Shuffle Expressions and Words with Nested Data

Henrik Björklund[1,⋆] and Mikołaj Bojańczyk[2,⋆⋆]

[1] University of Dortmund
[2] Warsaw University

Abstract. In this paper, we develop a theory that studies words with nested data values with the help of shuffle expressions. We study two cases, which we call "ordered" and "unordered". In the unordered case, we show that emptiness (of the two related problems) is decidable. In the ordered case, we prove undecidability. As a proof vehicle for the latter, we introduce the notion of higher-order multicounter automata.

1 Introduction

A data word is a word where each position, in addition to its finite alphabet label, carries a *data value* from an infinite domain. Recent times have seen a flurry of research on data languages, motivated chiefly by applications in XML databases and parameterized verification; see, e.g., [7,9,2,4,11,3,1]. One of the main results is that satisfiability for first-order logic with two variables is decidable over data words, as long as the only operation allowed on the data values is equality testing [2]. The same paper also demonstrates a close connection between data languages, *shuffle expressions*, and *multicounter automata*:

1. Multicounter automata. These are nondeterministic automata with many counters, which can be incremented and decremented, but zero tests are only allowed at the end of the word.
2. Shuffle expressions. These are regular expressions extended with intersection and the shuffle operation.
3. Two-variable data languages. These are properties of data words that can be expressed in certain two-variable logics.

The connection between multicounter automata (or Petri nets) and shuffle expressions was discovered in [5], while the connection between the first two and data languages was discovered in [2].

In this paper, we develop and investigate extensions of the above. We focus on *nested* data values and shuffle expressions. There are two principal motivations.

– When data values are only used to induce an equivalence relation on the positions, such as in the logics mentioned above, one of the chief applications is parameterized verification. A number of processes run in parallel, and in

⋆ Supported by the Deutsche Forschungsgemeinschaft Grant SCHW678/3-1.
⋆⋆ Supported by Polish government grant no. N206 008 32/0810.

L. Kučera and A. Kučera (Eds.): MFCS 2007, LNCS 4708, pp. 750–761, 2007.

the resulting sequence of actions, the individual actions are annotated by the unique id of the process that performed them. This data word can be used to check global properties (by looking at the whole string) and local properties (by considering the sequence of actions of a single process).

To model a system where processes can have subprocesses, and so on, we would want a data word with *nested* data values: each action carries the id of the subprocess which performed it as well as the id of the process that spawned the subprocess, and so on.

– In [5], nesting of the shuffle operation was not considered. This runs contrary to the unrestricted nesting of other operations in regular expressions, and begs the question what happens when unrestricted nesting of shuffles is allowed. We discover that the resulting languages are actually intimately related to languages with nested data. We also discover that this leads to undecidability; and decidability can be recovered only after adding a form of commutativity to the shuffle operation.

We note that our notion of shuffle expressions is different than the one used in some of the literature–see, e.g.,[6]–since we consider four different shuffle operators and allow intersection with regular languages.

Our main topic is logics over words with nested data. We study two two-variable fragments of first order, one with order and one without. The first one is shown to be undecidable, while the second is decidable. Nested shuffle expressions are the main proof vehicle, and an object of independent study. For instance, the expressions we consider capture the language:

$$a^{n_1}\#\cdots\#a^{n_k}\#b^{m_1}\#\cdots\#b^{m_k}\# \;\;:\;\; n_1,\ldots,n_k \text{ is a permutation of } m_1,\ldots,m_k$$

It is our opinion that the two concepts—logic with nested data and shuffle expressions—shed light on each other in a useful cross-fertilization.

Due to space restrictions, most proofs have been omitted, and will appear in the full version of the paper.

2 A Logic for Words with Nested Data

Let A be a finite alphabet and Δ an infinite set, whose elements will be called *data values*. For $k \in \mathbb{N}$, a *word with k layers of data* is a word where every position, apart from a *label* in A, has k labels $d_1,\ldots,d_k \in \Delta$. The label d_i is called the *i-th data value* of the position. Therefore, such a word is an element $w \in (A \times \Delta^k)^*$. In w, the data values can be seen as inducing k equivalence relations $\sim_1,\ldots,\sim_k$ on the positions of w. That is, two positions are related by $\sim_i$ if they agree on the i-th data value.

We are interested in the data values only insofar as equality is concerned. Therefore, the relations $\sim_1,\ldots,\sim_k$ supply all the information required about a word; and could be used as an alternative definition. Adding more structure—such as a linear order—to the data values leads very quickly to undecidable decision problems, already with one layer, and even for very weak formalisms.

A word with k layers of data is said to have *nested data* if for each $i = 2, \ldots, k$ the relation $\sim_i$ is a *refinement* of $\sim_{i-1}$. In other words, if two positions agree on value i, then they also agree on value $i-1$. For the rest of this section, we fix k and only talk about words and languages with k layers of nested data. Instead of "word with k nested layers of data", we will just write "word with nested data".

In the spirit of Büchi's sequential calculus, we will use logic to express properties of words with nested data. In this setting, a word with nested data is treated as a model for logic, with the logical quantifiers ranging over word positions. The data values are accessed via the relations $\sim_i$. For instance, the formula

$$\forall x \forall y \quad (x \sim_2 y \Rightarrow x \sim_1 y) \wedge \cdots \wedge (x \sim_k y \Rightarrow x \sim_{k-1} y)$$

states that the data values are nested. (This formula is a tautology in our setting, where only models representing words with nested data values are considered.) Each label a from the finite label set A can be accessed by a unary relation; for instance $\forall x\ a(x)$ is true in words where all positions have label a. The linear order on word positions can be accessed via the relation $<$. For instance,

$$\forall x \quad (\forall y\ y \leq x) \Rightarrow (\forall y < x\ y \not\sim_1 x)$$

says that the last position has a different first (and consequently, all others as well) data value than than the other positions. We also use the successor relation $x = y + 1$. Although it can be defined in terms of the order, a third variable z is required, which is too much for the two-variable fragments we considered. We mainly consider satisfiability: given a formula, determine if there is a nested data word that satisfies it. This problem is undecidable in general, but by restricting the formulas, we can obtain decidable fragments.

Satisfiability is undecidable already for the following fragments, see [2]:

- There is only one layer of data. The formulas can use three variables; but the order on word positions can be accessed only via $x = y + 1$ and not $<$.
- There are two layers of data, but these are not necessarily nested. The formulas can use only two variables, $x = y + 1$, $x = y + 2$ and not $<$.

The largest known decidable fragment was presented in [2]:

- There is only one layer of data. The formulas can use only two variables, and the positions can be accessed by both $x = y + 1$ and $<$.

We want to generalize the above result to words with multiple layers of nested data. The result from [2] fails already for two layers:

Theorem 2.1. *With two layers of nested data, satisfiability is undecidable for two-variable first-order logic with the relations $x = y + 1$ and $x < y$.*

Proof (Sketch). We encode computations of a two-counter machine with zero tests. Consider words with two layers of nested data, where the labels are $A = \{start, end, inc, dec\}$ and the positions are labeled according to the regular language $(start(inc{+}dec)^* end)^*$. This is easy to express in our two-variable logic.

Next, we express that each block of the form $start(inc + dec)^* end$ corresponds to a single data value on layer 1:

$$\begin{array}{cc} \forall x\ start(x) \Rightarrow \forall y < x\ x \not\sim_1 y & \forall x\ end(x) \Rightarrow \forall y > x\ x \not\sim_1 y \\ \forall x \exists y \geq x\ x \sim_1 y \wedge end(y) & \forall x \exists y \leq x\ x \sim_1 y \wedge start(y) \end{array} \tag{1}$$

Finally, the we can use equivalence classes (sets of positions with the same data value) on layer two to ensure that the operations *inc* and *dec* are balanced.

Consider a word with nested data that satisfies the above properties. This word can be seen as a computation of a one counter machine without states, where *inc* corresponds to a counter increment, *dec* corresponds to a counter decrement, while $start, end$ correspond to zero tests.

To get two counters instead of one, we expand the label alphabet from A to $\{1, 2\} \times A$. The regular language we use is now

$$start_1 start_2 (inc_1 + inc_2 + dec_1 + dec_2 + end_1 start_1 + end_2 start_2)^* end_2 end_1.$$

The rest of the construction also generalizes, by duplicating each formula, once for each counters, and adding consistency constraints.

Even with the above result, however, not all hope is lost. Satisfiability is decidable if we lose the order $<$, even with arbitrarily many layers of data.

Definition 1. *$FO^2(+1, \sim_1, \ldots, \sim_k)$ is the fragment of first-order logic that uses only the two variables x and y and the following predicates.*

$a(x)$	*x has label a, where a is a label from A*
$y = x + 1$	*y is the position directly to the right of x*
$x \sim_i y$	*x and y have the same layer i data value, with $i \in \{1, \ldots, k\}$*

Theorem 2.2. *Over words with nested data, satisfiability is decidable for $FO^2(+1, \sim_1, \ldots, \sim_k)$.*

The above result is a consequence of Theorem 3.4, which says that formulas of the logic can be compiled into a type of shuffle expression, and Theorem 5.2, which says that emptiness is decidable for these shuffle expressions.

3 Shuffle Expressions

Recall that a word $w \in A^*$ is called a *shuffle* of words $w_1, \cdots, w_m \in A^*$ if the positions of w can be colored using m colors so that the positions with color $i \in \{1, \ldots, m\}$, when read from left to right, form the word w_i. If $K \subseteq A^*$ is a (possibly infinite) set of words, then $shuffle(K) \subseteq A^*$ is defined as

$$\{w : w \text{ is a shuffle of some } w_1, \ldots, w_m \in K, \text{ for some } m \in \mathbb{N}\}.$$

Note that the words $w_1, \ldots, w_m$ above may include repetitions. For instance, $shuffle(\{a\})$ contains all words a^*. Just as finite automata are connected to regular expressions, multicounter automata are connected to shuffle expressions. This is witnessed by the following result, essentially due to [5]:

Theorem 3.1. *The following language classes are equal, modulo morphisms:*

1. *Languages recognized by multicounter automata;*
2. *Languages of the form $L \cap \mathit{shuffle}(K)$, where L, K are regular.*

The qualification "modulo morphisms" means that any language from class 1 is a morphic image of a language in class 2. (Class 2 is simply contained in class 1, without need for morphisms.) The point of the morphism is to erase bookkeeping, such as annotations with accepting runs.

We can now ask what we get if we add intersection and *shuffle* to regular expressions. The answer is that we get more than we want:

Theorem 3.2. *If regular expressions are extended with shuffle and intersection, all recursively enumerable languages can be defined modulo morphisms.*

In particular, emptiness is undecidable for such extended regular expressions. Disallowing intersection trivializes the emptiness problem — which is the problem we are most interested in here — since $\mathit{shuffle}(K)$ is nonempty if and only if K is. Theorem 3.2 follows directly from Theorems 4.1 and 4.2 below.

In this paper, however, we are most interested in decidability results, especially for logics with data values. It turns out that decidability can be recovered, if we consider a weaker form of shuffling, which is partly commutative.

3.1 Cutting and Combining

To express our modification of the shuffle operation, it is most convenient to decompose $\mathit{shuffle}(L)$ into two operations:

$$\mathit{shuffle}(L) = \mathit{combine}(\mathit{cut}(L)) \ . \tag{2}$$

The first *cut* operation sends a set of words $L \subseteq A^*$ to the set of traces obtained by cutting a word from L into pieces:

$$\mathit{cut}(L) = \{w_1|\cdots|w_k : w_1 \cdots w_k \in L\} \subseteq (A^*)^*.$$

In the above a *trace* is a sequence of finite words, which is written as $w_1|\cdots|w_k$, with $|$ separating consecutive words, called *segments*. Traces are denoted by θ or σ, and will be heavily used later on.

The second operation is called *combine*, and it sends a set of traces $L \subseteq (A^*)^*$ to the set of words that can be combined from these traces:

$$\mathit{combine}(L) = \{w : w \text{ is a combination of } \theta_1, \ldots, \theta_m \in L \ \} \subseteq A^*.$$

By saying that w is a *combination* of traces $\theta_1, \ldots, \theta_m$ we mean that positions of w can be colored with m colors, so that for each color $i = 1, \ldots, m$, the positions with color i give the trace θ_i. In the trace $\theta_i = w_1|\cdots|w_n$, the segments $w_1, \ldots, w_n$ correspond to *maximal* subwords of w that are assigned color i. Consider the following example.

$$\begin{array}{c} a\ b\ c\ c\ c\ b\ a\ c \\ 1\ 1\ 2\ 1\ 3\ 1\ 2\ 2 \end{array} \qquad \begin{array}{l} \theta_1 = ab|c|b \\ \theta_2 = c|ac \\ \theta_3 = c \end{array}$$

By the maximality requirement, the trace θ_2 cannot be replaced by $c|a|c$ Given the above definitions of *cut* and *combine*, it should be fairly clear that equation (2) holds. However, by tweaking the definitions of *cut* and *combine*, we will arrive at variants of the shuffle operation that are decidable and, more importantly, relevant to our investigation of nested data values.

The first modification gives us more control on the way words from $L \subseteq A^*$ are cut into traces. Let $K \subseteq A^*$ be a language. We define

$$cut_K(L) = \{w_1|\cdots|w_k : w_1 \cdots w_k \in L, w_1, \ldots, w_k \in K\} \subseteq (A^*)^*.$$

In other words, words from L are cut into traces where each segment belongs to K. Setting $K = A^*$ allows us to recover the standard cut operation, so cut_K is a generalization of *cut*. We only consider the case when K is regular.

The second modification concerns the operation *combine*. An *unordered trace* is a multiset of words, i.e. a trace where the order of segments is not important (however, the ordering of letters inside the segments is). The operation $ucombine(L)$ treats the set of traces L as unordered traces:

$$ucombine(L) = \{w : w \text{ is an unordered combination of } \theta_1, \ldots, \theta_m \in L\ \} \subseteq A^*.$$

In the above, w is an *unordered combination* of $\theta_1, \ldots, \theta_m$ if w is a combination of traces $\sigma_1, \ldots, \sigma_m$, such that σ_i is obtained from θ_i by rearranging the order of segments. Thus abc is an unordered combination of traces $\theta_1 = c|a$ and $\theta_2 = b$.

3.2 Four Kinds of Shuffle Expressions

From the above, we obtain four variants of the shuffle operation:

- Shuffle: $shuffle(L) = combine(cut(L))$.
- Controlled shuffle: $cshuffle_K(L) = combine(cut_K(L))$.
- Unordered shuffle: $ushuffle(L) = ucombine(cut(L))$.
- Unordered controlled shuffle: $ucshuffle_K(L) = ucombine(cut_K(L))$.

Each such operation gives rise to its own flavor of extended regular expressions. We will investigate and compare these flavors with respect to decidability and expressive power.

Definition 3.3. Controlled shuffle expressions (CSE) *denote languages obtained by nesting the following operations:*

- *Standard regular expression operations: single letters $a \in A$, the empty word ϵ, concatenation, union and Kleene star.*
- *Intersection with regular languages.*
- *Controlled shuffle $cshuffle_K(L)$, where L is defined by a CSE, but K is a regular word language.*
- *Images under morphisms $f : A^* \to B^*$.*

Shuffle expressions (SE), unordered shuffle expressions (USE) *and* unordered controlled shuffle expressions (UCSE) *are defined analogously, by replacing the type of shuffle operation allowed. All types of operations can be freely nested.*

The point of adding morphic images is to have a form of nondeterministic guessing in the expressions (and therefore more power). This will be illustrated in the following example. Note also that morphic images are necessary due to adding the intersection and shuffle operations; in standard regular expressions the projection operation does not add any power and can be eliminated.

Example 1. In the shuffle operation $\mathit{shuffle}(L)$, we have no control over the number of times the language L is used. In this example we show that we can enforce that it is used an even number of times. Let then $L \subseteq A^*$ be defined by an SE. The idea is to expand the alphabet A with a new symbol *start*; each word from L will be prefixed by this symbol. Consider then the expression:

$$K = \mathit{start} \cdot L \ .$$

If we now take the expression $\mathit{shuffle}(K)$, we can use the marker *start* to see how many times K was used. By intersecting with the regular language "even number of occurrences of *start*", we can make sure that it was used an even number of times. Finally, the markers can be removed using the erasing morphism $f : (A \cup \{\mathit{start}\})^* \to A^*$ defined by $f(\mathit{start}) = \epsilon$ and $f(a) = a$ for $a \in A$.

Example 2. Unordered shuffling is enough to express some counting properties: $\mathit{ushuffle}(ab)$ describes words in $(a + b)^*$ with the same number of a's and b's. Using intersection with the regular language a^*b^*, we get the language $\{a^n b^n\}$.

Example 3. Using the same idea as in the previous example, we can also get the language $L = \{a^n \# b^n \#\}$. Consider now the following expression:

$$(a^*\#)^*(b^*\#)^* \cap \mathit{ucshuffle}_K(L) \qquad \text{where } K = a^*\# + b^*\#$$

This expression defines the set of words

$$a^{n_1} \# \cdots \# a^{n_k} \# b^{m_1} \# \cdots \# b^{m_k} \#,$$

such that $n_1, \ldots, n_k$ is a permutation of $m_1, \ldots, m_k$.

3.3 From Logic to Shuffle Expressions

In this section we state the reduction of satisfiability for $FO^2(+1, \sim_1, \ldots, \sim_k)$ to the emptiness problem for unordered controlled shuffle expressions.

Theorem 3.4. *For every $FO^2(+1, \sim_1, \ldots, \sim_k)$-formula ϕ, a UCSE r can be effectively computed such that the language of r is non-empty if and only if ϕ is satisfiable.*

Proof (Sketch). We can assume that ϕ is in *data normal form*; a normal form very similar to the one used in [2]. This means that ϕ is a set of conjuncts, where each conjunct is fairly simple. The conjuncts express properties of classes by referring to types. When talking of types, these conjuncts only use the expressions "at

least one node", "has a node", and "at most one node". Thus, the only relevant information about a class string (where the class can be w.r.t. any of the k equivalence relations), is the number of times each type appears. Furthermore, if a type appears at least twice, the exact number of times is irrelevant.

If we have a footprint mapping $f : T \to \{0, 1, 2\}$, where T is the set of types appearing in ϕ, that tells us if a type appears 0, 1, or 2 or more times, we know everything we need about the class string. We can easily compute the set F of those footprints that are allowed by the conjuncts from ϕ. If we construct, for each $f \in F$, a shuffle expression that accepts exactly those strings that have footprint f, we can combine these expressions (using the + operator) into one that accepts all correct class strings. Using the shuffle operator on this expression gives an expression whose language is such that every word can be extended with (level k) data values in such a fashion that the conjuncts of ϕ that use $\sim_k$ are satisfied.

The idea is to use this construction inductively, starting with level k, until, after using k shuffle operations, all conjuncts of ϕ are taken care of.

The corresondence between r and ϕ is actually stronger: r contains words obtained from models of ϕ by erasing data values. We do not know if the converse translation—from expressions to logic—can be done; possibly the expressions are strictly stronger than the logic. A similar reduction, from logic with order to CSE is possible, but the proof is omitted.

4 Ordered Shuffle Expressions

The following theorem relates CSE, SE and higher-order multicounter automata. The latter, to our best knowledge, are a new model.

Theorem 4.1. *The following language classes are equal:*

1. *Languages defined by controlled shuffle expressions (CSE);*
2. *Languages defined by shuffle expressions (SE);*
3. *Languages defined by higher-order multicounter automata;*
4. *Recursively enumerable languages.*

We define higher-order multicounter automata in Section 4.1, and prove their Turing completeness. The rest of the proof of Theorem 4.1 is omitted.

4.1 Higher-Order Multicounter Automata

A multiset over A is a function $m : A \to \mathbb{N}$. We only consider finite multisets here, where all but a finite number of elements in A are assigned 0. We also consider higher-level multisets (which are also multisets). A level 1 multiset over A is a finite multiset over A. A level $k + 1$ multiset over A is a finite multiset of level k multisets over A.

A level k multicounter automaton is defined as follows. It has a state space Q, an input alphabet Σ, and a multiset alphabet A. All of these are finite. The

automaton reads an input word $w \in \Sigma^*$ from left to right. At each moment, its memory is a tuple $(q, m_1, m_2, \ldots, m_k)$, where q is one of the states in Q, and each m_i is a level i multiset over A, possibly undefined $\perp$. (We distinguish an empty multiset $\emptyset$ from an undefined one $\perp$.) The initial configuration is $(q_I, \perp, \perp, \ldots, \perp)$, where q_I is some designated initial state.

There is a finite set of transition rules, which say how the machine can modify its memory upon reading an input symbol (or doing an ϵ-transition). Each such transition rule is of the form: when in state q and upon reading the label $a \in \Sigma \cup \{\epsilon\}$, assume state p and do counter operation x. The counter operations are:

new_i: Change m_i from $\perp$ to $\emptyset$.
inc_a: Add $a \in A$ to the level 1 multiset m_1.
dec_a: Remove $a \in A$ from the level 1 multiset m_1.
$store_i$: Add m_i to the level $i+1$ multiset m_{i+1}; then set m_i to $\perp$.
$load_i$: Remove nondeterministically some element m from m_{i+1} and store it in m_i. This transition is enabled only when $m_1, \ldots, m_i$ are all $\perp$.

We use $Counterops_k$ to denote the possible counter operations in a level k automaton. Note here that the automaton knows which m_i are undefined, since this is controlled by transitions new_i and $store_i$. On the other hand, the automaton does not know if a defined multiset m_i is empty, or not.

What is the accepting condition? We say a level k multiset is *hereditarily empty* if it is empty, or it consists only of hereditarily empty level $k-1$ multisets. The automaton accepts if $m_1, \ldots, m_k$ are all hereditarily empty multisets in all memory cells; and the control state belongs to a designated accepting set.

The above definition is similar, but not identical, to the notion of *nested Petri nets* from, e.g., [8].

Here we show that the machines are Turing complete, already on level 2.

Theorem 4.2. *Level* 2 *multicounter automata recognize all recursively enumerable languages.*

Proof
We show that a level 2 multicounter automaton can simulate a two-counter machine with zero tests. Since the latter type of machine is capable of recognizing all recursively enumerable languages, the statement follows.

A configuration of the two-counter machine, where counter 1 has value i and counter 2 has value j, will be represented by the following level 2 multiset:

$$\{\{\underbrace{x, \ldots, x}_{i \text{ times}}, a\}, \{\underbrace{x, \ldots, x}_{j \text{ times}}, b\}, \underbrace{\emptyset, \ldots, \emptyset}_{k \text{ times}}\} \ . \tag{3}$$

The occurrences $\emptyset$ are used for bookkeeping; the number k will correspond to the number of zero-tests that have been carried out in the run leading to this configuration. A configuration as above is called *proper*. Our automaton will have the property that improper configurations always lead to improper configurations; furthermore, a failed zero-test will lead to an improper configuration.

We now show how to represent the operations of the simulated machine:

- Zero test on counter 1. We do the following sequence of operations:

$$load_1 \quad dec_a \quad store_1 \quad new_1 \quad inc_a \quad store_1 \ .$$

If the configuration was improper, it will remain so. If it was proper, the level 2 from (3) multiset will become:

$$\{\{a\}, \{\underbrace{x, \ldots, x}_{j \text{ times}}, b\}, \underbrace{\emptyset, \ldots, \emptyset}_{k \text{ times}}, \{\underbrace{x, \ldots, x}_{i \text{ times}}\}\} \ .$$

If i was not 0, the above configuration will be improper.

- Increment on counter 1. We do the following sequence of operations:

$$load_1 \quad dec_a \quad inc_x \quad inc_a \quad store_1 \ .$$

A decrement is done the same way.

- The operations on counter 2 are as above, except b is used instead of a.

One can easily see that the automaton can reach a proper configuration as in (3) if and only if the simulated two-counter machine could have counter values (i, j). Furthermore, the simulating machine can test (once, at the end of its run), if it has reached a proper configuration of the form:

$$\{\{a\}, \{b\}, \underbrace{\emptyset, \ldots, \emptyset}_{k \text{ times}}\} \ .$$

This is done by $load_1 \quad dec_a \quad store_1 \quad load_1 \quad dec_b \quad store_1$ and testing if all memory cells are hereditarily empty. □

5 Unordered Shuffle Expressions

In this section, we state the decidability of the emptiness problems for unordered shuffle expressions, controlled (UCSE) or not (USE). Since $ushuffle(L) = ucshuffle_{A^*}(L)$ if A is the alphabet of L, it is clear that USE is a special case of UCSE. Nevertheless, we chose to state the following independently:

Theorem 5.1. *Emptiness for unordered shuffle expressions is decidable.*

The reason is that the proof is considerably less involved than for the controlled case. It uses a reduction to finite word automata equipped with a Presburger counting condition.

As stated in the introduction, the main goal of this paper is to show decidability of satisfiability for the 2-variable logic from Definition 1 over words with nested data. Theorem 3.4 shows that this problem reduces to emptiness for UCSE. We are now ready to complete the proof of Theorem 2.2, by stating the main combinatorial result of the paper:

Theorem 5.2. *Emptiness for unordered controlled shuffle expressions is decidable.*

The proof is rather involved, and is based on a study of the Parikh images [10] of languages defined by UCSE. Due to space limitiations, we can can only present here a very brief outline of the key ideas.

The main technical result is the following Parikh-type theorem:

Theorem 5.3. *The Parikh image of a language defined by a UCSE is semilinear.*

Furthermore, since the semilinear set is effectively obtained, it can be tested for emptiness in a decision procedure for emptiness of UCSE, hence Theorem 5.2 follows. The proof of Theorem 5.3 is by induction, and only one step is nontrivial: when the expression is of the form

$$ucshuffle_M(L) \cap K, \tag{4}$$

where L is defined by a UCSE and K, M are regular languages.

What follows is a very informal description of some of the ideas used in showing that the Parikh image of the language above is semilinear. We first remind the reader how the language (4) is defined. We begin with traces $\theta_1, \ldots, \theta_n$ that are obtained from cutting words from L into segments from M. In other words, each trace $\theta_i \in (A^*)^*$ must belong to $cut_M(L)$. Then, the segments of these traces are rearranged and combined to get a word in the regular language K.

The basic idea for computing the Parikh image of (4) is as follows. To θ we assign two vectors: its Parikh image $\pi(\theta) \in \mathbb{N}^A$; and another vector $\rho(\theta) \in \mathbb{N}^B$, called the footprint of θ. The idea behind the footprint is that it contains information on the way θ can be combined with other traces to get a word in K. By using the induction assumption of Theorem 5.3, we can show that these two vectors are related in a semilinear way, i.e. the following vector set is semilinear:

$$Y = \{(\pi(\theta), \rho(\theta)) : \theta \in cut_M(L)\} \subseteq \mathbb{N}^{A \cup B}.$$

Using the mappings π and ρ, the job of calculating the Parikh image of (4) can be split into two phases. In the first phase, the question whether or not traces $\theta_1, \ldots, \theta_n$ can be combined into a word from K is rephrased as a condition (*) on the footprints $\rho(\theta_1), \ldots, \rho(\theta_n)$. In the second phase, the semilinear set Y is used to go from the from the footprints to the Parikh images. More precisely, we show that the following set is semilinear:

$$\{\pi(\theta_1) + \cdots + \pi(\theta_n) : \theta_1, \ldots, \theta_n \text{ are traces such that } \rho(\theta_1), \ldots, \rho(\theta_n) \text{ satisfy condition (*)}\}$$

This concludes, since the above set is the Parikh image of (4). Note that in the above, we do not need to quantify over the traces θ_i, since it is enough to verify that two vectors $\pi(\theta_i)$ and $\rho(\theta_i)$ satisfy the semilinear property Y.

We conclude by summarizing the expressive power of the expressions:

$$\text{USE} \subsetneq \text{UCSE} \subsetneq \text{SE} = \text{CSE}.$$

The strictness of the first inequality is not shown due to lack of space. The second inequality follows by undecidability of SE, while the equality was mentioned in Theorem 4.1.

References

1. Björklund, H., Schwentick, T.: On notions of regularity for data languages. In: FCT'07 (to appear, 2007)
2. Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: LICS'06, pp. 7–16 (2006)
3. Bouyer, P., Petit, A., Thérien, D.: An algebraic approach to data languages and timed languages. Information and Computation 182(2), 137–162 (2003)
4. Demri, S., Lazić, R.: LTL with the freeze quantifier and register automata. In: LICS'06, pp. 17–26 (2006)
5. Gischer, J.: Shuffle languages, petri nets, and context-sensitive grammars. Communications of the ACM 24(9), 597–605 (1981)
6. Jedrzejowicz, J., Szepietowski, A.: Shuffle languages are in P. TCS 250, 31–53 (2001)
7. Kaminski, M., Francez, N.: Finite-memory automata. TCS 132(2), 329–363 (1994)
8. Lomazova, I.A., Schnoebelen, P.: Some decidability results for nested petri nets. In: PSI'99, pp. 208–220 (2000)
9. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM transactions on computational logic 15(3), 403–435 (2004)
10. Parikh, R.: On context-free languages. Journal of the ACM , 570–581 (1966)
11. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)