational expressions, then so is e^* (denoting $1 + e + (e \cdot e) + \ldots$, where $1 = \{(\epsilon, \ldots, \epsilon)\}$

Problems of rational relations

Rational (transducer recognizable) relations are extremely useful in NLP. This is based on a number of properties:

- Closure under composition
- Closure under union
- Closure under concatenation and Kleene star

Each of these operations is very useful, because it allows to construct a complex relation by simpler ones by means of the operations. Closure ensures we still have finite-state transducers effectively computing the relation.

Problems of rational relations

Problems

Rational relations are not closed under intersection (for proof, see [Berstel, 1979]), and consequently not under complement.

- libraries as FOMA have a pseudo-intersection operation, but it is not guaranteed to yield a mathematically correct result
- without intersection and complement, we cannot decide whether two transducers compute the same relation.
- all existing libraries for transducers and rational relations only support binary relations

Outline

Synchronous rational relations

Convolution (tuple of strings)

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 $max(\{|w_j| : 1 \leq j \leq i\})$, 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.

Synchronous rational relations

Convolution (relation)

The convolution of a relation $R \subseteq (\Sigma^*)^i$ is defined by $\otimes R := \{\otimes(w_1, ..., w_i) \colon (w_1, ..., w_i) \in R\}$.

Synchronous regular relations

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 $\otimes R$.

Example

$((a, a) \cdot (a, \epsilon))^* \notin$ **SR**, because no ϵ-free transducer recognizes

$$\{(a^{2n}, a^n \perp^n) : n \in \mathbb{N}_0\}$$

Why Synchronous rational relations?

Largest natural subclass

The class **SR** is the largest known natural class smaller than the rational relations.

Advantages of **SR**

- **SR** has a number of important closure properties: composition, projection, cylindrification (see below)
- In particular, **SR** is a Boolean algebra, hence inclusion and equivalence are decidable!
- But: **SR** is not closed under concatenation and Kleene star!
- We will use the fact there is an interesting correlation between **SR** and the regular languages.

Synchronous rational relations: operations

Projection

We define for $i \leq n$, $R \subseteq (\Sigma^*)^n$,
$\pi_i(R) = \{(w_1, ..., w_{i-1}, w_{i+1}, ..., w_n) : (w_1, ..., w_n) \in R\}$

Cylindrification

For $i \leq n+1$, $R \subseteq (\Sigma^*)^n$,
$C_i(R) = \{(w_1, ..., w_{i-1}, v, w_i, ..., w_n) : v \in \Sigma^*, (w_1, ..., w_n) \in R\}$

Homomorphisms

$h : (\Sigma^*)^n \to (T^*)^n$ 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 $h(a) \in T$.

Synchronous rational relations: operations

Composition and generalized composition

These operations are not among the standard finite-state operations. But: together with Boolean operations, they allow to define

- Relation composition $((a, b) \circ (b, c) \mapsto (a, c))$
- Lossless relation composition $((a, b) \oplus (b, c) \mapsto (a, b, c))$
- Generalized composition of relations of higher arity (matching more than one component, e.g. $(a, b, c) \circ_2 (b, c, d) \mapsto (a, d)$)
- Same for lossless composition e.g. $(a, b, c) \oplus_2 (b, c, d) \mapsto (a, b, c, d)$)

Synchronous rational relations

Closure properties of **SR**

1. Boolean closure: f $R, S \subseteq (\Sigma^*)^n$, $R, S \in \mathbf{SR}$, then $(\Sigma^*)^n - R, S \cup R, S \cap R \in \mathbf{SR}$.
2. Projection/Cylindrification: If $R \subseteq (\Sigma^*)^n$, $R \in \mathbf{SR}$, then $\pi_i(R), C_i(R) \in \mathbf{SR}$.
3. Generalized (lossless) composition: 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. Relabelling: If $R \in \mathbf{SR}$, h a relabelling, then $h[R] \in \mathbf{SR}$. If h a homomorphism, then $h[L] \in \mathbf{R}$ (the rational relations).

Synchronous rational relations

Problem: concatenation and star

SR lacks closure under concatenation and Kleene star
if $R, S \subseteq (\Sigma^*)^n$, $R, S \in \textbf{SR}$, then $R \cdot S$ and R^* need not be in **SR**.

Example

$(a, a)^*$, $(b, \epsilon)^*$ and $((a, a) \cdot (a, \epsilon))$ are in **SR**, but

- $(b, \epsilon)^* \cdot (a, a)^* \notin \textbf{SR}$
- $((a, a) \cdot (a, \epsilon))^* \notin \textbf{SR}$

Interim summary

What we have showed

These properties allow us to use **SR** for multitape computing. However, the main problem remains: standard libraries do not support more than binary relations.

How we proceed

We will tackle this problem by encoding arbitrary synchronous rational relations as regular languages.

Outline

The encoding: synchronous factorizations

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

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

Faithfulness

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

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

This states that we can simulate operations on relations via operations on their code.

The encoding: synchronous factorizations

Our 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

$$(a, b)(c, d) \text{ (without } \cdot\text{)},$$

which is not a tuple of strings, but rather a *string of tuples*.

The encoding: synchronous factorizations

Factorization

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
2. $\vec{x}_1 \cdot ... \cdot \vec{x}_i = \vec{y}$.

Factorizations are not unique, consider factorizations as $(a, \epsilon)(\epsilon, b)$ of (a, b).

Synchronous factorization

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.

The synchronous factorization of a tuple is **unique**, hence we have a function $synfact(\vec{x})$

The encoding: synchronous factorizations

We generally put $f[X] := \{f(x) : x \in X\}$

Lemma

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

Example

$synfact[((a,a) \cdot (a,\epsilon))^*] = \{(a,a)^n(a,\epsilon)^n : n \in \mathbb{N}_0\}$, which is obviously not regular (isomorphic to $a^n b^n$!).

The encoding: synchronous factorizations

The previous lemma shows the tight relation between **SR** (of arbitary arity) and the regular languages. For the rational relations, we can show that no such encoding exists:

Lemma

There is no rational (i.e. finite-state computable) encoding

$$\psi : (\Sigma^*)^n \to T^*$$

such that for all rational relations R, $\psi[R]$ is regular.

This is the main motivation for using **SR**!

The encoding: synchronous factorizations

Here some faithful encodings of standard operations, given the encoding via synchronous factorizations.

Standard operations

	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.	$\psi(\overline{R})$	$\overline{\psi(R)} \cap code_\psi$
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.

The encoding: synchronous factorizations

Problem

Our encoding is *not* faithful for concatenation and Kleene star.
This follows from two facts:

Lemma

If we close the class of synchronous rational relations under concatenation and Kleene star, we obtain the rational relations.

And:

Lemma

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

The encoding: synchronous expressions

Still, we want to use concatenation and Kleene star in a restricted ("safe") fashion!

- Therefore, we devise a category system for expressions with $\cdot$ and $*$.
- Categories tell us, whether an expression can still be guaranteed to denote a synchronous relation, and
- for *every* synchronous rational relation, we have an expression of a "safe" category!

Note however that in general, it is undecidable whether a rational expression denotes a relation in **SR**!

The encoding: synchronous factorizations

We distinguish these categories of rational expressions:

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

The encoding: synchronous factorizations

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. If we extend these expressions with constructors for projection, cylindrification and Boolean operations, we obtain the following:

Lemma

(Soundness) Every extended synchronous rational expression denotes a synchronous rational relation.

Lemma

(Completeness) For every synchronous rational relation, there is an extended synchronous rational expression denoting it.

Outline

The implementation

The goal of our implementation is to be able to process multi-ary relations with a standard library, in a transparent way.

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

- We do not implement the standard operations, but use the ones of the library
- The language used for the input is as close as possible to the one of the library
- The input is type-checked, and encoded to be processed (or not) by the library

The implementation: example (type-checking)

The following input has to be ruled out by the type-checker:

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

$$(a, \varepsilon, b)^*(a, c, a)$$

The implementation: example (type-checking)

The following input has to be ruled out by the type-checker:

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

$$(a, \varepsilon, b)^*(a, c, a)$$

The implementation: example (type-checking)

The following input has to be ruled out by the type-checker:

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

$$(a, \varepsilon, b)^*(a, c, a)$$

The implementation: example (type-checking)

The following input has to be ruled out by the type-checker:

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

$$(a, \varepsilon, b)^*(a, c, a)$$

The implementation: example (type-checking)

The following input has to be ruled out by the type-checker:

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

$$(a, \varepsilon, b)^*(a, c, a)$$

The expression does not belong to the class, the process is stopped

The implementation: example (encoding)

For the following input, the type-checking is successful and the encoding can be given to the library

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

The implementation: example (encoding)

For the following input, the type-checking is successful and the encoding can be given to the library

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

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

should be forbidden, but an expression denoting the same language can be obtained by ε-shifting

The implementation: example (encoding)

For the following input, the type-checking is successful and the encoding can be given to the library

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

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

should be forbidden, but an expression denoting the same language can be obtained by ε-shifting

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

The implementation: example (encoding)

For the following input, the type-checking is successful and the encoding can be given to the library

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

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

should be forbidden, but an expression denoting the same language can be obtained by ε-shifting

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

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

Conclusion

- Idea: allow users to work with (synchronous) rational relations of arbitrary arity.
- Method: encoding arbitrary relations as simple languages to work with existing libraries.
- Problem: cannot work with the full class of rational relations.
- Synchronous rational relations: only concatenation and star are problematic.
- We presented a class of expressions which denotes all and only the synchronous rational expressions.
- Practical side: type checker for expressions and implementation of faithful encoding of operations.

Thank you!

Berstel, J. (1979).
Transductions and Context-free Languages.
Teubner, Stuttgart.

Hulden, M. (2009).
Foma: a finite-state compiler and library.
In Lascarides, A., Gardent, C., and Nivre, J., 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.