Visibly pushdown transducers

Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, Jean-Marc Talbot

Visibly Pushdown Transducers

Emmanuel Filiot[a], Jean-François Raskin[a], Pierre-Alain Reynier[b], Frédéric Servais[a,c], Jean-Marc Talbot[b]

[a] *Université Libre de Bruxelles*
[b] *Laboratoire d'Informatique Fondamentale de Marseille, Université Aix-Marseille-CNRS*
[c] *Haute École de Bruxelles*

Abstract

Visibly pushdown transducers (VPT) extend visibly pushdown automata (VPA) with outputs. They read nested words, i.e. finite words on a structured alphabet partitioned into call, return and internal symbols, and produce output words, which are not necessarily nested. As for VPA, the behavior of the stack is synchronized with the types of input symbols: on reading a call symbol, exactly one stack symbol is pushed onto the stack; on reading a return symbol, exactly one stack symbol is popped from the stack; and the stack is untouched when reading internal symbols. Along their transitions VPT can append words on their output tape, whose concatenation define the resulting output word. Therefore VPT define transformations from nested words to words and can be seen as a subclass of pushdown transducers. In this paper, we study the algorithmic, closure and expressiveness properties of VPT.

Keywords: Visbily Pushdown Automata, Transducers, Functionality

1. Introduction

General context

Over the years, XML [9] has become a standard formalism for data exchange over the internet. XML data, also known as XML documents, are essentially a textual representation of information. Compared to the traditional relational databases, the main feature of XML is its weak structure that is embodies by XML tags. Problems for XML databases are rather similar to the ones of relational databases. To cite a few, validation ("does an XML document has a correct shape wrt to some declaration ?"), querying ("what are the information associated with the tag $\langle a \rangle$ in the XML document ?"), computation of views/transformations ("compute from the XML document representing a library a document enumerating for each author the list of books he has contributed to") can be considered.

This XML formalism induced the development of practical tools [57, 26, 13, 8, 42] as well as theoretical studies on formal tools for manipulating XML documents. This latter has led to define and study new formal objects or to consider well-known ones with a new perspective, impacting significantly research in formal language theory.

More concretly, an XML document is built from information (or data) that are grouped by means of tags using a well-parenthesing policy. As for parenthesis, there exist some opening (e.g. $\langle article \rangle$) as well as closing tags (e.g. $\langle /article \rangle$). Such a document is examplified on Fig. 1.

As pairs of tags can be nested one into the other, this nesting induces a tree structure of the document, often refered to as the DOM (Document Object Model) tree. Hence, tree automata are good candidates as theoretical foundations of XML processing tools. Indeed, some new tree automata models, motivated by XML, have been developped and investigated [44, 45]. An alternative is to consider the document in

Email addresses: efiliot@ulb.ac.be (Emmanuel Filiot), jraskin@ulb.ac.be (Jean-François Raskin), pierre-alain.reynier@lif.univ-mrs.fr (Pierre-Alain Reynier), frederic.servais@gmail.com (Frédéric Servais), jean-marc.talbot@lif.univ-mrs.fr (Jean-Marc Talbot)

its textual form, as a sequence of symbols together with an additional nesting structure that associates an opening tag with its closing matched tag. This point of view has been adopted in the XML manipulating library SAX [42]. Words together with this implicit nesting given by tags are a particular case of nested words. Automata, named visibly pushdown automata, running on these objects have first been introduced in [4].

Our contribution to this research area is the design and the study of a formal mathematical tool to process nested words; the present paper proposes for this purpose visibly pushdown transducers (VPT), that extend visibly pushdown automata (VPA) with outputs, as an abstract model of machines for defining transformations of nested words.

Nested Words

Nested words are finite sequences of symbols equipped with a nesting structure. The term *nesting structure* refers to a structure that is organized on the basis of layers some of which are contained into the others. Special symbols, namely *call* and *return* symbols, can be used to add a nesting structure, through a call/return matching relation. Other (untyped) symbols are called *internal* symbols. Formally, a nested word is defined as a word over a structured alphabet. In a nested word, some call symbols (resp. return symbols) may be pending if they do not match any return symbol (resp. call symbol). Nested words without any pending symbols are called *well-nested words*.

Nested words arise naturally in many applications as models of data which are both linearly and hierarchically ordered. Executions of structured programs as well as XML documents are examples of data that are naturally modelled as nested words [5]. The sequence of calls and returns (resp. opening and closing tags), adds nesting structures to program executions and XML data respectively. More generally, any tree-structured data can be linearized into a nested word by considering a depth-first left-to-right traversal of the tree.

As an example, consider the XML document of Fig.1 representing an article with its (simplified) structure. Opening and closing tags correspond to call and return symbols, and the unstructured data are words over internal symbols. More precisely, the alphabet is partitioned into call symbols $\Sigma_c = \{\langle article\rangle, \langle section\rangle, \langle title\rangle, \langle content\rangle\}$, return symbols $\Sigma_r = \{\langle /article\rangle, \langle /section\rangle, \langle /title\rangle, \langle /content\rangle\}$ and internal symbols $\Sigma_i = \{a, \ldots, z, A, \ldots, Z\}$.

Automata for nested words

Alur and Madhusudan introduced *visibly pushdown automata* (VPA) as a pushdown machine to define languages of nested words [5], called visibly pushdown languages. More precisely, VPA operate on a structured alphabet, partitioned into call, return and internal symbols. When reading a call symbol the automaton must push one symbol on its stack, when reading a return symbol it must pop a symbol on top of its stack, and when reading an internal symbol it cannot touch its stack. As nested words can be naturally viewed as trees, visibly pushdown automata inherit all the good properties of tree automata [14]. Most notably, they are closed under all Boolean operations, and all the classical decision problems such as emptiness, universality, finiteness are decidable. This is in contrast to general pushdown automata that define context-free languages, for which it is well-known that most interesting problems are undecidable. The fundamental reason why VPA enjoy good properties is that the stack behavior, *i.e.* whether it pops, pushes or does not change, is driven by the input symbol. Therefore, all the stacks on a same input have the same height (we say that the stacks are *synchronized*). Although VPA are equivalent to tree automata, they run on words and operate from left to right. This perspective has led to many applications, such as new logics for nested words [2], streaming XML validation and queries [1, 24, 36, 35], and program analysis [31, 30, 3].

Transducers for nested words

Transducers are machines that define transformations (also called *transductions*) of input to output words. Fig. 1 gives two examples of transformations. The first transformation R_0 takes a nested word representing an XML document and outputs an (unstructured) text version of it. The second transformation

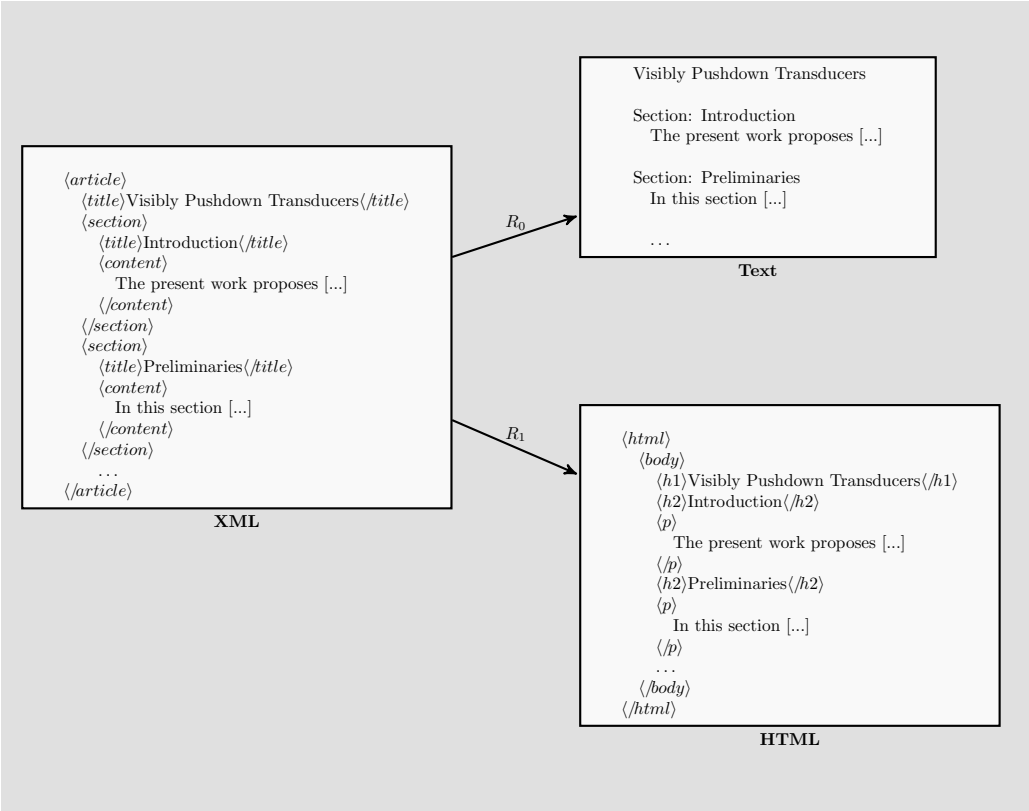

Figure 1: Transformations from XML to Text and HTML

R_1 transforms the XML document into an HTML document. While transformation R_0 completely flattens the input document, transformation R_1 preserves its structure.

Word transducers usually extend automata by producing partial output words on their transitions. The final output of an input word is obtained as the (left-to-right) concatenation of all the partial output words produced along a successful computation. A notable example of word transducers is the *finite state transducers*, that define rational relations, and are built on top of finite state automata. These transducers have been deeply studied [7] but are not suitable to define transformations of documents with a nesting structure. With such documents, before entering an additional level of nesting, it is usually important to store some information about the current nesting level, and to recover it when leaving the nested layer. This is normally implemented with a stack data structure. Another example is the *pushdown transducers* that extend pushdown automata. Clearly, such transducers inherit the bad properties of pushdown automata, and most problems of interest in the theory of transducers are undecidable for this class of transducers.

In this paper, we introduce *visibly pushdown transducers* (VPT), that extend VPA with outputs. The output alphabet is, however, not structured. VPT are therefore *nested word to word* transducers. If the output alphabet is structured, then obviously VPT output nested words. The goal of this paper is to show that, like VPA, VPT enjoy good properties and form a robust class of transducers.

Example 1. *As a first example, consider the* VPT T_0 *of Fig. 2 that implements the XML-to-text transformation R_0 of Fig. 1. Its runs start in state q_0. When reading the call symbol $\langle article \rangle$, T_0 pushes the symbol γ on the stack ($+\gamma$), outputs the empty word ε, and moves to state q_1, from which it can read only $\langle title \rangle$, outputs ε, pushes γ and moves to state q_2. At this point, the input document is expected to be a sequence of characters in Σ_i that corresponds to the title of the article. On the loop on q_2, α is intended to be any internal symbols from Σ_i. On that loop, nothing is pushed on the stack and α is simply copied to the output.*

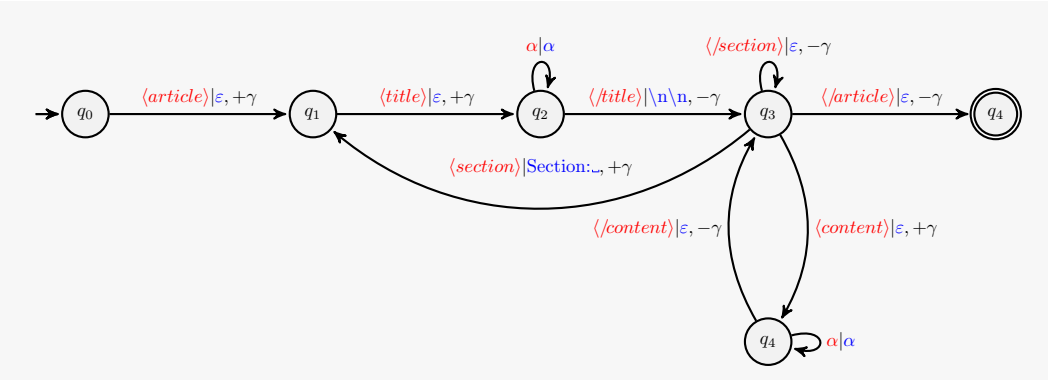

Figure 2: VPT implementing the transformation T_0 of Fig. 1

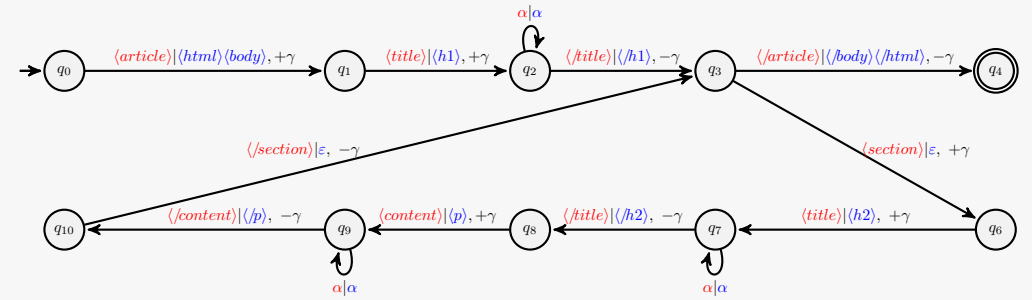

Figure 3: VPT implementing the transformation T_1 of Fig. 1

The only way to exit that loop is to read the closing symbol $\langle /title\rangle$. If it is the case, then two "new line" symbols are appended to the output and γ is popped from the stack ($-\gamma$).

The other transitions of T_0 are self-intuitive. Note that T_0 accepts only well-matched input nested words; for instance, the only way to match $\langle article\rangle$ is to close it with $\langle /article\rangle$. In general, the definition of nested words does not impose any well-matched restriction on the call and return symbols in the input word but it is something that can be syntactically ensured, for instance by using a different stack symbol for each pair of corresponding call/return symbols.

Finally, the domain of T_0, i.e. the set of all input nested words that have at least one output by T_0, is a visibly pushdown language that can be defined by the VPA obtained by projecting away the output words of the transitions of T_0. Note that this domain also contains documents where the sections can be nested in other sections, since it is not necessary to close an opening tag $\langle section\rangle$ before opening a new one.

Example 2. *The transformation R_1 can be implemented by the VPT T_1 of Fig. 3. The output alphabet is assumed to be structured, where call symbols are opening HTML tags and return symbols are closing HTML tags. Observe that in this VPT, sections cannot nest other sections, therefore T_1 and T_0 do not have the same domain. The main title of the article is processed by states q_1, q_2 and q_3, while section titles are processed by states q_6, q_7 and q_8, and those states can be accessed only if $\langle section\rangle$ is read on the input. Note that on reading $\langle article\rangle$, two call symbols are output. Similarly on reading $\langle /article\rangle$, two return symbols are output.*

Contributions

In this paper, we study the closure and algorithmic properties of classes and extensions of VPT. The results are summarized in Table 1. First we distinguish three classes of VPT:

- the class of deterministic VPT (dVPT), whose underlying input VPA is deterministic (and thus, define only functions),
- the class of functional VPT (fVPT), which might be non-deterministic but which define functions,
- the class of VPT, which can be non-deterministic and define relations of arbitrary arity.

Closure properties. Closure by Boolean operators usually do not carry over to transducers. As for finite state transducers, VPT-transductions are not closed by intersection, nor by complement (because any input has a finite number of outputs, while an output may have an infinite number of pre-images). Closure by union is however obtained classically by using non-determinism. Closure under composition is a desirable property of transducers. This property holds for FST. Unfortunately VPT are not closed under composition. We introduce later the class of well-nested VPT to recover closure by composition.

Decision Problems. First, the *emptiness* problem asks whether the transduction defined by a VPT is empty. Clearly, it is a property of the domain of the transduction, which is defined by the VPA underlying the VPT (obtained by projecting the output words). Therefore, emptiness is decidable in PTime for VPT. The *translation membership* problem asks, given a pair (u, v) of input and output words and a VPT T, whether T translates u into v. It is decidable in PTime for all classes of VPT we consider in this paper.

The *typechecking* problem is motivated by practical applications. It asks, given a VPT T, and an input and an output types given as VPA, whether any input document of the given input type is transformed into an output document of the given output type. In the setting of XML for instance, this amounts to check, given an XSLT transformation, an input and an output XML schemas, whether the image of any XML document, valid for the input schema, is a document valid for the output schema. This problem is unfortunately undecidable for VPT because one can define any context-free language (CFL) as the co-domain of a VPT, and as we show, testing the inclusion of a CFL into a visibly pushdown language is undecidable.

One of the most desirable property of transducers is decidable *equivalence checking*, i.e. testing whether two given transducers define the same relation. However even for finite state transducers this problem is undecidable. The classical way to recover decidability is to bound the arity of the relations defined by the transducers. Given $k \in \mathbb{N}$, a transducer is k-valued if every input word has at most k different translations. In particular, 1-valued transducers define exactly the class of functional transducers. For finite-state transducers, equivalence of k-valued transducers is decidable [58]. For VPT, the decidability of equivalence for k-valued transducers is still open, but we show that equivalence of functional VPT is Exptime-complete, and even decidable in PTime for dVPT, which makes this class very appealing.

Although functionality and more generally k-valuedness are semantical properties, we show that they are both decidable, in PTime for functionality and in coNP for k-valuedness. The PTime complexity is a nice consequence of strong result by Plandowski who shows that testing the equivalence of two morphisms applied on a context-free language is decidable in PTime [47]. For k-valuedness, one reduces the problem to deciding language emptiness of a reversal-bounded counter pushdown machine, which is known to be decidable [29], and we show that it is in coNP.

All these decision problems but emptiness and translation membership are undecidable for (non-visibly) pushdown transducers, making VPT an appealing class lying in between finite state transducers and pushdown transducers.

Well-nested VPT. Visibly pushdown transducers form a reasonable subclass of pushdown transductions, but, contrarily to the class of finite state transducers, it is not closed under composition and the type checking is undecidable against VPA. A closer exam of these weaknesses, shows that they are due to the fact that the output alphabet is not structured, or more precisely, to the fact that the stack behavior is not synchronized with the output word. We introduce the class of *well-nested visibly pushdown transducers*

(wnVPT). They are visibly pushdown transducers that produce output on a structured alphabet and whose output is somehow synchronized with the stack and thus, the input. We show that this class is closed under composition and that its type checking problem against visibly pushdown automata is decidable. This makes this class a very attractive one, as its properties are nearly as good as the properties of finite state transducers, but they enjoy an extra expressiveness that is desirable for dealing with documents with a nesting structure. Results on wnVPT and functional wnVPT are summarized in Table 1.

Moreover, the class of well-nested VPT define, modulo linearization, unranked tree to unranked tree transformations. We precisely compare the expressiveness of well-nested VPT with that of known classes of (unranked) tree transducers, most notably, top-down tree transducers running on binary encodings of unranked trees [14], uniform tree transducers [38], and macro tree transducers [19]. While VPT are incomparable to the first two classes, we show that they are strictly less expressive, modulo some tree encoding, than macro tree transducers.

VPT *with regular look-ahead.* We then study the extension of *visibly pushdown transducers with look-ahead* ($\mathsf{VPT}_{\mathsf{la}}$). They are visibly pushdown transducers that have the ability to inspect the suffix of the word. We show that this ability does not add expressiveness, and we exhibit a construction for removing, at an exponential cost, the look-ahead. Moreover, we show that the deterministic VPT with look-ahead exactly capture the functional VPT transductions. Finally, we show that, while they are exponentially more succinct than VPT, testing equivalence or inclusion is done with the same complexity. Results on $\mathsf{VPT}_{\mathsf{la}}$ and deterministic $\mathsf{VPT}_{\mathsf{la}}$ are summarized in Table 1.

Application: streaming transformations

Perhaps the best application and motivation for studying the VPT model, as they can read linearisations of trees as nested words, is streaming XML transformations. While this paper is a thorough study of the fundamental properties of VPT, we have also investigated various streamability problems in [20] for VPT transformations. Let us briefly mention here the main results. Deterministic VPT can efficiently processed XML documents in streaming with a memory that depends only on the height of the stack (i.e. the depth of the XML tree). However, there are some VPT transformations that can still be evaluated with height-dependent memory but are not realizable by any deterministic VPT. In [20], we have shown that checking whether a function realized by a (non-deterministic) VPT can be evaluated with a memory that depends only on the depth of the tree, is a decidable problem. The memory may however be exponential in the depth of the tree. Nevertheless, we have shown that the class VPT-definable functions that can be evaluated with a memory that depend only polynomially on the depth of the XML tree is decidable. Streamability problems for more expressive classes of tree transducers (such as macro tree transducers) have not yet been investigated, but the extra expressive power of those transducers (such as copying or moving subtrees) makes the transformations they define more likely to be non streamable. Related results on the application of visibly pushdown automata to stream validation and queries have also been obtained in [52, 35, 23, 24].

Content and organization of the paper

Visibly pushdown transducers have been first introduced in [48] and some (negative) results sketched in this publication are exposed in the present article. However, [48] focusses on transducers that allow productive ε-moves, leading to very powerful machines. Therefore, for such machines, most of the problems that we investigate here are undecidable. Instead for consider here the real-time version of these transducers. This article gathers material on visibly pushdown transducers that appeared as short version in conference proceedings; in particular, the results from [21] and [22] are extensively presented here. Additionaly, motivatived by XML applications, we consider visibly pushdown transducers as tree transformers. As a new contribution, we will draw a comparison betwwen visibly pushdown transducers and several classes of tree transducers that have been previously defined.

In Section 2, we introduce the basic notions used along this paper. Section 3 introduces visibly pushdown automata and surveys the main known results. In Section 4, we define VPT and study their closure properties. Section 5 presents our results on the decision problems introduced previously for the class of VPT, most

	dVPT	fVPT	VPT	fwnVPT	wnVPT	dVPT_{la}	VPT_{la}
Decision Problems							
Emptiness	P	P	P	P	P	ExpC	ExpC
Translation Membership	P	P	P	P	P	P	P
Typechecking (against VPA)	und	und	und	ExpC	ExpC	und	und
Functionality	-	-	P	-	P	-	ExpC
k-Valuedness	-	-	coNP	-	coNP	-	coNExp
Equivalence	P	ExpC	und	ExpC	und	ExpC	und
Closure Properties							
Union	✗	✗	✓	✗	✓	✗	✓
Intersection	✗	✗	✗	✗	✗	✗	✗
Composition	✗	✗	✗	✓	✓	✗	✗
Inverse	✗	✗	✗	✗	✗	✗	✗

Table 1: Decision Problems and closure properties for all classes of VPT

notably, functionality, equivalence of functional VPT and k-valuedness. In Section 6 we introduce the class of well-nested VPT and show their closure under composition and study the typechecking problem. We also provide a detailed comparison with tree transducers in Section 7. Finally in Section 8, we study the extension of VPT with regular look-ahead.

2. Preliminaries

2.1. Words.

An *alphabet* Σ is a non-empty finite set, its elements are called *symbols* or *letters*. A word u over Σ is a finite, possibly empty, sequence of letters of Σ, *i.e.* $u = a_1 \dots a_n$, with $a_i \in \Sigma$ for all $i \leq n$. The length of the word u is n, its number of letters. The empty word is denoted by ε and its length is 0. The set of all words over Σ is denoted by Σ^*.

Let $u, v \in \Sigma^*$ be two words over Σ. The *mirror image* of $u = a_1 \dots a_n \in \Sigma^*$, denoted by u^r, is the word $a_n \dots a_1$. We write $u \preceq v$ if u is a *prefix* of v, *i.e.* $v = uw$ for some $w \in \Sigma^*$, and we write $u^{-1}v$ for the word obtained after removing the prefix u from v, *i.e.* $u^{-1}v = w$. We denote by $w = u \wedge v \in \Sigma^*$ the *longest common prefix* of u and v, *i.e.* w is the longest word such that $w \preceq u$ and $w \preceq v$.

2.2. Morphism.

Let Σ, Δ be two finite alphabets. A *morphism* is a mapping $\Phi : \Sigma^* \to \Delta^*$ such that $\Phi(\varepsilon) = \varepsilon$ and for all $u, v \in \Sigma^*$, $\Phi(uv) = \Phi(u)\Phi(v)$. A morphism can be finitely represented by its restriction on Σ, i.e. by the set of pairs $(a, \Phi(a))$ for all $a \in \Sigma$. Therefore, for all words $u = a_1 \dots a_n \in \Sigma^*$, we have $\Phi(u) = \Phi(a_1) \dots \Phi(a_n)$. The size of Φ is $|\Sigma| + \Sigma_{a \in \Sigma} |\Phi(a)|$.

2.3. Languages.

Let Σ be an alphabet. A set of words over Σ, $L \subseteq \Sigma^*$, is called a *language* over Σ. The *concatenation* of two languages L_1 and L_2 is the language $L_1 \cdot L_2 = \{u_1 u_2 \mid u_1 \in L_1 \wedge u_2 \in L_2\}$; when clear from the context, we may omit the $\cdot$ operator and write $L_1 L_2$ instead of $L_1 \cdot L_2$. The set of words, L^n, is the set of words obtained by concatenating exactly n words of L, for $n > 0$, and $L^0 = \{\varepsilon\}$. The *star language* of L is defined as $L^* = \bigcup_{n \geq 0} L^n$, and L^+ denotes the set $\bigcup_{n \geq 1} L^n$. The *mirror*, resp. the *complement*, of L is defined as $L^r = \{u^r \mid u \in L\}$, resp. as $\overline{L} = \Sigma^* \setminus L$. A *class* of languages $\mathcal{L}$ is a set of languages, *i.e.* $\mathcal{L} \subseteq 2^{\Sigma^*}$.

We denote by NFA (resp. NPA) the class of non-deterministic finite state automata (resp. non-deterministic pushdown automata). It is well-known that NFA (resp. NPA) define the class of regular languages (resp. context-free languages). We denote by CFL the class of context-free languages.

2.4. Transductions.

Let Σ and Δ be two alphabets. A *transduction* R from Σ^* to Δ^* is a binary relation between words of Σ^* and words of Δ^*, i.e. $R \subseteq \Sigma^* \times \Delta^*$. We say that v is a *transduction*, or an *image*, of u by R whenever $(u, v) \in R$. The set of all images of u by R is denoted $R(u) = \{v \in \Delta^* \mid (u, v) \in R\}$. We extend this notation to sets of words, let $L \subseteq \Sigma^*$, we denote by $R(L) = \cup_{u \in L} R(u)$ the set of all transductions by R of words in L. The *domain* of R, denoted by $dom(R)$, is the set $\{u \in \Sigma^* \mid \exists v \in \Delta^* : (u, v) \in R\}$. The *range* of R, denoted by $range(\mathrm{R})$, is the set $\{v \in \Delta^* \mid \exists u \in \Sigma^* : (u, v) \in R\}$.

Transductions are subsets of $\Sigma^* \times \Delta^*$, as such, the set-related operations do apply to transductions. For example, the intersection of two transductions $R_1 \cap R_2 = \{(u, v) \mid (u, v) \in R_1 \wedge (u, v) \in R_2\}$ is the standard set intersection. The *inverse* transduction of a transduction R from Σ^* to Δ^* is the relation $R^{-1} = \{(u, v) \mid (v, u) \in R\}$ from Δ^* to Σ^*. The *composition* of a transduction R_1 from ${\Sigma_1}^*$ to ${\Sigma_2}^*$ and R_2 from ${\Sigma_2}^*$ to Δ^* is the transduction $R_2 \circ R_1$ from ${\Sigma_1}^*$ to Δ^* such that $R_2 \circ R_1 = \{(u, v) \mid \exists w \in {\Sigma_2}^* : (u, w) \in R_1 \wedge (w, v) \in R_2\}$. The *restriction* of a transduction R to a language L is a transduction, denoted by $R|_L$, such that its domain is the domain of R restricted to words that belong to L, and the image of a word is the image of this word by R, *i.e.* $R|_L = \{(u, v) \mid (u, v) \in R \wedge u \in L\}$.

Transducers are machines to define transductions. They usually extend automata with outputs. Two famous examples of transducers are the non-deterministic finite state transducers (NFT) and the non-deterministic pushdown transducers (NPT) that respectively extend NFA and NPA with outputs [7].

2.5. Structured Alphabets and Nested Words.

A structured alphabet, Σ, is a triple $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$, where Σ_c, Σ_i, and Σ_r, are disjoint alphabets that contain the *call*, the *internal*, and the *return* symbols respectively (or simply the *calls*, the *internals* and the *returns*). We identify Σ with the alphabet $\Sigma_c \cup \Sigma_i \cup \Sigma_r$ and write $a \in \Sigma$ when $a \in \Sigma_c$, $a \in \Sigma_i$ or $a \in \Sigma_r$.

The structure of the alphabet Σ induces a *nesting structure* on the words over Σ. A call position signals an additional level of nesting, while a return position marks the end of a nesting level. A word u is *well-nested* if it is of the following inductive form:

(i) the empty word $u = \varepsilon$,

(ii) an internal symbol $u = a$ with $a \in \Sigma_i$,

(iii) $u = cvr$, where $c \in \Sigma_c$ is the *matching call* (of r), $r \in \Sigma_r$ is the *matching return* (of c), and v is a well-nested word, or

(iv) $u = u_1 u_2$ where u_1 and u_2 are well-nested words.

Within a well-nested word, each call, resp. return, has a unique matching return, resp. call. The set of well-nested words over Σ is denoted by Σ^*_{wn}.

For a nested word u from Σ^*_{wn}, its*nesting level* is defined as: (i) 0 if u is equal to ε or to a for some a in Σ_i, (ii) the nesting level of u' plus one if $u = cu'r$ for c in Σ_c and r in Σ_r, and (iii) the maximum of the heights of u', u'' if $u = u'u''$.

Example 3. *Consider a structured alphabet Σ such that $a \in \Sigma_i$, and for all j, $c_j \in \Sigma_c$ and $r_j \in \Sigma_r$. The word $u = c_0c_1aar_1c_2ar_2r_0c_3r_3$ is well-nested. For all $0 \leq i \leq 3$, the matching return of the call c_i is the return r_i.*

The word $ar_1c_2ar_2r_0c_3c_4r_4$ is not well-nested. r_1 and r_0 are unmatched returns, while c_3 is an unmatched call. The matching return of c_2, resp. c_4, is r_2, resp. r_4.

3. Visibly Pushdown Automata

We introduce in this section visibly pushdown automata [4], which will be then generalized with output to define visibly pushdown transducers.

A VPA is a pushdown automaton that operates on a structured alphabet. Its behavior is constrained by the type of the input letter. This restriction on the use of the stack ensures that the height of the stack along reading a word depends on the word only and not on the particular run used to read it. Therefore, the stacks of all VPA reading a same input word have the same height at any given position in the input word. This enables the important product construction and yields closure under intersection. The structure of the input words and the restriction on the stack behavior make VPA very close to tree automata, as shown in [5], so that they inherit all the good properties of tree automata.

We now formally define VPA, then we recall all closure properties and the main decision procedures for this class of automata. Finally, we compare the expressive power of VPL, regular languages and context-free languages. All results and proofs (and much more) can be found in [5].

3.1. Visibly pushdown automata

A VPA is defined by a set of states Q (some are initial and some are final), a stack alphabet Γ and a transition relation δ. There are three types of transitions, *call, return and internal transitions*, which correspond to the type of the input letter it reads. A *call transition* occurs when the automaton reads a call symbol. In that case it must push exactly one symbol onto its stack (and cannot read the stack). A *return transition* occurs when the automaton reads a return symbol. In that case it must pop exactly one symbol from its stack. Additionally, a VPA can make return transitions on empty stack. The empty stack is represented by the unremovable bottom of the stack symbol $\perp$. An *internal transition* occurs when the automaton reads an internal symbol. In that case the stack is neither touched nor read. A VPA starts its execution in one of the predefined initial states and with its stack empty. It accepts a word if it terminates its reading in one of the predefined final states, possibly with a non-empty stack.

Definition 4 (Visibly Pushdown Automata)**.** A *visibly pushdown automaton* (VPA) on finite words over a structured alphabet $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$ is a tuple $A = (Q, I, F, \Gamma, \delta)$ where:

- Q is a finite set of states, $I \subseteq Q$, is the set of initial states, $F \subseteq Q$, is the set of final states,
- Γ is the (finite) stack alphabet, and $\perp \notin \Gamma$ is the bottom of the stack symbol,
- $\delta = \delta_c \uplus \delta_i \uplus \delta_r$ is the transition relation where :
  - $\delta_c \subseteq Q \times \Sigma_c \times \Gamma \times Q$ are the *call transitions*,
  - $\delta_r \subseteq Q \times \Sigma_r \times (\Gamma \cup \{\perp\}) \times Q$ are the *return transitions*, and
  - $\delta_i \subseteq Q \times \Sigma_i \times Q$ are the *internal transitions.*

A configuration of A is a pair (q, σ) where $q \in Q$ is a state and $\sigma \in \perp \cdot \Gamma^*$ a stack. Let $w = a_1 \dots a_l$ be a word on Σ, and $(q, \sigma), (q', \sigma')$ be two configurations of A. A *run* of the VPA A over w from (q, σ) to (q', σ') is a sequence of transitions $\rho = t_1 t_2 \dots t_l \in \delta^*$ (where δ^* is the set of words on δ considered as an alphabet) such that there exist $q_0, q_1, \dots q_l \in Q$ and $\sigma_0, \dots \sigma_l \in \perp \cdot \Gamma^*$ with $(q_0, \sigma_0) = (q, \sigma)$, $(q_l, \sigma_l) = (q', \sigma')$, and for each $0 < k \leq l$, we have either:

- $t_k = (q_{k-1}, a_k, \gamma, q_k) \in \delta_c$ and $\sigma_k = \sigma_{k-1}\gamma$,
- $t_k = (q_{k-1}, a_k, \gamma, q_k) \in \delta_r$, and $\sigma_{k-1} = \sigma_k\gamma$ or $\sigma_{k-1} = \sigma_k = \gamma = \perp$, or
- $t_k = (q_{k-1}, a_k, q_k) \in \delta_i$, and $\sigma_{k-1} = \sigma_k$.

Note that when $w = \varepsilon$ (i.e. $l = 0$), a run ρ over w is necessarily an empty sequence and this run goes from any configuration onto itself. We write $q \xrightarrow{c,+\gamma} q'$ when $(q, c, \gamma, q') \in \delta_c$, $q \xrightarrow{r,-\gamma} q'$ when $(q, r, \gamma, q') \in \delta_r$, and $q \xrightarrow{a} q'$ when $(q, a, q') \in \delta_i$. We also write $A \models (q, \sigma) \xrightarrow{u} (q', \sigma')$ when there is a run of A over u from (q, σ) to (q', σ'), moreover we may omit A or the stacks σ or σ', when it is clear or irrelevant in the context.

A run over w from (q, σ) to (q', σ') is *accepting* if $q \in I$, $\sigma = \perp$, and $q' \in F$. A word w is *accepted* by A if there exists an accepting run of A over w. Note that it implies that ε is accepted iff $I \cap F \neq \varnothing$. $L(A)$,

the *language* of A, is the set of words accepted by A. A language L over Σ is a *visibly pushdown language* (VPL) if there is a VPA A over Σ such that $L(A) = L$.

A VPA $A = (Q, I, F, \Gamma, \delta)$ is said to be *deterministic* if the following conditions hold: (i) if $(q, a, \gamma, q'), (q, a, \gamma', q'') \in \delta_c$ then $\gamma = \gamma'$ and $q' = q''$, (ii) $(q, a, \gamma, q'), (q, a, \gamma, q'') \in \delta_r$ then $q' = q''$, (iii) $(q, a, q'), (q, a, q'') \in \delta_i$ then $q' = q''$, and (iv) $|I| = 1$. The set of deterministic VPA is denoted dVPA. A VPA is said to be *unambiguous* if it admits at most one accepting run per input.

The size of a VPA A is defined by $|A| = |Q| + |\delta| + |\Gamma|$.

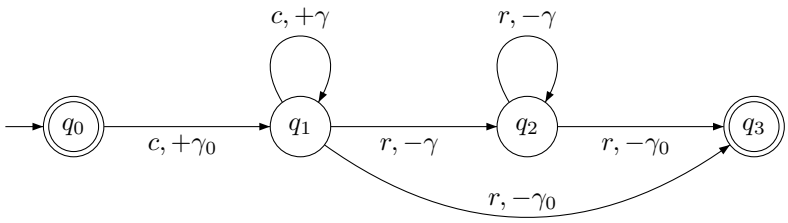

Figure 4: VPA A_3 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Example 5. *The deterministic* VPA $A_3 = (Q, I, F, \gamma, \delta)$ *is represented in Figure 4, where* $Q = \{q_0, q_1, q_2, q_3\}$, $I = \{q_0\}$, $F = \{q_0, q_3\}$, $\Gamma = \{\gamma_0, \gamma\}$, *and* δ *is represented by the edges of the graph. It recognizes the non regular language* $\{c^n r^n \mid n \geq 0\}$.

Example 6. *Let* $\Sigma = \{a, b, c, d\}$. *The context-free language* $\{a^n b^n c^m d^m \mid n, m \geq 0\}$ *is a* VPL *for the partition* $\Sigma_c = \{a, c\}$ *and* $\Sigma_r = \{b, d\}$. *However the context-free language* $L = \{a^n b^n c^m a^m \mid n, m \geq 0\}$ *is not a* VPL *for any partition of* Σ.

3.2. Closure Properties

Closure under union is obtained thanks to non-determinism, while closure under complement is obtained with a determinization procedure. We recall the product construction which underlies the closure under intersection.

The product of the VPA A and B is a VPA that simulates in parallel A and B. A state of the product is a pair of states: a state of A and one of B. The stack simulates both stacks. This is possible because the stacks of A and B are *synchronized*, *i.e.* they are either both popped or both pushed. Therefore the stack of the product contains pairs of symbols: a symbol from the stack of A and another from the stack of B.

The product is obtained by taking the cartesian product of the set of states and of the stack alphabet of the two VPA, and by a synchronized product of the transition relation.

Definition 7 (Product)**.** *Let* $A_1 = (Q_1, I_1, F_1, \Gamma_1, \delta_1)$ *and* $A_2 = (Q_2, I_2, F_2, \Gamma_2, \delta_2)$ *be two* VPA. *The product of* A_1 *and* A_2 *is a* VPA*:*

$$A_1 \otimes A_2 = ((Q_1 \times Q_2), (I_1 \times I_2), (F_1 \times F_2), (\Gamma_1 \times \Gamma_2), (\delta_1 \otimes \delta_2))$$

where $\delta_1 \otimes \delta_2$ *is defined as follows, for all* $a \in \Sigma, q_1, q_1' \in Q_1, q_2, q_2' \in Q_2, \gamma_1 \in \Gamma_1, \gamma_2 \in \Gamma_2$*:*

calls and returns $((q_1, q_2), a, (\gamma_1, \gamma_2), (q_1', q_2')) \in \delta_1 \otimes \delta_2$ *iff* $(q_1, a, \gamma_1, q_1') \in \delta_1$ *and* $(q_2, a, \gamma_2, q_2') \in \delta_2$.

returns on $\perp$ $((q_1, q_2), a, \perp, (q_1', q_2')) \in \delta_1 \otimes \delta_2$ *iff* $(q_1, a, \perp, q_1') \in \delta_1$ *and* $(q_2, a, \perp, q_2') \in \delta_2$.

internals $((q_1, q_2), a, (q_1', q_2')) \in \delta_1 \otimes \delta_2$ *iff* $(q_1, a, q_1') \in \delta_1$ *and* $(q_2, a, q_2') \in \delta_2$.

Each transition of the product $A_1 \otimes A_2$ corresponds to a pair formed by a transition of A_1 and a transition of A_2. Furthermore, any run ρ of $A_1 \otimes A_2$ over a word u is in a one to one correspondence with a pair formed by a run ρ_1 of A_1 over u and a run ρ_2 of A_2 over u. We therefore may write $\rho = \rho_1 \otimes \rho_2$. Clearly the language accepted by the product of two VPA is the intersection of the languages accepted by these automata.

A VPA that recognizes the complement of the language of a VPA A is obtained by determinization [5] of A, completion (adding a sink state and the corresponding transitions) and by complementing the set of final states. The resulting VPA can be exponentially larger.

Finally, note that for a given structured alphabet Σ, the class of VPL over Σ is not closed under mirror image. Indeed, Let $c \in \Sigma_c$ and $r \in \Sigma_r$, the language $L = \{c^n r^n \mid n \geq 0\}$ is a VPL, but its mirror image $L' = \{r^n c^n \mid n \geq 0\}$ is not. Moreover, it is not closed under morphism. For example, consider the language L and a morphism h with $h(c) = r$ and $h(r) = c$, then $h(L) = L'$ which is not a VPL.

The closure properties are summarized in the following proposition.

Proposition 8 (Closure [5])**.** *The class of* VPL *is closed under union, intersection, complement, Kleene star and concatenation. When the languages are given by* VPA*, these closure are effective and can be computed in exponential time for the complement and polynomial for the other operations. If a* VPA *is deterministic, its complement can be computed in polynomial time.*

We now compare the expressive power of VPL with regular languages and context-free languages.

Given an NFA A (running on some alphabet that turns to be structured), one can easily construct in linear time an equivalent VPA whose runs are in a one-to-one correspondence with the runs of A : for all $(p, a, q) \in \delta$ we have $(p, a, \gamma, q) \in \delta'_c$ if $a \in \Sigma_c$, $(p, a, \gamma, q) \in \delta'_r$ and $(p, a, \bot, q) \in \delta'_r$ if $a \in \Sigma_r$ and $(p, a, q) \in \delta'_i$ if $a \in \Sigma_i$, where δ' is the transition relation of the VPA and γ is a unique arbitrary stack symbol. Therefore the class of VPL is a strict generalization of regular languages. Note that, this does not hold if the automaton is not allowed to make return transitions on the empty stack ($\bot$), neither if it should accept only by empty stack (and not by final states). Indeed, it would not be able to recognize, in the former case, the language $\{r\}$ where $r \in \Sigma_r$, and, in the latter case, the language $\{c\}$ where $c \in \Sigma_c$.

While the class of VPL is a strict subclass of CFL (see Example 6), there is also a strong connection the other way round between those classes. Recall that the tagged alphabet $\hat{\Sigma} = (\overline{\Sigma}, \Sigma, \underline{\Sigma})$ is a structured alphabet where $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$ and $\underline{\Sigma} = \{\underline{a} \mid a \in \Sigma\}$ are the call and the return symbols respectively. The projection $\pi : \hat{\Sigma} \to \Sigma$ is defined as $\pi(\overline{a}) = \pi(\underline{a}) = \pi(a) = a$ for all $a \in \Sigma$. Any CFL can be obtained as the projection of some VPL.

Proposition 9 (Relation with CFL [5])**.** *We have* VPL $\subsetneq$ CFL*. Let $A \in$* VPL*, one can construct in linear time an equivalent* NPA*. Moreover, let $L \in$* CFL(Σ) *there exists a* VPL L' *on $\hat{\Sigma}$ such that $\pi(L') = L$.*

The main decision problems for VPA are all decidable. Decidability of the emptiness and of the membership problems are direct consequences of the corresponding decidability results for pushdown automata. Then, as a consequence of the decidability of the emptiness and the closure under Boolean operations, universality, inclusion and equivalence are all decidable.

Theorem 10 ([5])**.** *The emptiness and membership problem for* VPA *are decidable in* PTime*. Universality, inclusion and equivalence of* VPA *are* ExpTime*-c, and in* PTime *when the* VPA *are deterministic.*

The inclusion problems of NFA into VPA and conversely are both decidable. However, inclusion of NFA into non-deterministic pushdown automata (NPA) is undecidable, therefore so is the inclusion of VPA into NPA. However, inclusion of an NPA into an NFA is decidable as one can compute the product of the complement of NFA with the NPA. This gives a pushdown automaton that is empty if and only if the inclusion holds. In contrast, for the inclusion of a NPA into a VPA, we prove the next proposition.

Proposition 11 (Inclusion with CFL)**.** *Given $C \in$* CFL *(defined by some* NPA*) and $V \in$* VPL *(defined by some* VPA*), it is undecidable whether $V \subseteq C$ (respectively $C \subseteq V$) holds.*

Proof. The undecidability of $V \subseteq C$ is a direct consequence of the undecidability of the inclusion of NFA into NPA.

For the other direction, let $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ be an instance of the Post Correspondence Problem (PCP) defined on some finite alphabet Δ. We define a CFL and a VPL languages on the structured alphabet $\Sigma = (\Sigma_c, \varnothing, \Sigma_r)$ with $\Sigma_c = \Delta$ and $\Sigma_r = \{1 \ldots n\}$. For all j, we let $l_j = |u_j|$. The CFL language is $C = \{v_{i_1} \ldots v_{i_k} \#(i_k)^{l_k} \ldots (i_1)^{l_1} \mid i_1, \ldots, i_k \in \Sigma_r\}$. Let $V' = \{u_{i_1} \ldots u_{i_k} \#(i_k)^{l_k} \ldots (i_1)^{l_1} \mid i_1, \ldots, i_k \in \Sigma_r\}$. It is easy to check that V' is a VPL on Σ, therefore its complement $V = \overline{V'}$ also is. Clearly the PCP instance is negative if and only if $C \subseteq V$. Finally, note that C and V are obviously definable by NPA and VPA respectively. □

4. Visibly Pushdown Transducers

Visibly pushdown transducers (VPT) are visibly pushdown automata with outputs. They are a generalization of non-deterministic finite state transducers, and form a strict subclass of pushdown transducers.

We define a VPT as a VPA with an output morphism. While the input alphabet is structured and synchronizes the stack of a VPT (*i.e.* the stack of its underlying VPA), the output is not structured: the output alphabet is not necessarily a structured alphabet. In this section $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$ is the input structured alphabet and Δ is the arbitrary output alphabet.

Definition 12 (Visibly Pushdown Transducers)**.** *A visibly pushdown transducer (VPT) from Σ to Δ is a pair $T = (A, \Omega)$ where the VPA $A = (Q, I, F, \Gamma, \delta)$ is the underlying automaton and Ω is a morphism from δ to Δ^* called the output.*

A *run* ρ of T over a word $u = a_1 \ldots a_l$ is a run of its underlying automaton A (Definition 4), *i.e.* it is a sequence of transition $\rho = t_1 \ldots t_l \in \delta^*$. The output of ρ is the word $v = \Omega(\rho) = \Omega(t_1 \ldots t_l) = \Omega(t_1) \ldots \Omega(t_l)$. We write $(q, \sigma) \xrightarrow{u/v} (q', \sigma')$ when there exists a run on u from (q, σ) to (q', σ') producing v as output. The transducer T defines a binary word relation $R(T) = \{(u, v) \mid \exists q \in I, q' \in F, \sigma \in \Gamma^*,\ (q, \bot) \xrightarrow{u/v} (q', \sigma)\}$. We say that a transduction is a VPT transduction whenever there exists a VPT that defines it. test As usual, the *domain* of T (denoted by $dom(T)$) and the *range* of T (denoted by $range(T)$) are respectively the domain and the range of $R(T)$. We say that T is functional whenever $R(T)$ is, and that T is deterministic (resp. unambiguous) if A is.

The class of functional (resp. deterministic) VPT is denoted by fVPT (resp. dVPT), and the class of all VPT (resp. functional VPT, resp. deterministic VPT) transductions is denoted by R(VPT) (resp R(fVPT), resp. R(dVPT)). The size $|T|$ of $T = (A, \Omega)$ is $|A| + |\Omega|$.

Remark 13. *As VPA do not allow ε-transitions, neither do VPT on the input. However some transitions might output the empty word ε. In that sense the class of VPT defined in Definition 12 is the class of real-time VPT.*

Visibly pushdown transducers have been first introduced in [48] and independently in [55]. In these papers, VPT allow for ε-transitions that can produce outputs and only a single letter can be produced by each transition. Adding ε-transitions to VPT can only be done by disallowing the use of the stack for such transitions. Indeed, the visibly constraints, asks that the input word drives the behavior of the stack, therefore, this is only compatible with internal ε-transitions.

When the two classes of transducers NFT and NPT are augmented with ε-transitions, then these new classes enjoy closure under inverse, contrarily to their ε-free counterparts. However, with such internal ε-transitions, the enhanced class of VPT is still not closed under inverse. Indeed, the following transduction is a VPT transduction, but its inverse cannot be defined by such VPT:

$$\{(c^n r^n, \varepsilon) \mid n \leq 0 \wedge r \in \Sigma_r \wedge c \in \Sigma_c\}$$

Morevover, using ε-transitions the way of [48] and [55] causes many interesting problems to be undecidable, such as functionality and equivalence (even of functional transducers). Therefore we prefer to stick to

Definition 12 of VPT*, they are exactly the so called nested word to word transducers of [54] and correspond to the definition of [21]. XML-DPDTs [34], equivalent to left-to-right attribute grammars, correspond to the deterministic* VPT*.*

In the sequel, we do not investigate further VPT *augmented with ε-transitions.*

Relations between the classes of transducers and of transductions we have considered so far can be depicted as :

$$\mathsf{dVPT} \subsetneq \mathsf{fVPT} \subsetneq \mathsf{VPT} \qquad R(\mathsf{dVPT}) \subsetneq R(\mathsf{fVPT}) \subsetneq R(\mathsf{VPT})$$

For the classes of machines, the first inclusion on the left is trivial as a determinitic VPT admits at most one accepting run on any of its inputs and thus, is functional; moreover, it is strict as fVPT contains machines that are not deterministic. The inclusion on the right is strict as VPT can describe non-functional relations thanks to the non-determinism of their underlying VPA. The same argument holds for transductions, separating strictly $R(\mathsf{fVPT})$ from $R(\mathsf{VPT})$. The first inclusion for transductions comes from the inclusion of the corresponding classes of machines. It is strict since we will see an example in the next section of a functional VPT that has no equivalent deterministic VPT. Finally, note that an unambiguous VPT is necessarily functional but the converse is false as we will see in the next section.

4.1. Examples

We present various examples to illustrate the definition of VPT.

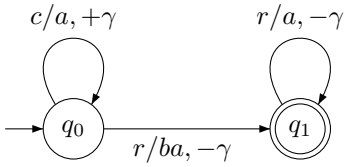

Figure 5: A VPT T_1 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Example 14. *This first example is a deterministic* VPT *$T_1 = (A, \Omega)$ represented in Figure 5. It operates on the input structured alphabet $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$ where $\Sigma_c = \{c\}$, $\Sigma_r = \{r\}$, and $\Sigma_i = \varnothing$. It produces words over the alphabet $\Delta = \{a, b\}$.*

The underlying automaton is the VPA *$A_1 = (Q_1, I_1, F_1, \Gamma_1, \delta_1)$ where the set of states is $Q_1 = \{q_0, q_1\}$, the set of initial states is $I_1 = \{q_0\}$, the set of final states $F_1 = \{q_1\}$, the stack alphabet is $\Gamma_1 = \{\gamma\}$, and the transition relation is $\delta_1 = \{(q_0, c, \gamma, q_0), (q_0, r, \gamma, q_1), (q_1, r, \gamma, q_1)\}$. The domain of T_1 is the language $L(A_1)$,* i.e. *the set:*

$$dom(T_1) = L(A_1) = \{c^n r^m \mid 1 \leq m \leq n\}$$

The output morphism Ω_1 is defined on δ as follows. $\Omega_1((q_0, c, \gamma, q_0)) = a$, $\Omega_1((q_0, r, \gamma, q_1)) = ba$, and $\Omega_1((q_1, r, \gamma, q_1)) = a$.

A run starts in the initial state q_0. When in state q_0, the transducer reads any number of c and for each of them it outputs an a and pushes γ onto the stack (push operations are represented as $+\gamma$). In state q_0 it can also read an r if the top of the stack is γ (that is if at least one c was read before), it pops the symbol γ (represented as $-\gamma$), outputs the word ba and changes its state to q_1. While in q_1, the transducer can read as many r as they are symbols γ on the stack, for each r it outputs a and pops γ. The run is accepting if and only if it ends in the final state q_1.

Clearly T_1 implements the following (functional) transduction $\{(c^n r^m, a^n b a^m) \mid 1 \leq m \leq n\}$. The range of T_1 is the context-free language range$(T_1) = \{a^n b a^m \mid 1 \leq m \leq n\}$ which is not a visibly pushdown language for any partition of the output alphabet.

The next example presents a non-deterministic VPT. This VPT is, however, functional, furthermore one can easily show that there is no equivalent deterministic VPT. In other words, it is an example of non-determinizable functional VPT.

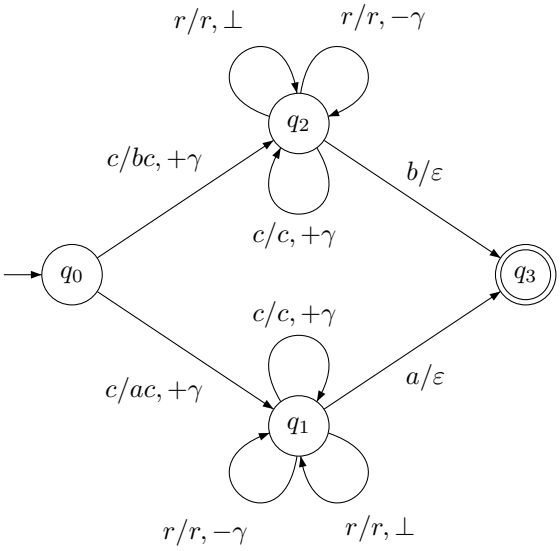

Figure 6: A VPT T_4 on $\Sigma_c = \{c\}$, $\Sigma_r = \{r\}$, and $\Sigma_i = \{a, b\}$.

Example 15. *The* VPT T_4 *of Figure 6 starts in state* q_0*, it first reads a* c *and guesses the last letter to be either an* a *or a* b*. If the last letter is an* a*, resp. a* b*, it outputs* ac*, resp.* bc*, and goes to state* q_1*, resp.* q_2*. In states* q_1 *and* q_2*, the transducer performs a simple copy of the words formed by* c *and* r *letters. Finally it checks that the last letter matches its initial guess, and if so enters the final state* q_3*. The* VPT T_4 *implements the functional transduction* $\{(cw\alpha, \alpha cw) \mid \alpha \in \{a, b\}, w \in \{c, r\}^*\}$*. Note that* T_4 *uses return transition on* $\bot$*, it permits to read the return symbol* r *even when the stack is empty.*

4.2. Expressiveness

The domain of a VPT is the language accepted by its underlying automaton, therefore the domain is a visibly pushdown language. The range of a VPT is the image of the language of its runs by the output morphism. The runs of a VPA form a CFL (even a VPL), and as the image of a CFL by a morphism is a CFL, so is the range of a VPT. Finally, by Proposition 9, it is easy to show that for any CFL L there exists a VPT such that its range is L.

Proposition 16 (Domain, range and image)**.** *Let* T *be a* VPT*. The domain of* T *is a* VPL*, and its range is a* CFL*. Moreover, for any* CFL L' *over* Σ*, there exists a* VPT *whose range is* L'*. All the constructions can be done in* PTIME.

The class of VPT is a strict subclass of pushdown transducers. Indeed, any VPT is clearly a non-deterministic pushdown tranducer. But some non-deterministic pushdown transductions are not VPT transduction, for instance those non-deterministic pushdown transduction whose domain is a context-free language but not a VPL. Obviously, VPT forms a strict superclass of finite state transducers.

This last result obviously does not hold for finite state tranducers with ε transitions, as our VPT do not have ε-transitions, they can, therefore, not implement all transductions defined by such transducers.

4.3. Closure Properties

We now turn to closure properties. The product of a VPT $T = (A, \Omega)$ by a VPA B is defined as the VPT whose underlying automaton is the product of A with B and whose output is the output of T, that is $T|_B = (A \otimes B, \Omega \circ \pi_1)$ where π_1 is the projection on the first component.

Proposition 17 (Domain restriction). $R(T|_B) = R(T)|_{L(B)}$.

The range of $T_{|B}$ is equal to the image of $L(B)$ by T. Therefore by Proposition 16, the image of a VPL by a VPT is a context-free language.

Corollary 18. *Let T be a* VPT*, and B be a* VPA*. The image of $L(B)$ by T is a context-free language and an* NPA *representing it can be computed in* PTIME.

Note also that for any word $u \in \Sigma^*$ the language $\{u\}$ is a VPL, therefore one can compute a NPA representing the image of a word in PTIME.

As usual, thanks to non-determinism, transductions defined by VPT are closed under *union*.

Proposition 19 (Closure under union). *The class of* VPT *is effectively closed under union. A* VPT *representing the union of T_1 and T_2 can be computed in $O(|T_1| + |T_2|)$.*

However, it is not closed for other operators.

Proposition 20 (Non-Closure). *The class of* VPT *and of* dVPT *are not closed under intersection, complement, composition, nor inverse. The class of* dVPT *is not closed under union.*

Proof. The class of VPT and dVPT is not closed under intersection : inspired by finite state transducers, we prove this result for a structured alphabet containing only internal symbols.

Let $\Sigma = \{a, b\}$ and $\Delta = \{c\}$ be two alphabets. Let T_1 be a transducer defining the transduction $R(T_1) = \{(a^m b^n, c^m) \mid m, n \geq 0\}$ and the transducer T_2 has the same underlying automaton and defines the transduction $R(T_2) = \{(a^m b^n, c^n) \mid m, n \geq 0\}$. We have

$$R(T_1) \cap R(T_2) = \{(a^n b^n, c^n) \mid n \geq 0\}$$

The domain of this transduction is not a VPL language (remind that a and b are internals). Therefore, this transduction cannot be defined by a VPT.

Non-closure under inverse is a consequence of Proposition 16: the language of the domain and the co-domain do not belong to the same family of languages. It is also a consequence of the fact that VPT, resp dVPT, are realtime, and thus for any VPT T, for any word w, $T(w)$ is a finite set while a word w can be the image of an infinite number of input words.

Non-closure under composition can be proved by producing two VPT whose composition transforms a VPL into a non CFL language. Formally, let $\Sigma = (\Sigma_c, \Sigma_r, \Sigma_i)$ be the structured alphabet with $\Sigma_c = \{c\}$, $\Sigma_r = \{r_1, r_2\}$, $\Sigma_i = \{a\}$. First consider the following VPL language: $L_1 = \{c^n r_1^n a^m \mid n, m \geq 0\}$. We can easily construct a VPT that transforms L_1 into the language $L_2 = \{c^n a^n r_2^m \mid n, m \geq 0\}$. Applying the identity transducer on L_2 restricted to well-nested words, it produces the non CFL language $L_3 = \{c^n a^n r_2^n \mid n \geq 0\}$. This identity transducer has a domain which is a VPL and thus it extracts from L_2 the well-nested words which form the non CFL set L_3. Note that these two transducers are deterministic. Therefore this completes the proof that neither VPT nor dVPT are closed under composition. □

5. Decision Problems

In this section, we study relevant decision problems for transductions for the class of VPT: emptiness, membership, type checking, functionality and k-valuedness, inclusion and equivalence.

5.1. Emptiness and Translation Membership

The *emptiness* problem asks, given a VPT T, whether $R(T) = \varnothing$ holds. Note that it amounts to check the emptiness of $dom(T)$.

The *translation membership problem* asks, given a VPT T and a pair of input and output words, $u \in \Sigma^*$ and $v \in \Delta^*$ whether $(u, v) \in R(T)$ holds.

The emptiness and translation membership problems are decidable in PTIME for pushdown transducers. Therefore, since VPT are pushdown transducers, these results trivially hold for VPT.

Proposition 21 (Decidable problems). *The emptiness and the translation membership problems are decidable in* PTIME *for* VPT.

5.2. Type Checking

The *type checking problem* for a VPT T against VPA asks, given two VPA A, B defining respectively an input language $L(A)$ and an output language $L(B)$ whether the images of any word of the input language belong to the output language, *i.e.* whether $R(T)(L(A)) \subseteq L(B)$ holds.

Theorem 22 (Type Checking)**.** *The type checking problem for* VPT *against* VPA *is undecidable.*

Proof. Let Σ be an alphabet and $\hat{\Sigma} = (\Sigma_c, \Sigma_i, \Sigma_r)$ the tagged alphabet on Σ. Recall that π_Σ is the morphism from $\hat{\Sigma}$ into Σ that 'removes' the tag, *i.e.* $\overline{a}$, $\underline{a}$ and a are all mapped to a. Let A be a NPA and $L = L(A)$ and A_2 a VPA with $L_2 = L(A_2)$. Let T be the VPT from $\hat{\Sigma}$ into Σ such that for all $u \in \hat{\Sigma}^*$ we have $T(u) = \pi_\Sigma(u)$. By Proposition 9, one can construct a VPA A_1 with $\pi_\Sigma(L(A_1)) = L$, that is, if $L_1 = L(A_1)$, $T(L_1) = L$. Therefore $T(L_1) \subseteq L_2$ if and only if $L \subseteq L_2$. As the inclusion of a NPA into a VPA is undecidable (Proposition 11), so is the type checking of VPT against VPA. □

Note that this undecidability result is not a weakness of VPT as it already holds for the type checking of finite state transducers against VPA (the transduction of the previous proof can be implemented by some finite-state transducer).

One way to overcome this problem is to restrict the output class of languages to regular languages. In this case, the type checking becomes decidable in ExpTime. We will see later how to get decidability of type checking against VPA by restricting the class of VPT.

5.3. Functionality

A VPT T is *functional* if $R(T)$ is a function. The *functionality* problem aims to decide whether a VPT T is functional. In this section we prove that this problem is in PTime.

Let us start with some example.

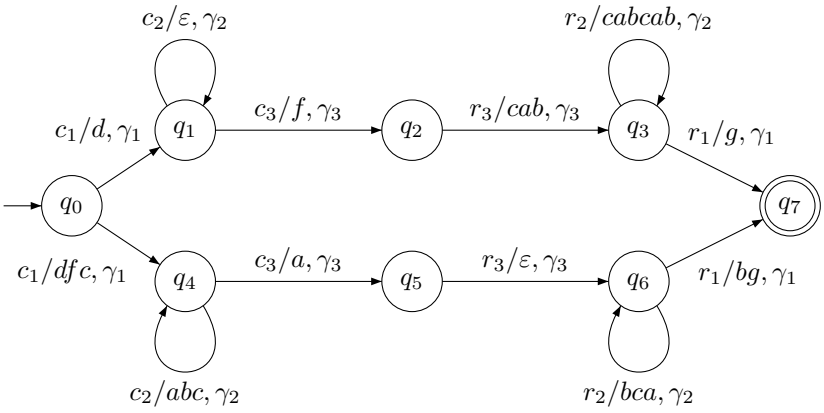

Figure 7: A functional VPT on $\Sigma_c = \{c_1, c_2, c_3\}$ and $\Sigma_r = \{r_1, r_2, r_3\}$.

Example 23. *Consider the* VPT T *of Figure 7. Call and return symbols are denoted by c and r respectively. The domain of T is $dom(T) = \{c_1(c_2)^n c_3 r_3 (r_2)^n r_1 \mid n \in \mathbb{N}\}$. For each word of $dom(T)$, there are two accepting runs, corresponding respectively to the upper and lower part of T (therefore it is not unambiguous). For instance, when reading c_1, it pushes γ_1 and produces either d (upper part) or dfc (lower part). By following the upper part (resp. lower part), it produces words of the form $dfcab(cabcab)^n g$ (resp. $dfc(abc)^n a(bca)^n bg$).*

The VPT T of Example 23 is functional. Indeed, the upper part performs the transformation:

$$c_1 c_2^n c_3 r_3 r_2^n r_1 \to dfcab(cabcab)^n g = df(cab)^{2n+1} g$$

and the lower part performs:

$$c_1c_2^nc_3r_3r_2^nr_1 \to dfc(abc)^na(bca)^nbg = df(cab)^{2n+1}g$$

The challenge for deciding functionality for VPT lies in the fact that for some functional VPT outputs are produced in some desynchronised manner and the difference in their lengths might get arbitrarily long. This notion has been formalized as the *delay* between two output words u and v as $\Delta(u,v) = ((u \wedge v)^{-1}u, (u \wedge v)^{-1}v)$. Note that the fact that delays remains bounded is the core of the proof of the decidability of functionality for finite-state transducers [6]. As shown by this example, the delay between two outputs for a VPT can grow arbitrarily, even if it is functional. Indeed, after reading $c_1c_2^n$, the upper part outputs just d, while the lower part outputs $dfc(abc)^n$: the delay between both outputs is $fc(abc)^n$ and grows linearly with the height of the stack. For T to be functional, the upper run must catch up its delay before reaching an accepting state. In this case, it will catch up with the outputs produced on the return transitions.

The decision procedure for functionality of VPT is based on the squaring construction [6] and on the decidability of the morphism equivalence problem [47]. The *square* T^2 of a VPT T is realized through a product construction of the underlying automaton with itself. Intuitively, it is a transducer that simulates any two parallel transductions of T, where by *parallel transductions* we mean two transductions over the same input word. Each transition of the square simulates two transitions of T that have the same input letter. The output of a transition t of T^2 that simulates the transitions t_1 and t_2 of T is the pair of output words formed by the outputs of t_1 and t_2. If Ω is the output morphism of T, we write O_T for $\Omega(\delta)$, that is, the set of words that are output of transitions of T, then the output alphabet of T^2 is the set of pairs of words in O_T, *i.e.* $O_T \times O_T$.

Recall that the transitions of the product $A_1 \otimes A_2$ of two VPA, A_1 and A_2 (See Definition 7), are in a one to one correspondence with pairs formed by a transition of A_1 and a transition of A_2 over the same input letter. We can therefore write (t_1, t_2), where t_1 and t_2 are transitions of A_1 and A_2, to denote a transition of $A_1 \otimes A_2$.

Definition 24 (Square)**.** *Let $T = (A, \Omega)$ be a* VPT*, where $A = (Q, I, F, \Gamma, \delta)$ and Ω is a morphism from δ to Δ^*, and let $O_T = \Omega(\delta)$. The* square *of T is the* VPT *$T^2 = (A \otimes A, \Omega')$ where Ω' is a morphism from $\delta \otimes \delta$ to $O_T \times O_T$ defined as: $\Omega'((t_1, t_2)) = (\Omega(t_1), \Omega(t_2))$.*

Note that the square T^2 is a VPT and therefore its range is a context-free language (Proposition 16) over the alphabet $O_T \times O_T$.

The procedure to decide functionality of a VPT T is based on the following observation. A run $\rho = (t_1, t'_1) \dots (t_k, t'_k)$ of T^2 simulates two runs, say $\rho_1 = t_1 \dots t_k$ and $\rho_2 = t'_1 \dots t'_k$, of T over a same input word. The output of ρ is the sequence of pairs of words $\Omega'(\rho) = \Omega'((t_1, t'_1) \dots (t_k, t'_k)) = (\Omega(t_1), \Omega(t'_1)) \dots (\Omega(t_k), \Omega(t'_k))$. Therefore the projection on the first component, resp. second, of the output of ρ gives the output of the corresponding run of T, that is $\Omega(\rho_1)$ and $\Omega(\rho_2)$ respectively. These projections on the first and second components can be defined by two morphisms Π_1 and Π_2 as follows: $\Pi_1((u_1, u_2)) = u_1$ and $\Pi_2((u_1, u_2)) = u_2$. Clearly, T is functional if and only if these two morphisms are equal on any output of T^2, *i.e.* if for all $w = (u_1, u'_1) \dots (u_k, u'_k) \in range(T^2)$ we have $\Pi_1(w) = \Pi_2(w)$ that is $u_1 \dots u_k = u'_1 \dots u'_k$. In other words the two morphisms must be equivalent on the range of T^2.

This last problem is called the *morphism equivalence problem*. It asks, given two morphisms and a language, whether the images of any word of the language by the first and the second morphism are equal.

Lemma 25. *T is functional iff Π_1 and Π_2 are equivalent on range(T^2).*

Plandowski showed that the morphism equivalence problem is decidable in PTIME when the language is a context-free language given by a grammar in Chomsky normal form.

Theorem 26 ((Plandowski [47]).)**.** *Let Φ_1, Φ_2 be two morphisms from Σ to Δ and a CFG G (alternatively a* NPA*), testing whether Φ_1 and Φ_2 are equivalent on $L(G)$ can be done in* PTIME*.*

By Proposition 16, one can construct in PTIME a pushdown automaton that recognizes the range of T^2. Then, applying Theorem 26 directly yields a PTIME procedure for testing the functionality of T.

Theorem 27 (Functionality)**.** *Functionality of* VPT *is decidable in* PTIME*.*

5.4. Equivalence and inclusion

The *equivalence, resp. the inclusion, problems* for transducers, asks whether two VPT T_1 and T_2 satisfy $R(T_1) = R(T_2)$ (resp. $R(T_1) \subseteq R(T_2)$).

These problems are already undecidable for finite-state transducers [25]. Therefore, they are also undecidable for VPT.

Let us consider now functional VPT: given two functional VPT, T_1 and T_2, they are equivalent, resp. T_1 is included into T_2, if and only if their union is functional and they have the same domains, resp. the domain T_1 is included into the domain of T_2. The domains being VPLs, testing their equivalence or the language inclusion is ExpTime-c [5]. Testing the equivalence or the inclusion of the domains is easier when the VPT are deterministic, it can be done in PTime [5]. Therefore both procedures, *i.e.* equivalence or inclusion testing, can be done in PTime when the VPT are deterministic. If we assume that both transducers are total, then obviously we do not have to test the equality or inclusion of their domains. Therefore, in that case equivalence and inclusion of their transductions can also be tested in PTime.

Theorem 28 (Equivalence and inclusion of fVPT)**.** *The inclusion and the equivalence problems are :*

- *undecidable for* VPT
- ExpTime-c *for functional* VPT
- *in* PTime *for either* deterministic VPT *[54] or* total *functional* VPT

5.5. k-valuedness

Given some (fixed) integer $k \in \mathbb{N}$ and a VPT T, T is said to be *k-valued* if u has at most k images by T for all $u \in \Sigma^*$. The *k-valuedness problem* asks, given as input a VPT T, whether it is k-valued.

The k-valued transductions are a slight generalization of functional transductions. In the case of finite state transducers, k-valued and functional transducers share the interesting characteristic to have decidable equivalence and inclusion problems. In this section we show that deciding k-valuedness for VPT can be done in co-NPTime. We, however, leave open the question of deciding the equivalence or the inclusion of k-valued VPT.

The idea is to generalize the construction for deciding functionality presented in the previous section. This previous construction is based on the square and the morphism equivalence problem. We use here the k-power and the multiple morphisms equivalence problem [29], which generalizes the morphism equivalence problem to k morphisms, instead of 2.

The main tool of the decision procedure is the class of bounded reversal counter automata introduced by Ibarra [33]. We proceed in three steps. First, we have shown in [21] that the procedure of [29] for deciding emptiness of such machines can be executed in co-NPTime. Second, as a direct consequence, we show that the procedure for deciding the multiple morphism equivalence problem in [29] can be executed in co-NPTime. Finally, we show how to use this last result to decide the k-valuedness problem for VPT in co-NPTime.

5.5.1. Reversal-Bounded Counter Automata

A *counter automaton* is a finite state or pushdown automaton, called the underlying automaton, augmented with one or several counters. On an input letter, a counter automaton behaves like the underlying automaton, but on each transition it can also increment or decrement one or several of its counters. Moreover, the transitions can be guarded by zero tests on some of its counters, that is, transitions can be triggered conditionally depending on whether the current value of some counters is zero.

Definition 29. *Let $m \in \mathbb{N}$, a* counter automaton *with m counters is a tuple $C = (A, inc, dec, zero)$, where A is an automaton, either* NFA *or* PA*, δ is the transition relation of A, inc, dec and zero are functions from δ to $\{0,1\}^m$.*

A run ρ of a counter automaton with m counters is a run of its underlying automaton, that is a sequence of transitions $\rho = t_1 \dots t_k \in \delta^*$, such that there exist for each $0 < i \leq m$ some $c_1^i, c_2^i \dots c_k^i \in \mathbb{N}$ which are the successive values of the i-th counter such that the following conditions hold:

$$c_{j+1}^i = c_j^i + \pi_i(inc(t_j) - dec(t_j))$$

$$c_j^i = 0 \text{ if } \pi_i(zero(t_j)) = 1$$

The first condition states that the counter is incremented or decremented according to the value of the functions *inc* and *dec* for the transition, and the second condition asks that the value of the i-th counter be 0 when i-th component of the value of the *zero* function for the transition is 1. Note that the values c_j^i of the counters are natural numbers, this implies that a transition that induces a decrease on a counter cannot occur when this counter value is 0. A run is accepting if it is an accepting run of the underlying automaton. As usual, the language $L(A)$ of A is the set of words with at least one accepting run.

Counter automata are powerful and thus some essential problems are undecidable: with just two counters, emptiness is undecidable [43].

We say that a counter is in *increasing mode*, resp. *decreasing mode*, if the last operation on that counter was an increment, resp. a decrement. *Reversal-Bounded Counter Automata*, introduced by O. Ibarra in [33], are counter automata such that each counter can alternate at most a predefined number of times between increasing and decreasing modes. This restriction on the behavior of the counters yields decidability of the emptiness problem.

Definition 30. *Let $r, m \in \mathbb{N}$, an r-reversal m-counter automaton A is an m-counter automaton such that in any run (accepting or not) each counter alternates at most r times. We denote the set of r-reversal m-counter automaton by* $\mathsf{RBCA}(r, m)$*, resp.* $\mathsf{RBCPA}(r, m)$*, when the underlying automaton is an* NFA*, resp. when it is an* NPA*.*

For finite state automata with reversal-bounded counters emptiness is decidable in PTime (if r and m are fixed)[27]. The proof is developed in two steps. First they prove that there is a witness of non-emptiness if and only if there is one of polynomial size. Then, testing emptiness can be done with an NLogSpace algorithm similar to the one for testing emptiness of standard NFA.

Interestingly, adding a pushdown store (in other words, a stack) to these machines, does not break decidability of the emptiness. A procedure for deciding emptiness of pushdown automata with reversal-bounded counters was first published in [33]. In [21], we show that this procedure can be executed in co-NPTime (for a fixed number of counters). For that we use a recent result that permits the construction in linear time of an existential Presburger formula representing the Parikh image of a context-free language [56]. Our result has been improved in [28] where it is shown that the co-NPTime upper bound still holds even if the number of counters is not fixed, for a slightly more general model (where increment/decrement/comparisons of arbitrary constants are allowed). Moreover, [28] shows that this co-NP upper bound is also a lower bound. Even if the model of [28] is slightly more general (a translation to our setting requires an exponential blow-up), the proof of this lower bound can be easily adapted to our setting [53]. Therefore we have the following theorem.

Theorem 31 ([33, 21, 28])**.** *Let $m, r \in \mathbb{N}$ be fixed integers. The emptiness problem for r-reversal m-counters pushdown automata is co-*NPComplete*. This result still holds if m is not fixed [28].*

5.5.2. Multiple Morphism Equivalence Problem

The *multiple morphism equivalence problem* is a generalization of the morphism equivalence problem (see Theorem 26). It asks, given a set of pairs of morphisms and a language L, whether for any word u in L there is always at least one of the pairs of morphisms such that the image of u by both morphisms are equal.

Definition 32 (Multiple Morphism Equivalence Problem)**.** *Given ℓ pairs of morphisms $(\Phi_1, \Psi_1), \dots, (\Phi_\ell, \Psi_\ell)$ from Σ^* to Δ^* and a language $L \subseteq \Sigma^*$, $(\Phi_1, \Psi_1), \dots, (\Phi_\ell, \Psi_\ell)$ are equivalent on L if for all $u \in L$, there exists i such that $\Phi_i(u) = \Psi_i(u)$.*

The multiple morphism equivalence problem was proved to be decidable in [29] on any class of languages whose Parikh images are effectively semi-linear. In the case of context-free languages, we show that it can be decided in co-NPTime. It is in fact a consequence of the co-NPTime bound on the emptiness test for RBCPA.

Theorem 33. *Let $\ell \in \mathbb{N}$ be fixed. Given ℓ pairs of morphisms and a pushdown automaton A, testing whether they are equivalent on $L(A)$ can be done in co-*NPTime.

Proof. In order to prove this theorem, we briefly recall the procedure of [29] in the particular case of pushdown machines. In order to decide the morphism equivalence problem of ℓ pairs of morphisms on a CFL L, the idea is to construct an RBCPA$(1, 2\ell)$ that accepts the language $L' = \{w \in L \mid \Phi_i(w) \neq \Psi_i(w) \text{ for all } i\}$. Clearly, $L' = \varnothing$ iff the morphisms are equivalent on L. We construct a pushdown automaton A' augmented with 2ℓ counters $c_{11}, c_{12}, \ldots, c_{\ell 1}, c_{\ell 2}$ that simulates A on the input word and counts the lengths of the outputs by the 2ℓ morphisms. For all $i \in \{1, \ldots, \ell\}$, A' guesses some position p_i where $\Phi_i(w)$ and $\Psi_i(w)$ differ: it increments in parallel (with ε-transitions) the counters c_{i1} and c_{i2} and non-deterministically decides to stop incrementing after p_i steps. Then when reading a letter $a \in \Sigma$, the two counters c_{i1} and c_{i2} are decremented by $|\Phi_i(a)|$ and $|\Psi_i(a)|$ respectively (by possibly several transitions as the counters can be incremented by at most one at a time). When one of the counter reaches zero, A' stores the letter associated with the position (in the state). At the end of the computation, for all $i \in \{1, \ldots, \ell\}$, one has to check that the two letters associated with the position p_i in $\Phi_i(w)$ and $\Psi_i(w)$ are different. If n is the number of states of A and m is the maximal length of an image of a letter $a \in \Sigma$ by the 2ℓ morphisms, then A' has $O(n \cdot m \cdot |\Delta|^{2\ell})$ states, because for all 2ℓ counters one has to store the letters at the positions represented by the counter values. This is polynomial as ℓ is fixed. Note that the resulting machine is 1-reversal bounded (counters start at zero and are incremented up to a position in the output word, and then are decremented to zero).

We can conclude the proof by combining this result with the co-NPTime procedure for testing the emptiness of 1-reversal counter pushdown automata (Theorem 31). □

5.5.3. Deciding k-valuedness

We extend the squaring construction we used for deciding functionality. We define a notion of k-power for the class of VPT, where $k \in \mathbb{N}$. The k-power of T simulates k parallel executions on the same input. Note that this construction is possible for VPT (but not for general PTs) because two runs along the same input have necessarily the same stack behavior. Let $T = (A, \Omega)$ be a VPT with $A = (Q, I, F, \Gamma, \delta)$ and $O_T = \Omega(\delta)$ the set of outputs of the transitions of T. As this set is finite, it, and all its power O_T^k, can be regarded as an alphabet. The k-power of T is a VPT from words over Σ to words over $(O_T)^k$ defined as follows.

Definition 34 (k-Power)**.** *The k-power of T, denoted T^k, is the* VPT *defined from Σ to $(O_T)^k$ by $T^k = (A^k, \vec{\Omega})$, where $\vec{\Omega}$ is the morphism from δ^k (k times the $\otimes$-product of δ) to $(O_T)^k$ defined by $\vec{\Omega}(t_1, \ldots, t_k) = (\Omega(t_1), \ldots, \Omega(t_k))$.*

The transducer T^k can be viewed as a machine that simulates k copies of T. In other words let $u = a_1 \ldots a_n \in \Sigma^*$, we have:

$$T^k \models (p_1, \ldots p_k) \xrightarrow{u/(v_{11}, \ldots v_{1k}) \ldots (v_{n1}, \ldots v_{nk})} (q_1, \ldots q_k)$$

if and only if

$$T \models p_i \xrightarrow{u/v_{1i} \ldots v_{ni}} q_i \quad \text{for all } 1 \leq i \leq k$$

The outputs of T^k are sequences of k-tuples on O_T. We consider now the morphisms that realize the projection of the outputs of T_k onto one of its k components. For all $k \geq 0$, we define the morphisms $\Pi_1, \ldots, \Pi_k$ as follows:

$$\begin{array}{rccc} \Pi_i & : & (O_T)^k & \to & \Sigma^* \\ & & (u_1, \ldots, u_k) & \mapsto & u_i \end{array}$$

Clearly, we obtain the following equivalence:

Lemma 35. *T is k-valued iff $(\Pi_i, \Pi_j)_{1 \leq i \neq j \leq k+1}$ are equivalent on $range(T^{k+1})$.*

Proof. First note that $dom(T) = dom(T^k)$.

Suppose that T is k-valued and let $\beta \in range(T^{k+1})$. We prove that there exists $i \neq j$ such that $\Pi_i(\beta) = \Pi_j(\beta)$. There exists $u \in dom(T^{k+1})$ such that $T^{k+1}(u) = \beta$. The word β can be decomposed as $\beta = (v_1^1, \ldots, v_{k+1}^1) \ldots (v_1^n, \ldots, v_{k+1}^n)$ where $(v_1^j, \ldots, v_{k+1}^j) \in (O_T)^{k+1}$ for all j. Since T is k-valued, there exists $i \neq j$ such that $v_i^1 \ldots v_i^n = v_j^1 \ldots v_j^n$. By definition of Π_i and Π_j, $\Pi_i(u) = v_i^1 \ldots v_i^n = w_j^1 \ldots w_j^n = \Pi_j(u)$.

Conversely, suppose that T is not k-valued. Therefore there exists $u \in dom(T) = dom(T^{k+1})$ and $v_1, \ldots, v_{k+1}$ all pairwise different such that $v_i \in T(u)$ for all i. By definition of T^{k+1}, each v_i can be decomposed as $v_i^1 \ldots v_i^n$ such that $\beta = (v_1^1, \ldots, v_{k+1}^1) \ldots (v_1^n, \ldots, v_{k+1}^n) \in T^{k+1}(u)$. By definition of $\Pi_1, \ldots, \Pi_{k+1}$, we get $\Pi_i(\beta) = v_i \neq v_j = \Pi_j(\beta)$ for all $i \neq j$. Therefore, Π_i and Π_j are not equivalent on $range(T^{k+1})$. □

By Proposition 16, the language $range(T^k)$ is a context-free language. By Theorem 33, as $range(T^k)$ is represented by an automaton of polynomial size if k is fixed, we get:

Theorem 36 (k-valuedness)**.** *Let $k \geq 0$ be fixed. The problem of deciding whether a* VPT *is k-valued is in co-*NPTime*.*

We conjecture that the equivalence of two k-valued VPT is decidable, but leave it as an open problem. For k-valued FST, it is known to be decidable, and the procedure relies on the existence of a decomposition of any k-valued FST as a finite union of functional FST [58, 50]. This decomposition is based on a notion of delay between output of runs on the same input. Generalising this notion of delay to VPT is a challenging open problem, and would likely lead to an effective decomposition result for k-valued VPT, as finite unions of functional VPT. We now show that equivalence of finite unions of functional VPT is decidable. As a consequence, a decomposition result for VPT would give decidability of equivalence for k-valued VPT.

Lemma 37. *Given $T, T_1, \ldots, T_k$ functional* VPT*, on can decide whether $R(T) \subseteq \bigcup_{i=1}^k R(T_i)$?*

Proof. We reduce this problem to the emptiness of a reversal-bounded counter pushdown automaton A, such that $L(A) \neq \varnothing$ iff $R(T) \not\subseteq \bigcup_{i=1}^k R(T_i)$. A simulates a run of T on an input word $u \in \Sigma^*$, and in parallel, check that for all $i \in \{1, \ldots, k\}$, either $u \notin dom(T_i)$ or $T_i(u) \neq T(u)$. The construction of A is similar to the k-valuedness test. A will guess a partition $P_1 \uplus P_2$ of $\{1, \ldots, k\}$, will check that (1) for all $i \in P_1$, $u \notin dom(T_i)$, by simulating the run of the complement of the underlying VPA of T_i (assumed to be deterministic), and (2) for all $i \in P_2$, $T_i(u) \neq T(u)$, by using two counters c_i and d_i, initialised non-deterministically to the same value, that point a mismatch in the output of $T(u)$ and $T_i(u)$ respectively at the c_i-th position, or that exhibit different output length. As for k-valuedness test, the counter have only 1-reversal. □

This lemma gives immediately the following theorem:

Theorem 38. *The following problem is decidable: given k_1 functional* VPT *$T_1, \ldots, T_{k_1}$, and k_2 functional* VPT *$G_1, \ldots, G_{k_2}$, does $\bigcup_{i=1}^{k_1} R(T_i) = \bigcup_{i=1}^{k_2} R(G_i)$ hold ?*

6. Well-Nested VPT

The main results of the previous section showed that functionality and, more generally, k-valuedness are decidable for VPT. As a consequence, the equivalence and inclusion problem for functional VPT is decidable. All these problems are undecidable for pushdown transducers.

On the other hand, contrary to the class of NFT, the class of VPT is not closed under composition and the type checking problem against VPA is undecidable. A closer look at the reason for this non-closure and undecidability results points to an interesting subclass of VPT. Both of these weaknesses of the model can be solved by adding a 'visibly' constraint on the output of the VPT.

Indeed, the non-closure under composition can be viewed as a consequence of the fact that the stack of the two involved VPT are not synchronized. The stack of the first VPT is guided by the input word,

while the stack of the second is guided by the output of the first VPT. Constraining the VPT with some synchronization between its input and its output yields closure under composition.

A similar observation holds for the undecidability of the type checking. In this problem three stacks are involved: the stack of the input VPA, the one of the involved transducer and the one of the output VPA. The stack of the input VPA and the one of the VPT are both synchronized by the input word. On the other hand, the stack of the output VPA is guided by the output of the VPT.

In this section we introduce the class of well-nested VPT (wnVPT). These transducers are VPT that produce words over a *structured output* alphabet $\Delta = (\Delta_c, \Delta_i, \Delta_r)$. The "visibly" restriction for wnVPT asks the nesting level of the input and the output words to be synchronized, that is the nesting level of the output just before reading a call (on the input) must be equal to the nesting level of the output just after reading the matching return (on the input). This simple syntactic restriction yields a subclass of VPT that is closed under composition and has a decidable type checking problem against VPLs. Recently, these positive results have been extended to more general classes of VPT suitable to describe tree-to-tree transformations [49].

6.1. Definition

A well-nested VPT is a VPT with a notion of synchronization between the input and the output. This synchronization is enforced with the following syntactic restriction: for all call and return transitions that use the same stack symbol, the concatenation of the output word of the call transition with the output word of the return transition must be a well-nested word. Moreover the output word of any internal transition must be a well-nested word.

Definition 39. *Let $T = (A, \Omega)$ be a* VPT *with $A = (Q, I, F, \Gamma, \delta)$ and Ω a morphism from δ into Δ^*, T is* well-nested *if:*

- *For all $\gamma \in \Gamma$, all $t_c = (q, c, \gamma, q') \in \delta_c$ and all $t_r = (p, r, \gamma, p') \in \delta_r$, $\Omega(t_c)\Omega(t_r)$ is well-nested.*
- *For all $t \in \delta_i$, then $\Omega(t)$ is well-nested.*
- *For all $t_\perp = (q, r, \perp, q') \in \delta_r$, we have that $\Omega(t_\perp)$ is call-matched (all call symbols have a matching return).*

We denote by wnVPT the class of well-nested VPT.

Example 40. *In the example of Figure 8 the transducer is well-nested. Indeed, one can check that the outputs associated with γ_1 are $u_1 = c_1c_2r_2c_2a$ for the call and $v_1 = r_1ar_2$ for the return, their concatenation forms a well-nested word: $u_1v_1 = c_1c_2r_2c_2ar_1ar_2 \in \Sigma^*_{wn}$. Similarly, the outputs associated with γ_2 are $u_2 = c_1r_1a$ for the call and $v_2 = \varepsilon$ for the return, their concatenation produces a well-nested word: $u_2v_2 = c_1r_1a \in \Sigma^*_{wn}$. Finally, the lone internal transition produces a well-nested word: $c_1r_1 \in \Sigma^*_{wn}$.*

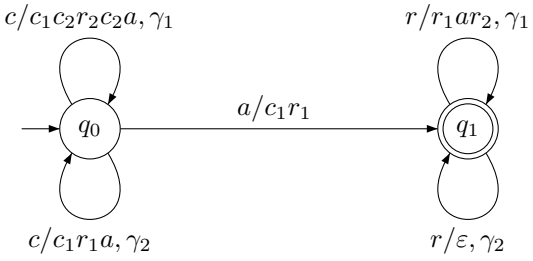

Figure 8: A wnVPT from $\Sigma_c = \{c\}, \Sigma_r = \{r\}$, and $\Sigma_i = \{a\}$ to $\Delta_c = \{c_1, c_2\}, \Delta_r = \{r_1, r_2\}$, and $\Delta_i = \{a\}$.

Note that this restriction implies that, in contrast with the class of VPT, some NFT have no equivalent wnVPT. Indeed, the transduction that maps all input letter to a single call symbol c cannot be defined by a wnVPT, while it easily can with an NFT.

Moreover, there are context-free languages that are not the range of any wnVPT. For example the context-free language $\{r^n c^n \mid n \geq 0\}$, where $r \in \Delta_r$ and $c \in \Delta_c$, is not the range of any wnVPT. But the context-free language $\{a^n b^n \mid n \geq 0\}$, where $a, b \in \Delta_i$, is not a VPL, but it is the range of the VPT transduction $\{(c^n r^n, a^n b^n) \mid n \geq 0\}$, which is easily defined by a wnVPT. Furthermore, any VPL, and therefore any regular language, is, trivially, the range of some wnVPT. So the class of languages formed by the range of wnVPT lies in between VPLs and CFLs.

The next proposition states that the image of a well-nested word by a wnVPT is always a well-nested word. This is easily proved by induction on the length of the input word (note that on a well-nested word, a VPT does not use any return transition on empty stack).

Proposition 41. *Let $T \in$ wnVPT, $u \in \Sigma^*_{wn}$ and $v \in T(u)$, then $v \in \Sigma^*_{wn}$. Moreover, for all $u \in \Sigma^*_{wn}$, $v \in \Sigma^*$, and $q, q' \in Q$ if $(q, \bot) \xrightarrow{u|v} (q', \bot)$ is a run of T then $v \in \Sigma^*_{wn}$.*

In the next sections, we show that the class of well-nested VPT is closed under composition and has a decidable type checking problem.

6.2. Composition

The closure under composition of wnVPT follows from the following observation. The well-nested property of wnVPT states that two transitions, one that pushes a stack symbol and another that pops the same symbol, produce two words whose concatenation forms a well-nested word. This well-nested property of a wnVPT carries over to sequences of transitions. This is formalized in the following lemma and is proved easily by induction on the length of the sequence.

Lemma 42. *Let T be a* wnVPT *and v, v' be two words such that $vv' \in \Sigma^*_{wn}$. For any two runs $(q_1, \bot) \xrightarrow{v/w} (q'_1, \sigma)$, $(q_2, \sigma) \xrightarrow{v'/w'} (q'_2, \bot)$ of T, then $ww' \in \Sigma^*_{wn}$.*

Given two well-nested VPT T_1 and T_2, consider a run of T_1 over an input word and a run of T_2 over the output of the run of T_1. Because T_1 is a VPT, its stack content before reading a well-nested sub-word u is the same than after reading u, that is the stack is unchanged after reading a well-nested word. Moreover, because T_1 is well-nested, the output of the sub-run of T_1 over u is a well-nested word v. Therefore, the stack content of T_2 is the same before reading v than after reading it. In other words, the stacks of T_1 and T_2 are synchronized on well-nested input words (when performing the composition of T_1 with T_2), but they do not necessarily have the same height. As a consequence, it is possible to simulate both stacks with a single stack. This is achieved by considering as a single move the sequence of moves of the second stack on the sub-word v (output of the transition of T_1).

The wnVPT T that implements the composition of T_1 and T_2 simulates both VPT as follows. A state of T is a pair of states of T_1 and T_2. On input letter a, there is a transition from (q_1, q_2) to (q'_1, q'_2) if there is a transition from state q_1 to q'_1 in T_1 producing v, and if there is a sequence of transitions ρ of T_2 from q_2 to q'_2 over v, all this assuming the stack content of each VPT does allow the moves. The output of this transition is the output of ρ. The content of the stack of T consists of pairs (γ, σ) where γ is the stack symbol of T_1 and σ is the sequence of stack symbols produced or consumed when T_2 reads v.

We now formally provide the construction and prove its correctness.

Theorem 43 (Closure by composition)**.** *The class of* wnVPT *is closed under composition. Moreover, given two* wnVPT*, one can construct in* PTIME *a* wnVPT *defining their composition.*

Proof. Wlog we suppose that the alphabets have no internal symbol (they can be simulated by a call directly followed by a return, *e.g.* if $a \in \Sigma_i$ then replace it by $c_a r_a \in \Sigma_c \Sigma_r$). To simplify the presentation, we do not give the details of the constuction of return transitions on the empty stack. Using the third item of

Definition 39, one can prove that when a VPT T_2 is applied on the ouputs produced by some wnVPT T_1, then each time the stack of T_1 is empty, so is that of T_2.

Consider two wnVPT $T_1 = (A_1, \Omega_1)$ and $T_2 = (A_2, \Omega_2)$. We note $A_i = (Q_i, I_i, F_i, \Gamma_i, \delta_i)$ for $i \in \{1, 2\}$. We build a wnVPT T such that

$$\forall u, w \in \Sigma^*, w \in T(u) \iff \exists v \in \Sigma^*,\ v \in T_1(u) \wedge w \in T_2(v)$$

Let $\Gamma' = \{\sigma \in \Gamma_2^* \mid \exists q, q' \in Q_2, v \in \Omega_1(\delta_1) : (q, \bot) \xrightarrow{v/.} (q', \sigma)\}$. We define $T = (A, \Omega)$ with $A = (Q, I, F, \Gamma, \delta)$ where $Q = Q_1 \times Q_2$, $I = I_1 \times I_2$, $F = F_1 \times F_2$, and $\Gamma = \Gamma_1 \times \Gamma'$. The transition relation δ is defined as follows:

Calls: Let $c \in \Sigma_c$. Then $(q_1, q_2) \xrightarrow{c/w,(\gamma,\sigma)} (q_1', q_2') \in \delta$ if there exists $v \in \Sigma^*$ such that $(q_1, \bot) \xrightarrow{c/v} (q_1', \gamma)$ is a run of T_1 and $(q_2, \bot) \xrightarrow{v/w} (q_2', \sigma)$ is a run of T_2.

Returns: Let $r \in \Sigma_r$. Then $(q_1, q_2) \xrightarrow{r/w,(\gamma,\sigma)} (q_1', q_2') \in \delta$ if there exists $v \in \Sigma^*$ such that $(q_1, \gamma) \xrightarrow{r/v} (q_1', \bot)$ is a run of T_1 and $(q_2, \sigma) \xrightarrow{v/w} (q_2', \bot)$ is a run of T_2.

First, we establish that T is well nested. Let consider two transitions $(q_1, q_2) \xrightarrow{c/w,(\gamma,\sigma)} (q_1', q_2')$ and $(q_3, q_4) \xrightarrow{r/w',(\gamma,\sigma)} (q_3', q_4')$ in δ. Let us show that $ww' \in \Sigma_{wn}^*$. By definition of δ, there exist words $v, v' \in \Sigma^*$ such that we have $T_2 \models (q_2, \bot) \xrightarrow{v/w} (q_2', \sigma)$, $T_2 \models (q_4, \sigma) \xrightarrow{v'/w'} (q_4', \bot)$ and $vv' \in \Sigma_{wn}^*$. Therefore by Lemma 42 ww' is well-nested. So we have shown that T is a wnVPT.

Second, we prove that T recognizes the composition of the transducers T_1 and T_2. Let $u, w \in \Sigma^*$, we prove by induction on the length of u that there exists a run ρ on u in T, starting in an initial configuration, and producing w as output, if and only if there exists a word $v \in \Sigma^*$, a run ρ_1 on u in T_1, starting in an initial configuration and producing v as output, and a run ρ_2 on v in T_2, starting in an initial configuration and producing w as output. Moreover, if ρ_i ends in some configuration $(q_i, \gamma_1^i \dots \gamma_{n_i}^i) \in Q_i \times \Gamma_i^*$, and ρ ends in configuration $((p_1, p_2), (\gamma_1, \sigma_1) \dots (\gamma_k, \sigma_k)) \in Q \times \Gamma^*$, then we can require that $p_1 = q_1$, $p_2 = q_2$, $\gamma_1 \dots \gamma_k = \gamma_1^1 \dots \gamma_{n_1}^1$ and $\sigma_1 \dots \sigma_k = \gamma_1^2 \dots \gamma_{n_2}^2$.

The base case $u = \varepsilon$ is trivial. For the induction, we distinguish two cases whether the last symbol of u is a call or a return:

- if $u = u' \cdot c$ with $c \in \Sigma_c$, then the result follows from the definition of the transitions of T: there exists in T a push transition on c of the form $((q_1, q_2), \bot) \xrightarrow{c/w} ((q_1', q_2'), (\gamma, \sigma))$ if and only if there exists a word v and two runs in T_1 and T_2 of the form $(q_1, \bot) \xrightarrow{c/v} (q_1', \gamma)$ and $(q_2, \bot) \xrightarrow{v/w} (q_2', \sigma)$.
- if $u = u' \cdot r$ with $r \in \Sigma_r$, then again by definition of the transitions of T, there exists in T a pop transition on r of the form $((q_1, q_2), (\gamma, \sigma)) \xrightarrow{r/w} ((q_1', q_2'), \bot)$ if and only if there exists a word v and two runs in T_1 and T_2 of the form $(q_1, \gamma) \xrightarrow{r/v} (q_1', \bot)$ and $(q_2, \sigma) \xrightarrow{v/w} (q_2', \bot)$. By the induction property, we have that the stacks are equal on each component, and thus the same pop transitions can be triggered.

By definition, the run ρ is accepting if and only if both runs, ρ_1 and ρ_2 are accepting. This concludes the proof of the correctness of our construction. □

Remark 44. *One can observe that the above construction, taking as input two transducers T_1 and T_2, is still valid if T_2 is not well-nested. In this case, the result of the construction is a* VPT *that correctly implements the composition of T_1 and T_2.*

6.3. Type Checking against VPL

In this section we show that the type checking problem is decidable for wnVPT.

Theorem 45 (Type Checking)**.** *The type checking problem for* wnVPT *against* VPA *is* ExpTime-c*. It is in* PTime *if the* VPA *constraining the output language is deterministic.*

Proof. For the ExpTime-hard part, first note that we can construct a wnVPT T_{id} whose domain is the set of well-nested words on the structured alphabet Σ and whose relation is the identity relation. Given any VPA A_1, A_2, we have that $T_{id}(L(A_1)) \subseteq L(A_2)$ if and only if $L(A_1) \subseteq L(A_2)$. This later problem is ExpTime-c (See Theorem 10).

To prove it is in ExpTime, note that given a wnVPT T and two VPA A_1, A_2, we have $T(L(A_1)) \subseteq L(A_2)$ iff $T|_{A_1} \circ Id_{\overline{A_2}} = \varnothing$, where $Id_{\overline{A_2}}$ is a wnVPT that defines the identity function over the complement of $L(A_2)$ (it is of exponential size in the size of A_2 if A_2 is non-deterministic, and of polynomial size if A_2 is deterministic). By Theorem 43, one can construct a wnVPT T' for $T|_{A_1} \circ Id_{\overline{A_2}}$ in ExpTime (and in PTime if A_2 is deterministic). Finally, the emptiness test can be done in PTime in the size of T'.

□

7. Comparison to Tree Transducers

In this section we investigate the relative expressive power of wnVPT and several classes of tree transducers.

Relation between trees, tree automata, structured words and VPA have been thoroughly investigated in [1]. Unranked trees can be encoded as well-nested words. The linearization of a tree is such an encoding, it corresponds to a depth-first left-to-right traversal of the tree. This encoding corresponds to the common interpretation of XML documents as trees [32]. A node n is encoded by a call (*i.e.* an opening tag) and its matching return (*i.e.* closing tag), and the encoding of the subtree rooted at n lies in between this call and return.

Through linearization, wnVPT can be used to define unranked tree transformations. An unranked tree transformations is definable by a wnVPT if the relation on words, induced by the linearization of the input and output trees, is. We compare the expressive power of wnVPT on unranked trees to several models of tree transducers.

With the aim to apprehend the expressive power of wnVPT with regards to unranked tree transductions, we first define the unranked tree transducers (UTT). They are *macro forest transducers* without parameters [46]. We show that wnVPT are strictly less expressive than UTT. As a consequence, wnVPT are also less expressive than macro forest transducers, a class that includes UTT.

A second model of unranked tree transformations is formed by the uniform tree transducers [38], a model inspired by XSLT [13]. We show that the expressiveness of this model is incomparable with the one of wnVPT.

Finally, a popular trick for defining unranked tree transformations, is to first encode trees as binary trees and then use a binary tree transducer, such as the top-down tree transducers (TDTT) [14]. We recall the first-child next-sibling encoding of unranked trees into binary trees, and we show that TDTT, with this encoding, are incomparable to wnVPT. However, both classes are included into macro tree transducers, the binary (or ranked) tree version of macro forest transducers.

7.1. Unranked Trees and Hedges - Ranked Trees

Unranked Trees and Hedges. We define *unranked trees* and *hedges* (that are sequences of unranked trees) over an alphabet Σ. They are defined as terms over the binary operator $\cdot$ (concatenation of an unranked tree with an hedge), the constant 0 (the empty hedge) and the alphabet Σ. They are generated by the following grammar:

$$h := 0 \mid t \cdot h \qquad t := f(h) \text{ where } f \in \Sigma$$

We denote by $\mathcal{H}_\Sigma$ and $\mathcal{U}_\Sigma$ the set of hedges and unranked trees respectively. We identify the tree t and the hedge $t \cdot 0$, so that $\mathcal{U}_\Sigma \subset \mathcal{H}_\Sigma$. For all $f \in \Sigma$, we may write f instead of $f(0)$. When it is clear from the

context we may also omit the operator $\cdot$. Note that all hedges are of the form $t_1 \cdot \cdots \cdot t_k \cdot 0$ for $k \geq 0$ and $t_1, \ldots, t_k \in \mathcal{U}_\Sigma$. We extend $\cdot$ to concatenation of hedges in the following manner:

$$(t_1 \cdot \cdots \cdot t_k \cdot 0) \cdot (t'_1 \cdot \cdots \cdot t'_{k'} \cdot 0) \;=\; t_1 \cdot \cdots t_k \cdot t'_1 \cdot \cdots \cdot t'_{k'} \cdot 0$$

The *height* of an hedge is defined inductively as: $\mathsf{height}(0) = 0$, $\mathsf{height}(t \cdot h) = \max(\mathsf{height}(t), \mathsf{height}(h))$ and $\mathsf{height}(f(h)) = 1 + \mathsf{height}(h)$ where $f \in \Sigma$.

Example 46. *The following tree $f(f(a\ a\ a)b\ b\ b)$ is an unranked tree. Its root is labelled by f and has 4 children labelled f, b, b, and b respectively. The b's are leaves while, the f child has 3 children labelled a.*

Ranked Trees. A ranked tree is a tree whose symbols have a predefined fixed number of children. Unranked trees can be encoded by ranked trees via, for example, a first-child next-sibling encoding (fcns).

A *ranked alphabet* is a pair (Σ, ar) where Σ is an alphabet and ar is a function that associates with each letter its arity: $\mathsf{ar} : \Sigma \to \mathbb{N}$. For $k \in \mathbb{N}$, the set of letters of arity k is denoted by Σ_k.

The set $\mathcal{T}^r_\Sigma$ of *ranked trees* over Σ is defined as the set of terms generated by the following grammar :

$$t := a \mid f(t_1, \ldots t_k) \quad \text{where } a \in \Sigma_0 \text{ and } f \in \Sigma_k$$

A *binary tree* is a ranked tree whose symbols have arity 0 or 2. The set of binary trees is denoted by $\mathcal{T}^2_\Sigma$.

First-Child Next-Sibling Encoding. Unranked trees and hedges can be encoded into binary trees. We present the first-child next-sibling (fcns) encoding [14]. Each node of the unranked tree is encoded by a node of the binary tree. Consider the binary tree encoding t_1 of the unranked tree t_2 and suppose that the node n_1 in t_1 encodes the node n_2 in t_2. The left child of n_1 encodes the first child of n_2, while the right child of n_1 encodes the next sibling of n_2. if n_2 has no child (resp. no sibling) the left (resp. right) child of n_2 is labelled with the special symbol $\perp \notin \Sigma$.

The encoding is defined on hedges by the function $\mathrm{fcns} : \mathcal{H}_\Sigma \to \mathcal{T}^2_\Sigma$ such that:

$$\begin{array}{lcl} \mathrm{fcns}(0) & = & \perp \\ \mathrm{fcns}(f(h) \cdot h') & = & f(\mathrm{fcns}(h), \mathrm{fcns}(h')) \end{array}$$

Example 47 (Complete Binary Trees)**.** *Let t_n denote the complete binary tree of height n over the alphabet $\Sigma = \{f, a\}$ where f, resp. a, has arity 2, resp. 0. We have $t_0 = a$, $t_1 = f(a, a)$, $t_2 = f(f(a, a), f(a, a))$, and more generally $t_n = f(t_{n-1}, t_{n-1})$. Their* fcns *encoding is defined inductively as:*

$$\begin{array}{lcl} \mathrm{fcns}(t_0) & = & a(\perp, \perp) \\ \mathrm{fcns}(t_1) & = & f(a(\perp, a(\perp, \perp)), \perp) \\ \mathrm{fcns}(t_n) & = & f(f(\mathrm{fcns}(t_{n-2}), \mathrm{fcns}(t_{n-1})), \perp) \end{array}$$

Therefore the height of $\mathrm{fcns}(t_n)$ *is equal to* $2 + \mathsf{height}(\mathrm{fcns}(t_{n-1}))$ *that is*

$$\mathsf{height}(\mathrm{fcns}(t_n)) = 2 * \mathsf{height}(t_n)$$

Example 48 (Hedge)**.** *The hedge $a_1 a_2 \ldots a_n$ is encoded as*

$$a_1(\perp, a_2(\perp, a_3(\perp, \ldots a_n(\perp, \perp) \ldots)))$$

Note that the height of the hedge is 1 while the height of its fcns *encoding is equal to the number of nodes in the hedge.*

7.2. Unranked Tree Transductions.

An unranked tree transduction is a relation R_u between unranked trees, *i.e.* it is a subset of $\mathcal{U}_\Sigma \times \mathcal{U}_\Sigma$. We present some examples that we use later to separate classes of transductions.

Example 49 (Yield)**.** *The yield transduction transforms a hedge into the hedge containing its leaves. For example, the tree $f(g(ag(bc))def)$ is mapped to the hedge $abcdef$.*

Example 50 (Duplicate)**.** *The* duplicate *transduction duplicates a subtree as follows:*

$$R_2 = \{f(t) \to f(tt) \mid t \in \mathcal{U}_\Sigma \wedge f \in \Sigma\}$$

Example 51 (Swap)**.** *The* swap *transduction swap two subtrees. It is defined as $R_{swap} = \{f(t_1t_2) \to f(t_2t_1) \mid t_1, t_2 \in \mathcal{U}_\Sigma \wedge f \in \Sigma\}$.*

Example 52 (Odd)**.** *The* odd *transduction transforms trees of the form $f(a^n)$ by replacing the odd, resp. even, leaves with leaves labelled a, resp. b. It is defined as $R_{odd} = \{f(a^{2n}) \to f((ab)^n) \mid n \geq 0\}$.*

A ranked tree transduction is a relation R_r between ranked trees, *i.e.* it is a subset of $\mathcal{T}^r_\Sigma \times \mathcal{T}^r_\Sigma$. Through first-child next-sibling encoding, a ranked tree transduction R_r can be used to define an unranked tree transduction R_u as $R_u = \text{fcns}^{-1} \circ R_r \circ \text{fcns}$.

7.3. wnVPT *on Unranked Trees*

Given an alphabet Σ, recall that the tagged alphabet $\hat{\Sigma}$ is the structured alphabet defined as: $\hat{\Sigma} = (\overline{\Sigma}, \Sigma, \underline{\Sigma})$ where $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$ is the set of call symbols, and $\underline{\Sigma} = \{\underline{a} \mid a \in \Sigma\}$ is the set of return symbols.

Hedges to words. The linearization function $\mathsf{lin} : \mathcal{H}_\Sigma \to \hat{\Sigma}^*$ transforms an hedge into a word. It corresponds to a depth-first left-to-right traversal of the tree. It is defined as follows:

$$\mathsf{lin}(0) = \varepsilon \qquad \mathsf{lin}(t \cdot h) = \mathsf{lin}(t)\mathsf{lin}(h) \qquad \mathsf{lin}(f(0)) = f \qquad \mathsf{lin}(f(h)) = \overline{f}\ \mathsf{lin}(h)\ \underline{f} \quad (h \neq 0)$$

We extend the function lin to sets of hedges as usual. Let $S \subset \mathcal{H}_\Sigma$, then $\mathsf{lin}(S) = \{\mathsf{lin}(t) \mid t \in S\}$.

Example 53. *The word $\overline{f}\ \overline{g}a\ \overline{g}bc\underline{g}\ \underline{g}\ def\ \underline{f}$ is the linearization of the tree $f(g(ag(bc))def)$.*

wnVPT *on trees.* With the linearization of trees, we can now define unranked tree transduction with wnVPT. A wnVPT T implements a tree transduction $TT \subseteq \mathcal{U}_\Sigma \times \mathcal{U}_\Sigma$ if for all $t \in \mathcal{U}_\Sigma$ we have:

$$R(T)(\mathsf{lin}(t)) = \mathsf{lin}(TT(t))$$

Example 54 (Yield)**.** *The yield transduction of Example 49 is easily defined by a* wnVPT *that deletes all calls and returns and copy any internal to the output.*

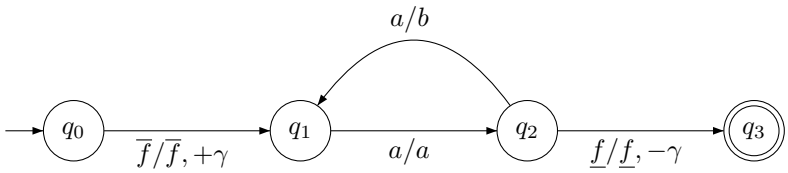

Figure 9: A wnVPT T_{odd} on $\Sigma_c = \{\overline{f}\}$, $\Sigma_r = \{\overline{r}\}$, $\Sigma_i = \{a, b\}$.

Example 55 (Odd)**.** *The odd transduction of Example 52 maps trees of the form $f(a^{2n})$ onto trees of the form $f((ab)^n)$, the linearization gives:*

$$\overline{f}a\ a \ldots a\ a\underline{f} \to \overline{f}a\ b \ldots a\ b\underline{f}$$

It can be defined by the wnVPT *T_{odd} of Figure 9.*

Clearly, the duplicate and the swap transductions of Example 50 and 51 cannot be implemented by wnVPT. Indeed, one can easily check that the linearization of the range of R_2 (the duplicate transduction) is not context-free. While regarding R_{swap}, with a classical pumping argument one can prove that it is not definable by a wnVPT.

7.4. Transducers for Unranked Trees

Unranked Tree Transducers. we present here a model of unranked tree transducers (UTT) that run directly on unranked trees and more generally, on hedges. They are defined as parameter-free *macro forest transducers* (MFT) [46].

With UTT, an hedge, $f(h)h'$, is rewritten in a top down manner. The root node of the left most tree, $f(h)$, is transformed into an hedge over the output alphabet according to an initial rule. Some of the leaves of this output hedge are insertion points for the result of the recursive application of the rules on either the hedge h of the children or on the hedge h' containing the siblings.

Let Σ be an (unranked) alphabet. An *unranked tree transducer* (UTT) over Σ is a tuple $T = (Q, I, \delta)$ where Q is a set of states, $I \in Q$ is a set of initial states and δ is a set of rules of the form:

$$q(0) \to 0 \qquad\qquad q(f(x_1) \cdot x_2)) \to r$$

where $q \in Q$, $h \in \mathcal{H}_\Sigma$, $f \in \Sigma$, and r is a right-hand side generated by the following grammar:

$$\begin{array}{rcl} r & ::= & (q, x) \mid 0 \mid ur \mid rr \\ u & ::= & f(r) \end{array}$$

with $q \in Q$ and $x \in \{x_1, x_2\}$.

The semantics of T is defined via mappings $[\![q]\!] : \mathcal{H}_\Sigma \to 2^{\mathcal{H}_\Sigma}$ for all $q \in Q$ as follows:

$$\begin{array}{lcl} [\![q]\!](0) & = & \{0\} \\ [\![q]\!](f(h) \cdot h') & = & \bigcup_{q(f(x_1)\cdot x_2)\to r} [\![r]\!]_{[x_1 \mapsto h, x_2 \mapsto h']} \end{array}$$

where $[\![.]\!]_\rho$ for a valuation $\rho : \{x_1, x_2\} \to \mathcal{H}_\Sigma$ is defined by:

$$\begin{array}{lcl} [\![0]\!]_\rho & = & \{0\} \\ [\![(q, x)]\!]_\rho & = & [\![q]\!](\rho(x)) \\ [\![ur]\!]_\rho & = & \{t \cdot h \mid t \in [\![u]\!]_\rho, h \in [\![r]\!]_\rho\} \\ [\![r_1 r_2]\!]_\rho & = & \{h_1 \cdot h_2 \mid h_1 \in [\![r_1]\!]_\rho, h_2 \in [\![r_2]\!]_\rho\} \\ [\![f(r)]\!]_\rho & = & \{f(h) \mid h \in [\![r]\!]_\rho\} \end{array}$$

The transduction of a UTT $T = (Q, I, \delta)$ is defined as $R(T) = \{(t, t') \mid \exists q \in I,\ t' \in [\![q]\!](t)\}$.

Example 56 (Yield)**.** *The yield transduction of Example 49 is defined by a* UTT $T = (\{q_0, q\}, \{q\}, \delta)$ *with the following rules:*

$$\begin{array}{lcll} q(f(x_1) \cdot x_2) & \to & q(x_1) \cdot q(x_2) & \text{for all } f \in \Sigma \\ q(f(x_1) \cdot x_2) & \to & f(q_0(x_1)) \cdot q(x_2) & \text{for all } f \in \Sigma \\ q_0(0) & \to & 0 & \end{array}$$

where the procedure (or state) q guesses (with non-determinism) whether the first tree, with its root labelled f, has children or not. If it has no child then it outputs f and applies q_0 (which only accepts 0). Otherwise, it recursively calls q on the hedge of the children.

Example 57 (Swap)**.** *The swap transduction of Example 51 is defined by a* UTT $T = (Q, \{q_0\}, \delta)$ *with $Q = \{q_0, q, q', q_\perp\}$ and the following rules:*

$$\begin{array}{lcll} q_\perp(0) & \to & 0 & \\ q_{id}(0) & \to & 0 & \\ q_0(f(x_1) \cdot x_2) & \to & f(q_{swap}(x_1)) \cdot q_\perp(x_2) & \text{for all } f \in \Sigma \\ q_{swap}(f(x_1) \cdot x_2) & \to & q'(x_2) \cdot f(q_{id}(x_1) & \text{for all } f \in \Sigma \\ q_{id}(f(x_1) \cdot x_2) & \to & f(q_{id}(x_1)) \cdot q_{id}(x_2) & \text{for all } f \in \Sigma \\ q'(f(x_1) \cdot x_2) & \to & f(q_{id}(x_1) \cdot q_\perp(x_2)) & \text{for all } f \in \Sigma \end{array}$$

where the procedure (or state) q_0 ensure that the transduction only accepts trees, q_{swap} perform the swap operation, q' ensures that the root node has exactly 2 children, and q_{id} performs the identity transduction.

Example 58 (Duplicate and Odd)**.** *The duplicate and odd transduction of Example 50 and Example 52 can be defined with a transducer similar to the* UTT *of the previous example.*

UTT can in fact simulate wnVPT. Therefore they are clearly strictly more expressive than wnVPT, as duplicate and swap examples are definable by UTT but not by wnVPT.

Proposition 59. UTT *are strictly more expressive than* wnVPT*.*

Proof. Let Σ be an alphabet and $T = (Q, I, F, \{\gamma\}, \delta)$ be a wnVPT over $\hat{\Sigma}$ that implements an unranked tree transduction. We construct an equivalent UTT $T' = (Q', I', \delta')$. We let $Q' = Q \times Q$ and $I' = I \times F$. We informally explain the rules of T' on an example. Let $h = f(h_1) \cdot h_2$ be an hedge. Suppose that there exists a run of T' on h from a pair of states (p_1, q_1). It means that there exists a run of T on $\mathsf{lin}(h)$ from the configuration $(p_1, \bot)$ to $(q_1, \bot)$. When reading f, T' must apply a call transition of T on $\overline{f}$ of the form $t_c = (p_1, \overline{f}, \gamma, p_1')$ for some p_1' together with a return transition $t_r = (q_1', \underline{f}, \gamma, p_1'')$, for some p_1''. Therefore T' has to guess the transitions to apply and continue its evaluation of h_1 from the state (p_1', q_1') and the evaluation of h_2 from the state (p_1'', q_1). If h_2 is empty, then T' requires that $p_1'' = q_1$. The rules of T' are formally defined as follows, for all $p, p', p'', q, q' \in Q$:

$$
\begin{array}{lcl}
(p,p)(0) & \to & 0 \\
(p,q)(f(x_1)\cdot x_2) & \to & \mathsf{lin}^{-1}(uyv)\cdot(p'',q)(x_2) \text{ if } \left\{\begin{array}{l} t_c = (p,\overline{f},\gamma,p') \in \delta_c \\ t_r = (q',\underline{f},\gamma,p'') \in \delta_r \\ u = \Omega(t_c) \\ v = \Omega(t_r) \\ y = (p',q')(x_1) \end{array}\right.
\end{array}
$$

Note that $\mathsf{lin}^{-1}(uyv)$ (where y is considered as an internal symbol) is well-defined as T is a well-nested VPT that implements an unranked tree transduction. Therefore the symbols of u and v are well-matched and form an unranked tree. □

A UTT $T = (Q, I, \delta)$ is *non-duplicating*, resp. *order-preserving*, if for all rules $q(f(x_1) \cdot x_2)) \to r \in \delta$, neither x_1 nor x_2 occur more than one time in r, resp. x_2 never occurs before x_1 in r in a depth-first left-to-right order. In fact non-duplicating and order-preserving UTT are still more expressive than wnVPT. Restricting further this class, one may consider non-duplicating and order-preserving UTT such that the rule $r ::= rr$ is replaced by $r ::= rq(x_2)$ where $q \in Q$. In other words, wnVPT are UTT such that the result of the transformation of the siblings hedge must be directly concatenated to the result of the transformation of the leftmost tree. Let us denote UTT_S this class.

Proposition 60. UTT_S *and* wnVPT *are equally expressive.*

Proof. The inclusion of wnVPT into UTT_S follows from the proof of Proposition 59 where the built UTT is actually a UTT_S.

Conversely, for a UTT_S $T = (Q, I, \delta)$, we define a wnVPT $T' = (A', \Omega)$ over $\hat{\Sigma}$ with $A' = (Q', I', F', \Gamma', \delta')$ such that $Q' = Q$, $I' = F$, $F' = \{q \mid q(0) \to 0 \in \delta\}$, $\Gamma' = \delta$ and δ' is given by:

- $t_c = (q, \overline{f}, q_1, f(x_1, x_2) \to u_1 q_1(x_1) u_2 q_2(x_2))$ belongs to δ_c with $\Omega(t_c) = u_1$
- $t_r = (q', \underline{f}, q_2, f(x_1, x_2) \to u_1 q_1(x_1) u_2 q_2(x_2))$ belongs to δ_r with $\Omega(t_r) = u_2$ if $q'(0) \to 0$ is a rule from δ

It can be shown by induction that for any well-nested word u over $\hat{\Sigma}$, there exists a run in T' from $(q, \bot)$ to $(q', \bot)$ with $q'(0) \to 0$ is a rule from δ producing the word v iff in T, $v \in [\![\mathsf{lin}^{-1}(u)]\!]$. □

Uniform Tree Transducers.. Inspired by XSLT transformations, Neven and Martens [38] have introduced *uniform tree transducers* (UUTT) as a simple model of unranked tree transductions. They form a strict subclass of UTT. They also operate in a top-down manner but a rule is always applied to all children uniformly. Their work mainly investigates the complexity of the type checking problem parameterized by several restrictions on the transformations [39, 40, 41].

A *uniform tree transducer* over an alphabet Σ is a tuple $T = (Q, q_0, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and δ is a set of rules of the form $q(a) \to r$ where r is inductively defined by:

$$r \ ::= \ 0 \mid ur \qquad u \ ::= \ a \mid q \mid f(r)$$

where $a, f \in \Sigma$, $q \in Q$.

The semantics of T is defined via mappings $[\![q]\!] : \mathcal{H}_\Sigma \to 2^{\mathcal{H}_\Sigma}$ for all $q \in Q$ as follows:

$$\begin{array}{lcl} [\![q]\!](0) & = & \{0\} \\ [\![q]\!](t_1 \cdot \cdots \cdot t_k) & = & \{t'_1 \cdot \cdots \cdot t'_k \mid t'_i \in [\![q]\!](t_i),\ 1 \le i \le k\} \\ [\![q]\!](f(h)) & = & \bigcup_{q(f) \to r \in \delta} [\![r]\!]_h \end{array}$$

where $[\![.]\!]_h$ for an hedge $h = t_1 \cdot \cdots \cdot t_k \in \mathcal{H}_\Sigma$ is defined by:

$$\begin{array}{lcl} [\![0]\!]_h & = & \{0\} \\ [\![ur]\!]_h & = & \{t'h' \mid t' \in [\![u]\!]_h,\ h' \in [\![r]\!]_h\} \\ [\![a]\!]_h & = & \{a\} \\ [\![q]\!]_h & = & [\![q]\!](h) \\ [\![f(r)]\!]_h & = & \{f(h') \mid h' \in [\![r]\!]_h\} \end{array}$$

The transduction of $T = (Q, q_0, \delta)$ is defined as $R(T) = \{(t, t') \mid t' \in [\![q_0]\!](t)\}$.

As these transducers have the ability to duplicate subtrees, they can clearly define some transductions that cannot be defined by wnVPT. On the other hand, the transduction R_{odd} of Example 52 cannot be defined by a uniform tree transducer as different transformations are applied on odd and even children respectively.

Proposition 61. wnVPT *and* UUTT *are incomparable.*

Macro Forest Transducers.. We defined the UTT as a restriction of macro forest transducers (MFT), they are MFT without parameters. Therefore, MFT are strictly more expressive than wnVPT. They are actually also strictly more expressive than UTT [46].

7.5. Transformations by means of Ranked Tree Transducers.

In this section we investigate the expressive power of ranked (binary) tree transducers (that run on fcns encodings of unranked trees) to define unranked tree transductions.

Top-down binary tree transducers.. Let (Σ, ar) be a ranked alphabet with constant and binary symbols only. A top-down binary tree transducer (TDTT) over Σ [14] is a tuple $T = (Q, I, \delta)$ where Q is a set of states, $I \subseteq Q$ is a set of initial states and δ is a set of rules of the form

$$q(a) \to t \qquad\qquad q(f(x_1, x_2)) \to r$$

where $q \in Q$, $t \in \mathcal{T}^r_\Sigma$, $a \in \Sigma_0$, $f \in \Sigma_2$, and r is a term generated by the following grammar:

$$r \ ::= \ a \mid (q, x) \mid f(r, r)$$

where $a \in \Sigma_0$, $f \in \Sigma_2$, $q \in Q$, and $x \in \{x_1, x_2\}$.

The semantics of T is defined via mappings $[\![q]\!] : \mathcal{T}^r_\Sigma \to 2^{\mathcal{T}^r_\Sigma}$ for all $q \in Q$ as follows:

$$\begin{array}{lcl} [\![q]\!](a) & = & \{t \mid q(a) \to t \in \delta\} \\ [\![q]\!](f(t_1, t_2)) & = & \bigcup_{q(f(x_1,x_2)) \to r} [\![r]\!]_{[x_1 \mapsto t_1, x_2 \mapsto t_2]} \end{array}$$

where $[\![.]\!]_\rho$ for a valuation $\rho : \{x_1, x_2\} \to \mathcal{T}^T_\Sigma$ is inductively defined by:

$$\begin{array}{lcl} [\![a]\!]_\rho & = & \{a\} \\ [\![(q,x)]\!]_\rho & = & [\![q]\!](\rho(x)) \\ [\![f(r_1,r_2)]\!]_\rho & = & \{f(t_1,t_2) \mid t_1 \in [\![r_1]\!]_\rho, t_2 \in [\![r_2]\!]_\rho\} \end{array}$$

The transduction of a TDTT $T = (Q, I, \delta)$ is defined as

$$R(T) = \{(t, t') \mid \exists q \in I,\ t' \in [\![q]\!](t)\}$$

The height of the image of a tree t by a TDTT is linearly bounded by the height of t.

Proposition 62. *Let T be a* TDTT*. There exists $k \in \mathbb{N}$, such that for all $(t, t') \in R(T)$, $h(t') < k\, h(t)$ holds.*

We recall that TDTT can define unranked tree transductions since a TDTT T on $\Sigma \cup \{\bot\}$ defines the unranked tree transduction $T_{\mathcal{U}}$ given by $\text{fcns}^{-1} \circ R(T) \circ \text{fcns} = T_{\mathcal{U}}$.

Example 63 (Yield)**.** *The yield transduction maps every tree onto the hedge formed by its leaves. Yield is not definable by a* TDTT*. Indeed, the* fcns *encoding of an hedge $a_1 \dots a_n$ has height n. Consider the complete binary tree of height n. The height of its* fcns *encoding is proportional to n. However it has 2^n leaves. Therefore the height of the* fcns *encoding of its yield is 2^n. According to Proposition 62 this transduction cannot be defined by a* TDTT*.*

The yield example shows that wnVPT can define transduction that are not definable with TDTT. On the other hand, because TDTT can duplicate and swap parts of the input, they can define some transductions that wnVPT cannot.

Proposition 64. TDTT *and* wnVPT *are incomparable.*

Macro Tree Transducers.. We say that, contrary to TDTT, wnVPT have the ability to concatenate hedges (and so do UTT). This is the reason why TDTT cannot define the yield transduction, while wnVPT can.

We have already mentionned MFT; they turn out to be a slight generalization of macro tree transducers (MTT) [19]. MFT can be viewed as macro tree transducers (MTT) with the ability to concatenate hedges. However, thanks to parameters, MTT have also the possibility to express hedge concatenation (for the first-child-next-sibling encoding).

A transduction is linear size increase when the size of every output tree is linearly bounded by the size of the corresponding input tree. For linear size increase transduction, the class of functional macro tree transductions is closed under composition [37]. Moreover, any MFT transduction can be obtained as the composition of two MTT [46]. Therefore, for linear size increase functional unranked tree transductions the class of macro forest and macro tree transducers are equivalent. Moreover, the linear size increase macro tree transductions are exactly the MSO definable functional transductions [18]. Functional wnVPT transductions are linear size increase (as there is no duplication), therefore MSO transductions and macro tree transductions subsume functional wnVPT transductions.

7.6. Summary

In this section, we summarize on Fig. 10 the relative expressiveness of the classes we have considered.

UTT are strictly more expressive than TDTT. However, if the derivation $r ::= rr$ is disallowed in the definition of the right-hand sides of UTT, we obtain a class which is equivalent to TDTT with the fcns encoding of unranked trees. Indeed, a rule $q(f(x_1) \cdot x_2) \to r$ of a UTT corresponds to a rule $q(f(x_1, x_2)) \to \text{fcns}(r)$ of a TDTT.

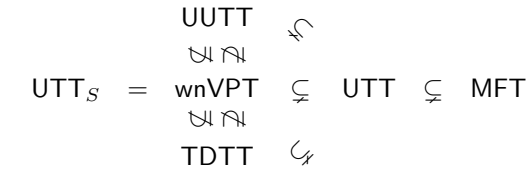

Figure 10: Relative expressiveness of classes of transducers for unranked trees

8. Visibly Pushdown Transducers with Regular Look-ahead

In this section, we investigate the extension of visibly pushdown transducers with visibly pushdown look-ahead ($\mathsf{VPT}_{\mathsf{la}}$). They are visibly pushdown transducers that can check, at any time, whether the longest well-nested subword starting at the current position belongs to a regular (VPL) language. First, we show that $\mathsf{VPT}_{\mathsf{la}}$ are not more expressive than VPT, but are exponentially more succinct. Removing look-aheads, even from a deterministic $\mathsf{VPT}_{\mathsf{la}}$, can be done at the price of adding non-determinism, but only in an unambiguous manner. This closure by regular look-aheads shows the robustness of the class of VPT. We then prove that the class of deterministic $\mathsf{VPT}_{\mathsf{la}}$ coincides exactly with the class of functional VPT, yielding a simple characterization of functional VPT. More generally, we show that any relation definable by a VPT can be uniformized by a function definable by a deterministic $\mathsf{VPT}_{\mathsf{la}}$.

An interesting corollary of these results is that any functional VPT is equivalent to an unambiguous one, a result which was already known [10] for VPT, and also for finite-state transducers [15, 51, 16], and which was extended to k-valued transducers and k-ambiguous finite-state transducers [50].

Finally, we show that while $\mathsf{VPT}_{\mathsf{la}}$ are exponentially more succinct than VPT, checking equivalence or inclusion of functional $\mathsf{VPT}_{\mathsf{la}}$ is, as for VPT, ExpTime-c. Additionally, we derive similar results for visibly pushdown automata with look-ahead.

8.1. Definitions

In order to simplify notations and proofs we suppose that the structured alphabet has no internal symbols, that is $\Sigma = (\Sigma_c, \varnothing, \Sigma_r)$. Indeed, as previously suggested, one can encode an internal symbol with a specific call symbol directly followed by a specific return symbol. Moreover we also assume for simplicity that words are well-nested. More precisely, we suppose that VPA and VPT do not allow return transitions on the empty stack ($\perp$). Furthermore, their stack must be empty in order to accept a word. It is not difficult however to extend the results of this section to the general definitions of VPA and VPT.

Given a word w over Σ we denote by $\mathrm{pref}_{\mathrm{wn}}(w)$ the longest well-nested prefix of w. E.g. $\mathrm{pref}_{\mathrm{wn}}(crc) = cr$, $\mathrm{pref}_{\mathrm{wn}}(crcrrcrr) = crcr$. We define a VPT T with visibly pushdown look-ahead (simply called look-ahead in the sequel) informally as follows. The look-ahead is given by a VPA A. On a call symbol c, T can trigger the look-ahead starting from a chosen initial state p of the VPA. The look-ahead tests membership of the longest well-nested prefix of the current suffix of the input (that starts by the letter c) to $L(A[p])$, where $A[p]$ is the VPA A with a single initial state fixed to be p. If the prefix is in $L(A[p])$ then the transition of T can be fired. When we consider nested words that encode trees, look-ahead corresponds to inspecting the subtree rooted at the current node and all right sibling subtrees (in other words, the current hedge).

A visibly pushdown automaton with look-ahead is a visibly pushdown automaton A and a look ahead VPA B. Each call transition of A is associated with a state of B. A call transition can be triggered only if the longest well-nested prefix starting at the current call position belong to $L(B[q])$.

Definition 65 ($\mathsf{VPA}_{\mathsf{la}}$)**.** *A* VPA *with look-ahead (*$\mathsf{VPA}_{\mathsf{la}}$*) is a triple $A_{la} = (A, B, \xi)$ where A, B are two* VPA *and $\xi : \delta_c^A \to Q^B$ is a function that associates with any call transition of A a state of B.*

Let $u \in \Sigma^*$. A run of A_{la} on $u = a_1 \dots a_l$ is a run $\rho = t_1 \dots t_l \in \delta^A$ of A such that, if $t_k \in \delta_c^A$, then we have $\mathrm{pref}_{\mathrm{wn}}(a_{k+1} \dots a_l) \in L(B[\xi(t_k)])$. The size of a $\mathsf{VPA}_{\mathsf{la}}$ $A_{la} = (A, B, \xi)$ is the size of A plus the size of B. We denote transitions of A_{la} as tuples $(q, \alpha, p, \gamma, q')$, where $t = (q, \alpha, \gamma, q')$ is a transition of A and $p = \xi(t)$. Such transitions are also denoted by $q \xrightarrow{\alpha,p,\gamma} q'$ in the sequel.

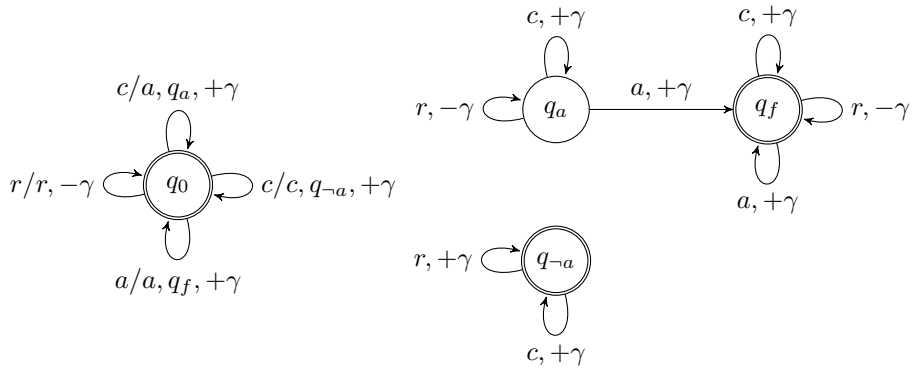

Figure 11: A $\mathsf{VPT}_{\mathsf{la}}$ (left) and its look-ahead (right) on $\Sigma_c = \{c, a\}$ and $\Sigma_r = \{r\}$

A $\mathsf{VPA}_{\mathsf{la}}$ is deterministic when the input symbol, the top of the stack symbol and the look-ahead determine univocally the next transition. For call transitions, this means that the look-aheads must be disjoint, *i.e.* for a given state q and a given call symbol c, the language of the look-aheads associated with all transitions from q with input letter c must be disjoint. Moreover, the look-ahead automaton must be deterministic.

Definition 66 (Deterministic $\mathsf{VPA}_{\mathsf{la}}$)**.** *A $\mathsf{VPA}_{\mathsf{la}}$ $A_{la} = (A, B, \xi)$ is deterministic if*

- *B is deterministic*
- *A is deterministic on return transitions*
- *for all $t_1, t_2 \in \delta_c^A$, if t_1 and t_2 are transitions from the same state and on the same symbol, then $L(B[\xi(t_1)]) \cap L(B[\xi(t_2)]) = \varnothing$.*

Note that deciding whether some $\mathsf{VPA}_{\mathsf{la}}$ is deterministic can be done in PTime. One has to check that for each state q and each call symbol c, the languages of the VPLs guarding the transitions from state q and reading c are *pairwise* disjoint.

Definition 67 ($\mathsf{VPT}_{\mathsf{la}}$)**.** *A visibly pushdown transducer with look-ahead ($\mathsf{VPT}_{\mathsf{la}}$) from some alphabet Σ to some alphabet Δ is a pair $T = (A_{la}, \Omega)$ where $A_{la} = (A, B, \xi)$ is a $\mathsf{VPA}_{\mathsf{la}}$ and Ω is the output morphism define from δ^A to Δ^*, where δ^A is the transition relation of A.*

All the notions and notations defined for VPT, like transduction relation, domain, range, and so on, can be defined similarly for $\mathsf{VPT}_{\mathsf{la}}$.

Transducers with look-ahead are particularly well-suited for defining transductions whose behavior may depend on some part of the input that lies further after the current position in the input word. We now give an example of such a transduction.

Example 68. *Let $\Sigma_c = \{c, a\}, \Sigma_r = \{r\}$ be the call and return symbols of the alphabet. Consider the transductions such that: (i) a and r are mapped to a and r respectively; (ii) c is mapped either to c if no a appears in the longest well-nested word starting at c, and to a if an a appears. E.g. $ccrrarcr$ is mapped to $acrrarcr$, and $cccrrcrcarrr$ to $aacrraraarrr$.*

The $\mathsf{VPT}_{\mathsf{la}}$ T represented in Figure 11 implements this transduction. The look-ahead automaton is depicted on the right, while the transducer in itself is on the left. When starting in state q_a, respectively $q_{\neg a}$, the look-ahead automaton accepts well-nested words that contain an a, respectively do not contain any a. When starting in state q_f it accepts any well-nested word. The transducer rewrites c into a if the well-nested word starting at c contains an a (transition on the top), otherwise it just copy a c (transition on the right). This is achieved using the states q_a and $q_{\neg a}$ of the look-ahead automaton. Other input symbols, i.e. a and r, are just copied to the output (left and bottom transitions).

8.2. Expressiveness

We show in this section that adding look-aheads to VPT does not add expressiveness. In other words, every $\mathsf{VPT}_{\mathsf{la}}$ is equivalent to a VPT. Furthermore, we show that one can effectively construct an equivalent VPT with an unavoidable exponential blow-up.

First, let us present in Example 69 a VPT that is equivalent to the $\mathsf{VPT}_{\mathsf{la}}$ defined in Example 68.

Example 69. *The* VPT T *defines the transduction of Example 68, it is defined by* $Q = \{q, q_a, q_{\neg a}\}$, $I = \{q\}$, $F = Q$, $\Gamma = \{\gamma, \gamma_a, \gamma_{\neg a}\}$ *and* δ *contains the following transitions:*

$$
\begin{array}{llll}
q \text{ or } q_a \xrightarrow{c/a,\gamma} q_a & q \text{ or } q_a \xrightarrow{c/a,\gamma_a} q & q \xrightarrow{c/c,\gamma_{\neg a}} q_{\neg a} & \\
q \text{ or } q_a \xrightarrow{a/a,\gamma} q & q_{\neg a} \xrightarrow{c/c,\gamma_{\neg a}} q_{\neg a} & & \\
q \text{ or } q_{\neg a} \xrightarrow{r/r,\gamma_a} q_a & q \text{ or } q_{\neg a} \xrightarrow{r/r,\gamma} q & q_{\neg a} \xrightarrow{r/r,\gamma_{\neg a}} q_{\neg a} &
\end{array}
$$

The state q_a, resp. $q_{\neg a}$, means that there is, resp. is not, an a in the longest well-nested word that starts at the current position. The state q indicates that there is no constraints on the appearance of a. If T is in state q and reads a c, there are two cases: it outputs an a or a c. If it chooses to output an a, then it must check that an a occurs later. There are again two cases: either T guesses there is an a in the well-nested word that starts just after c and takes the transitions $q \xrightarrow{c/a,\gamma} q_a$, or it guesses an a appears in the well-nested word that starts after the matching return of c, in that latter case it takes the transition $q \xrightarrow{c/a,\gamma_a} q$ and uses the stack symbol γ_a to carry over this information. If on c it chooses to output c, it must check that there is no a later by using the transition $q \xrightarrow{c/a,\gamma_{\neg a}} q_{\neg a}$. Other cases are similar.

Note that the VPT of Example 69 relies heavily on non-determinism. The construction we present replaces look-aheads with non-determinism.

The main challenge when constructing a VPT equivalent to a given $\mathsf{VPT}_{\mathsf{la}}$ is to simulate an unbounded number of look-aheads at once. Indeed, a look-ahead is triggered at each call and is 'live' until the end of the well-nested subword starting at this call. If the input word has height $k \geq 1$, then for a given run of a $\mathsf{VPT}_{\mathsf{la}}$, there might be k simultaneously running look-aheads. For example, on the word $c^k r^k$ there are k running look-aheads after reading the last c, that is, there is one look-ahead for each strictly smaller nesting level. Furthermore, there is another case that might produces many simultaneous running look-aheads. Consider the word $c_1 crcrcr \dots crr_1$, in this case when reading c a new look-ahead is triggered, this look-ahead will run until r_1, therefore, after reading k successive cr there are (at least) k simultaneous running look-aheads. Note that these k look-aheads all started at the same nesting level.

Recall that summaries, that were defined in the context of the determinization of VPA (see [5]), are pairs of states. More precisely, for a given VPA, a pair (p, q) is a summary if there exists a well-nested word w such that $(q, \bot)$ is accessible from $(p, \bot)$ by reading w. In the next theorem, we uses summaries to handle the simulation of look-aheads that started at a strictly less deeper nesting level and a subset construction for those that started at the same nesting level.

Theorem 70. *For any* $\mathsf{VPT}_{\mathsf{la}}$ *(resp.* $\mathsf{VPA}_{\mathsf{la}}$*)* T_{la}*, with n states, one can construct an equivalent* VPT *(resp.* VPA*)* T'*, with $O(n \cdot 2^{n^2+1})$ states. Moreover, if T_{la} is deterministic, then T' is unambiguous.*

Proof. We first prove the result for $\mathsf{VPA}_{\mathsf{la}}$. Let $A_{la} = (A, B, \xi)$ with $A = (Q, I, F, \Gamma, \delta)$ and $B = (Q^{la}, F^{la}, \Gamma^{la}, \delta^{la})$. We construct the VPA $A' = (Q', I', F', \Gamma', \delta')$ as follows (where $Id_{Q^{la}}$ denotes the identity relation on Q^{la}):

- $Q' = Q \times 2^{Q^{la} \times Q^{la}} \times 2^{Q^{la}}$,
- $I' = \{(q_0, Id_{Q^{la}}, \varnothing) \mid q_0 \in I\}$,
- $F' = \{(q, R, L) \in Q' \mid q \in F, L \subseteq F^{la}\}$,
- $\Gamma' = \Gamma \times 2^{Q^{la} \times Q^{la}} \times 2^{Q^{la}} \times \Sigma_c$.

The automaton A' simulates A and its running look-aheads as follows. A state of A' is a triple (q, R, L). The first component is the state of A. The second and third components are used to simulate the running look-aheads. When reading a call c, A' non-deterministically chooses a new look-ahead triggered by A. This look-ahead is added to all running look-aheads that started at the same nesting level. A' ensures that the run will fail if the longest well-nested prefix starting at c is not in the language of the chosen look-ahead. The L component contains the states of all running look-aheads triggered at the current nesting level. The R component is the set of summaries necessary to update the L-component. When reading a call the L component is put onto the stack. When reading a return, A' must check that all look-ahead states in L are final, i.e. A' ensures that the chosen look-aheads are successful.

After reading a well-nested word w if A' is in state (q, R, L), with $q \in Q$, $R \subseteq Q^{la} \times Q^{la}$ and $L \subseteq Q^{la}$, we have the following properties. The pair $(p, p') \in R$ iff there exists a run of B from p to p' on w. If some p'' is in L, there exists a run of a look-ahead that started when reading a call symbol of w at depth 0 which is now in state p''. Conversely, for all look-aheads that started when reading a call symbol of w at depth 0, there exists a state $p'' \in L$ and a run of this look-ahead that is in state p''.

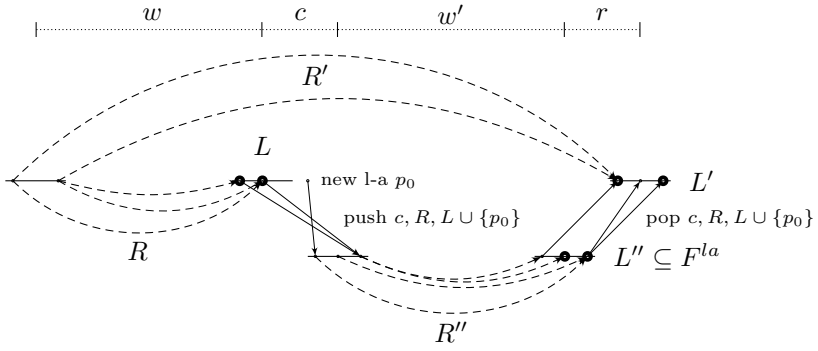

Figure 12: Simulating the look-aheads

Let us consider a word $wcw'r$ for some well-nested words w, w' (depicted on Fig. 12). Assume that A' is in state (q, R, L) after reading w (on the figure, the relation R is represented by dashed arrows and the set L by bold points, and other states by small points). We do not represent the A-component of the states on the figure but rather focus on R and L. The information that we push on the stack when reading c is the necessary information to compute a state (q', R', L') of A' reached after reading $wcw'r$. After reading the call symbol c, we go in state $(q', Id_{Q^{la}}, \varnothing)$ for some q' such that $q \xrightarrow{c,p_0,+\gamma} q' \in \delta_c$, where $p_0 \in Q^{la}$ is the starting state of a new look-ahead. Note that determinism of A is preserved. On the stack we put the tuple $(\gamma, R, L \cup \{p_0\}, c)$ where γ, R, L, p_0, c have been defined before.

Now, suppose that after reading wcw' the VPA A' is in state (q'', R'', L''). It means that A is in state q'' after reading wcw', and $(p, p') \in R''$ iff there exists a run of B from p to p' on w', and L'' is the set of states reached by the look-aheads that started at the same depth as w'. Therefore we first impose that any transition from (q'', R'', L'') reading r must satisfy $L'' \subseteq F^{la}$. Clearly, R' can be constructed from c, R and R''. Finally, L' is a set such that for all $p \in L \cup \{p_0\}$, there exists $p' \in L'$ and a run of A from p to p' on $cw'r$. If such an L' does not exist, there is no transition on r. The set L' can be constructed from $L \cup \{p_0\}$ and R''.

We now define the transitions formally:

1. for all q, R, L, c, γ, we have: $(q, R, L) \xrightarrow{c,(\gamma,R,L\cup\{p_0\},c)} (q', Id_{Q^{la}}, \varnothing) \in \delta'_c$ whenever $q \xrightarrow{c,p_0,\gamma} q' \in \delta_c$,

2. for all $R, L, r, \gamma, q'', R'', L'', q', R', L'$, we have: $(q'', R'', L'') \xrightarrow{r,(\gamma,R,L,c)} (q', R', L') \in \delta'_r$ iff the following conditions hold:

   (i) $q'' \xrightarrow{r,\gamma} q' \in \delta_r$,

(*ii*) $L'' \subseteq F^{la}$,

(*iii*) $R' = \{(p,p') \mid \exists s \xrightarrow{c,\gamma} s' \in \delta_c^{la} \cdot \exists (s',s'') \in R'' \cdot (p,s) \in R$ and $s'' \xrightarrow{r,\gamma} p' \in \delta_r^{la}\}$,

(*iv*) for all $p \in L$, there exist $p' \in L'$, $\gamma \in \Gamma$, $s, s' \in Q^{la}$ such that $(s,s') \in R''$, $p \xrightarrow{c,\gamma} s \in \delta_c^{la}$, $s' \xrightarrow{r,\gamma} p' \in \delta_r^{la}$.

We sketch the proof of correctness of the construction. Let $w \in \Sigma^*$ such that w is a prefix of a well-nested word, *i.e.* it is a word with no unmatched return, but it may have some unmatched calls. We define $sh(w)$ as the longest well-nested suffix of w, we call $sh(w)$ the *subhedge* of w. For instance, if $w = c_1c_2r_2c_3r_3$, then $sh(w) = c_2r_2c_3r_3$. However if $w = c_1c_2$, then $sh(w) = \varepsilon$.

First, one can check (e.g. by induction on the length of w) that the successive computations of the R component of the state ensures that the following property holds: for all words $w \in \Sigma^*$ prefix of a well-nested word, if there is a run of A' from q'_0 to (q,R,L) on w, then for all $p,p' \in Q^{la}$, $(p,p') \in R$ iff there is a run of A on $sh(w)$ from p to p'.

With this last property it is easy to show that the following property also holds: let $w = c_1w_1r_1c_2w_2r_2 \dots c_nw_nr_n$ where all w_i are well-nested. A run of T on w will trigger a new look-ahead at each call c_i, all these look-aheads will still be 'live' until r_n. These look-aheads are simulated by the L component of the state of A'. If there is a run of A on w, it means that all look-aheads accepts the respective remaining suffixes of w, and therefore after reading r_i there are i accepting runs of the previous look-aheads. Suppose that those accepting runs are in the states Q_i after reading r_i. By suitable choices of L-components (A' is non-deterministic on L-components), we can ensure that there is an accepting run of A' such that after reading r_i the L-component of the states is Q_i, for all i. Conversely, if there is an accepting run of A' on w, then one can easily reconstruct accepting runs of the look-aheads.

Next, let show that if A is deterministic, then A' is unambiguous. Indeed, it is deterministic on return transitions. If there are two possible transitions $q \xrightarrow{c,p_1,\gamma_1} q_1$ and $q \xrightarrow{c,p_2,\gamma_2} q_2$ on a call symbol c, as A is deterministic, we know that either the look-ahead starting in p_1 or the look-ahead starting in p_2 will fail. In A', there will be two transitions that will simulate both look-aheads respectively, and therefore at least one continuation of the two transitions will fail as well. Therefore there is at most one accepting computation per input word in A.

Finally, let show that the construction also works for transducers. Let $T = (A_{la}, \Omega)$ where $A_{la} = (A,B,\xi)$ is a $\mathsf{VPA_{la}}$. We construct $T' = (A', \Omega')$ where A' is the VPA obtained as above, and Ω' is obtained thanks to the following observation. Each transition t' of the new VPA A' is associated with one transition t of the original $\mathsf{VPA_{la}}$ A (but several transitions of A' might be associated with the same transition of A). We simply define the output morphism $\Omega'(t')$ as $\Omega(t)$. One can easily check that $R(T) = R(T')$.

□

8.3. Succinctness.

The exponential blow-up in the construction of Theorem 70 is unavoidable. Indeed, it is obviously already the case for finite state automata with regular look-ahead. These finite state automata can be easily simulated by VPA on flat words (in $(\Sigma_c\Sigma_r)^*$, recall that we suppose the alphabet Σ has no internal symbol) in that case the stack is useless. For example, consider for all n the language $L_n = \{vuv \mid |v| = n\}$. One can construct a finite state automaton with regular look-ahead with $O(n)$ states that recognizes L_n. For all $i \leq n$, when reading the i-th letter a_i the automaton uses a look-ahead to test whether the $m-n-i$-th letter is equal to a_i, where m is the length of the word. Without a regular look-ahead, any automaton has to store the first n letters of w in its states, then it guesses the $m-n$-th position and checks that the prefix of size n is equal to the suffix of size n. A simple pumping argument shows that the automaton needs at least $|\Sigma|^n$ states.

Proposition 71 (Succinctness)**.** $\mathsf{VPA_{la}}$ *are exponentially more succinct than* VPA*.*

8.4. Functional VPT *and* $\mathsf{VPT}_{\mathsf{la}}$

While there is no known syntactic restriction on VPT that captures all functional VPT, we show that the class of deterministic $\mathsf{VPT}_{\mathsf{la}}$ captures all functional VPT. Given a functional VPT we construct an equivalent deterministic $\mathsf{VPT}_{\mathsf{la}}$. This transformation yields an exponentially larger transducer.

We prove a slightly more general result: for a given VPT T we construct a deterministic $\mathsf{VPT}_{\mathsf{la}}$ T_{la} such that $R(T_{la})$ is included into $R(T)$ and the domain of T and T_{la} are equal. Clearly, this implies that if T is functional then T_{la} and T are equivalent.

For a given VPT the number of accepting runs associated with a given input might be unbounded. The equivalent $\mathsf{VPT}_{\mathsf{la}}$ has to choose only one of them by using look-aheads. This is done by ordering the states and extending this order to runs. Similar ideas have been used in [17] to show an equivalent result for top-down tree transducers. The main difficulty with VPT is to cope with nesting. Indeed, when the transducer enters an additional level of nesting, its look-ahead cannot inspect the entire suffix but is limited to the current nesting level. When reading a call, choosing (thanks to some look-ahead) the smallest run on the current well-nested prefix is not correct because it may not be possible to extend this run to an accepting run on the entire word. Therefore the transducer has to pass some information from one level to the next level of nesting about the chosen global run. For a top-down tree transducer, as the evaluation is top-down, the transformation of a subtree is independent of the transition choices done in the siblings subtrees.

Theorem 72. *For all* VPT T*, one can construct a deterministic* $\mathsf{VPT}_{\mathsf{la}}$ T_{la} *with at most exponentially many more states such that* $R(T_{la}) \subseteq R(T)$ *and* $dom(T_{la}) = dom(T)$*. If* T *is functional, then* $R(T_{la}) = R(T)$*.*

Proof. We first sketch the construction and then formally define it.

Sketch of the construction We order the states of T and use look-aheads to choose the smallest runs wrt to an order on runs that depends on the structure of the word. Let $T = (A, \Omega)$ be a functional VPT with $A = (Q, q_0, F, \Gamma, \delta)$. Wlog we assume that for all $q, q' \in Q$, all $\alpha \in \Sigma$, there is at most one $\gamma \in \Gamma$ such that $(q, \alpha, \gamma, q') \in \delta$. A transducer satisfying this property can be obtained by duplicating the states with transitions, i.e. by taking the set of states $Q \times \delta$.

We construct an equivalent deterministic $\mathsf{VPT}_{\mathsf{la}}$ $T' = ((A', B, \xi), \Omega')$, where (A', B, ξ) is a deterministic $\mathsf{VPA}_{\mathsf{la}}$ with $A' = (Q', q_0, F', \Gamma', \delta')$ and

- $Q' = \{q_0\} \cup Q^2$,
- $F' = F \times Q$ if $q_0 \notin F$ otherwise $F' = (F \times Q) \cup \{q_0\}$.
- $\Gamma' = \Gamma \times Q \times Q$.

The look-ahead automaton B is defined later.

Before defining δ' formally, let us explain it informally. There might be several accepting runs on an input word w, each of them producing the same output, as T is functional. To ensure determinism, when T' reads one symbol it must choose exactly one transition, the look-ahead are used to ensure that the transition is part of an accepting run. The idea is to order the states by a total order $<_Q$ and to extend this order to runs. The look-ahead will be used to choose the next transition of T that has to be fired, so that the choice will ensure that T follows the smallest accepting run on w. However the look-ahead can only visit the current longest well-nested prefix, and not the entire word, so it can not, on its own, check that the run on this well-nested prefix is compatible with a complete accepting run on the whole input word. Therefore the "parent" of the call c has to pass some information about the global run to its "child" c. In particular, when T' is in state (q, q') for some state q', it means that T is in state q and the state reached after reading the last return symbol of the longest-well nested current prefix must be q'.

Consider a word of the form $w = c_1 w_1 r_1 w_2 c_3 w_3 r_3$ where w_i are well-nested, this word is depicted on Fig. 13. Suppose that, before evaluating w, T' is in state (q_1, q_3). It means that the last transition T has to fire when reading r_3 has q_3 as a target state. When reading the call symbol c_1, T' uses a look-ahead to determine the smallest triple of states (q'_1, q'_2, q_2) such that there exists a run on w that starts in q_1 and such that after reading c_1 it is in state q'_1, before reading r_1 it is in state q'_2, after reading r_1 it is in state

q_2 and after reading r_3 it is in state q_3. Then, T' fires the call transition on c_1 that with source and target states q_1 and q'_1 respectively (it is unique by hypothesis), put on the stack the states (q_2, q_3) and passes to w_1 (in the state) the information that the chosen run on w_1 terminates by the state q'_2, i.e. it goes to the state (q'_1, q'_2). (see Fig. 13). On the figure, we do not explicit all the states and anonymous components are denoted by _. When reading r_1, T' pops from the stack the tuple (γ, q_2, q_3) and therefore knows that the transition to apply on r_1 has target state q_2 and the transition to apply on r_3 has target state q_3. Then it passes q_3 to the current state.

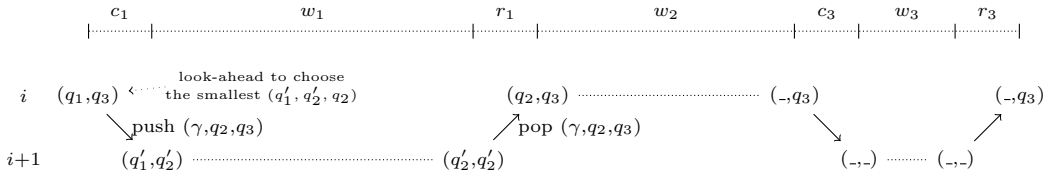

Figure 13: From VPT to deterministic $\mathsf{VPT}_{\mathsf{la}}$.

When the computation starts in q_0, we do not know yet what return transition has to be fired at the end of the hedge. This case can be easily treated separately by a look-ahead on the first call symbol that determine the smallest 4-tuple of states (q_1, q'_2, q_2, q_3) which satisfies the conditions described before, but to simplify the proof, we assume that the VPT accepts only words of the form cwr, where w is well-nested, so that one only needs to consider triples of states.

Formal construction of T' We define the transition relation formally. For all states $q_1, q'_1, q'_2, q_2, q_3 \in Q$, it is easy to define a VPA $A_{q_1,q'_1,q'_2,q_2,q_3}$ whose size is polynomial in the size of T that accepts a word w iff it is of the form $c_1 w_1 r_1 w_3$ where w_1, w_3 are well-nested and there exists a run of T on w that starts in state q_1 and is in state q'_1 after reading c_1, in state q'_2 before reading r_1, in state q_2 after reading r_1 and in state q_3 after reading w_3. Note that if $w_3 = \varepsilon$ then if $q_3 \neq q_2$, then $w \notin L(A_{q_1,q'_1,q'_2,q_2,q_3})$. We denote by $\overline{A_{q_1,q'_1,q'_2,q_2,q_3}}$ the complement of $A_{q_1,q'_1,q'_2,q_2,q_3}$.

Let $<$ be a total order on states, extended lexicographically to tuples. We let $B_{q_1,q'_1,q'_2,q_2,q_3}$ be a VPA with initial state $p_{q_1,q'_1,q'_2,q_2,q_3}$ that defines the language:

$$L(B_{q_1,q'_1,q'_2,q_2,q_3}) = L(A_{q_1,q'_1,q'_2,q_2,q_3}) \cap \bigcap_{\substack{(s_1,s'_2,s_2) \in Q^3 \\ (s_1,s'_2,s_2) < (q_1,q'_2,q_2)}} L(\overline{A_{q_1,s_1,s'_2,s_2,q_3}})$$

Such a VPA exists as VPA are closed by intersection and complement. Its size however may be exponential in $|Q|$. We define the look-ahead VPA as the union of all those VPA, $B = \biguplus B_{q_1,q'_1,q'_2,q_2,q_3}$. We now define the call and return transitions of A', as well as the mappings Ω' and ξ, as follows. For all $c \in \Sigma_c, r \in \Sigma_r, \gamma \in \Gamma, q_1, q'_1, q'_2, q_3, q \in Q, u \in \Sigma^*$:

- if $t = (q_1 \xrightarrow{c,\gamma} q'_1) \in \delta_c^A \ \wedge\ \Omega(t) = u$, then

$$t' = (q_1, q_3) \xrightarrow{c,\ (\gamma,q_2,q_3)} (q'_1, q'_2) \in \delta_c^{A'} \ \wedge\ \Omega'(t') = u \ \wedge\ \xi(t') = p_{q_1,q'_1,q'_2,q_2,q_3}$$

- if $t = q_0 \xrightarrow{c,\gamma} q'_1 \in \delta_c^A \ \wedge\ \Omega(t) = u$, then

$$t' = q_0 \xrightarrow{c,\ (\gamma,q_3,q_3)} \in \delta_c^{A'} \ \wedge\ \Omega'(t') = u \ \wedge\ \xi(t') = p_{q_0,q'_1,q'_2,q_3,q_3}(q'_1, q'_2)$$

- if $t = (q'_2 \xrightarrow{r,\gamma} q_2) \in \delta_r^A \ \wedge\ \Omega(t) = u$, then

$$t' = (q'_2, q) \xrightarrow{r,(\gamma,q_2,q_3)} (q_2, q_3) \in \delta_r^{A'} \ \wedge\ \Omega'(t') = u$$

Let us show now that the transducer T' is deterministic: return transitions are fully determined by the states q'_2, q_2, q_3 and the input letter r (by our first assumption there is at most one transition in T from q'_2 to q_2). For call transitions, suppose that from (q_1, q_3) there are two possible look-aheads from states $p_{q_1,q'_1,q'_2,q_2,q_3}$ and $p_{q_1,s'_1,s'_2,s_2,s_3}$. By definition of the look-aheads, we have $L(B[p_{q_1,q'_1,q'_2,q_2,q_3}]) \cap L(B[p_{q_1,s'_1,s'_2,s_2,s_3}]) = \varnothing$. Moreover, there cannot be two transitions with the same look-ahead as transitions are fully determined by $q_1, q_3, q_2, q'_2, q'_1$ (there is at most one call transition by our assumption with source and target states q_1 and q'_1 respectively). A simple analysis of the complexity shows that the look-ahead A has exponentially many more states than T (the exponentiation comes from the complement in the definition of $B_{q_1,q'_1,q'_2,q_2,q_3}$ and from the intersection).

□

This construction, followed by the construction of Theorem 70, allows one to recover a result that was already shown in [11]:

Corollary 73. *For all functional* VPT T*, one can effectively construct an equivalent unambiguous* VPT T'*.*

The resulting unambiguous VPT might be doubly exponentially larger, while the construction of [11], which is more direct, is singly exponential.

8.5. Decision Problems

In this section, we study the decision problems for $\mathsf{VPA}_{\mathsf{la}}$ and $\mathsf{VPT}_{\mathsf{la}}$. In particular, we prove that while being exponentially more succinct than VPA, resp. VPT, the equivalence and inclusion of $\mathsf{VPA}_{\mathsf{la}}$ and functional $\mathsf{VPT}_{\mathsf{la}}$ remains decidable in ExpTime, as equivalence of VPA and functional VPT.

Theorem 74. *The emptiness problem for* $\mathsf{VPA}_{\mathsf{la}}$*, resp.* $\mathsf{VPT}_{\mathsf{la}}$*, is* ExpTime-c*, even when the look-aheads are deterministic.*

Proof. The upper bound for $\mathsf{VPA}_{\mathsf{la}}$ is obtained straightforwardly by first removing the look-aheads (modulo an exponential blow-up) and then checking the emptiness of the equivalent VPA (in PTime). Checking emptiness of a $\mathsf{VPT}_{\mathsf{la}}$ amounts to check emptiness of its domain, which is a $\mathsf{VPA}_{\mathsf{la}}$.

For the lower-bound, we reduce the problem of deciding emptiness of the intersection of n deterministic top-down tree automata, which is known to be ExpTime-c [14].

Given n deterministic top-down binary tree automata $T_1, \ldots, T_n$ over an alphabet Δ, one can construct in linear-time n *deterministic* VPA $A_1, \ldots, A_n$ that define the same languages as $T_1, \ldots, T_n$ respectively, modulo the natural encoding of trees as nested words over the structured alphabet $\tilde{\Delta} = \{c_a \mid a \in \Delta\} \uplus \{r_a \mid a \in \Delta\}$ [23] The encoding corresponds to a depth-first left-to-right traversal of the tree. For instance, $enc(f(f(a,b),c)) = c_f c_f c_a r_a c_b r_b r_f c_c r_c r_f$.

We now construct a $\mathsf{VPA}_{\mathsf{la}}$ A over the alphabet $\Sigma = \tilde{\Delta} \uplus \{c_i \mid 1 \leq i \leq n\} \cup \{r_i \mid 1 \leq i \leq n\}$ such that A is empty iff $\bigcap_i L(T_i) = \varnothing$. The language of A contains all words of the form $w_{n,t} = c_1 r_1 \ldots c_n r_n enc(t)$ for some ranked tree $t \in \bigcap_i L(T_i)$ over Δ.

The $\mathsf{VPA}_{\mathsf{la}}$ A works as follows. The n first call symbols are used to run n look-aheads. When the i-th call c_i is read, a look-ahead B_i checks that $enc(t) \in L(A_i)$: it first count that $2(n-i+1)$ symbols have been read and go to the initial state of A_i. If the look-ahead does not accept the suffix, then the computation stops. Therefore there is an accepting run of A on $w_{n,t}$ iff $enc(t) \in \bigcap_i L(A_i)$. Clearly, A is empty iff $\bigcap_i L(A_i) = \varnothing$.

Finally, note that the size of $|A|$ is polynomial in $\sum_i |A_i|$ and that as the A_i are deterministic so are the look-aheads. □

It is now easy to prove that testing functionality is ExpTime-c.

Theorem 75. *Functionality of* $\mathsf{VPT}_{\mathsf{la}}$ *is* ExpTime-c*, even for deterministic look-aheads.*

Proof. For the ExpTime upper-bound, we first apply Theorem 70 to remove the look-aheads. This results in a VPT possibly exponentially bigger. Then functionality can be tested in PTime (Theorem 27).

The lower bound is a direct consequence of the lower bound for the emptiness problem. Indeed, one can construct a $\mathsf{VPT}_{\mathsf{la}}$ that produces two different outputs for each word in its domain, this $\mathsf{VPT}_{\mathsf{la}}$ is therefore functional if and only if it is empty. □

The equivalence and inclusion problems for VPA are ExpTime-c (see [5]). Therefore, one can decide the equivalence and inclusion for $\mathsf{VPA}_{\mathsf{la}}$ by first removing the look-ahead with an exponential blow-up, and then use the ExpTime procedure for VPA. This yields a 2-ExpTime procedure. We show in the next result, that it is possible to decide it in ExpTime. The idea is to construct in PTime two alternating (ranked) tree automata equivalent to the VPA modulo the first-child next-sibling encoding. Look-aheads are encoded as universal transitions. The result follows from the fact that the equivalence and inclusion problems for alternating tree automata are decidable in ExpTime [14].

Theorem 76. *The equivalence and inclusion problems for* $\mathsf{VPA}_{\mathsf{la}}$ *is* ExpTime-c*, even when the look-aheads are deterministic.*

Proof. Let A_1, A_2 be two VPA. We show how to check $L(A_1) = L(A_2)$ in ExpTime. Well-nested words over the alphabet $\Sigma = \Sigma_c \uplus \Sigma_r$ can be translated as unranked trees over the alphabet $\tilde{\Sigma} = \Sigma_c \times \Sigma_r$. Those unranked trees can be again translated as binary trees via the classical first-child next-sibling encoding [14]. VPA over Σ can be translated into equivalent top-down tree automata over first-child next-sibling encodings on $\tilde{\Sigma}$ of well-nested words over Σ in PTime [23]. Look-aheads of VPA inspect the longest well-nested prefix of the current suffix. This corresponds to subtrees in first-child next-sibling encodings of unranked trees. Therefore VPA with look-aheads can be translated into top-down tree automata with look-aheads that inspect the current subtree. Top-down tree automata with such look-aheads can be again translated into alternating tree automata: triggering a new look-ahead corresponds to a universal transition towards two states: the current state of the automaton and the initial state of the look-ahead. This again can be done in PTime. The VPA A_1 and A_2 are equivalent if and only if their associated alternating trees are equivalent, which can be tested in ExpTime [14].

Lower bounds The lower bounds are a direct consequence of the lower bound for the emptiness problem. Indeed, a $\mathsf{VPA}_{\mathsf{la}}$ is empty if and only if it is equivalent to, resp. included into, the empty language. □

As a consequence of the ExpTime bound for testing equivalence or inclusion of $\mathsf{VPA}_{\mathsf{la}}$ and the ExpTime bound for testing functionality, the equivalence of two functional $\mathsf{VPT}_{\mathsf{la}}$ is in ExpTime. Indeed, it amounts to check the equivalence or inclusion of the domains and to, then, check that the union is still functional.

Theorem 77. *The equivalence and inclusion problems for functional* $\mathsf{VPT}_{\mathsf{la}}$ *is* ExpTime-c*, even if the transducers and their look-aheads are deterministic.*

9. Conclusion

In this paper, we have introduced the class of visibly pushdown transducers. We have shown that, contrarily to the more expressive class of pushdown transducers, it enjoys good algorithmic and closure properties. We however leave some problems as open. Let us mention the two that we consider the most important ones:

- **Problem 1** Given two k-valued VPT T_0, T_1, decide whether they are equivalent.
- **Problem 2** Given a functional VPT T, decide whether it is equivalent to a deterministic one.

The first problem is known to be decidable for finite state transducers [50] and undecidable for pushdown transducers. The second problem is challenging for efficient evaluation of streaming transformations. Indeed, a transformation that can be defined by a deterministic VPT can be evaluated with a memory that only stores the stack and the current state, which is, in practice, very efficient on very wide and reasonably deep structured documents, such as XML ones. Again, this problem is decidable (in PTime) for finite state transducers [12, 59, 6] and undecidable for pushdown transducers.

10. Acknowledgments

Emmanuel Filiot is an FNRS research associate and Jean-François Raskin is Professeur Francqui de Recherche 2015-2018 (ULB) funded by the Francqui foundation. This work was partially supported by ANR project DELTA, grant ANR-16-CE40-0007, by the ARC project "Transform" (Fédération Wallonie-Bruxelles), and the FNRS CDR project "Flare" J013116F.

References

[1] Alur, R., 2007. Marrying words and trees. In: 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2007. Vol. 5140 of LNCS. pp. 233–242.
[2] Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L., 2008. First-order and temporal logics for nested words. Logical Methods in Computer Science 4 (4).
[3] Alur, R., Chaudhuri, S., 2010. Temporal reasoning for procedural programs. In: Barthe, G., Hermenegildo, M. V. (Eds.), VMCAI. Vol. 5944 of Lecture Notes in Computer Science. Springer, pp. 45–60.
URL http://dx.doi.org/10.1007/978-3-642-11319-2
[4] Alur, R., Madhusudan, P., 2004. Visibly pushdown languages. In: 36th ACM symposium on Theory of computing, STOC 2004. pp. 202–211.
[5] Alur, R., Madhusudan, P., 2009. Adding nesting structure to words. Journal of the ACM 56 (3), 1–43.
[6] Béal, M.-P., Carton, O., Prieur, C., Sakarovitch, J., 2003. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. Theoretical Computer Science 292 (1), 45–63.
[7] Berstel, J., dec 2009.
[8] Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., Simon, J., 2007. XQuery 1.0: An XML query language, W3C recommendation.
[9] Bray, T., Paoli, J., Maler, E., Yergeau, F., Sperberg-McQueen, C. M., Nov. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, http://www.w3.org/TR/2008/REC-xml-20081126/.
[10] Chistikov, D. V., Majumdar, R., 2013. A uniformization theorem for nested word to word transductions. In: Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings. Vol. 7982 of Lecture Notes in Computer Science. Springer, pp. 97–108.
[11] Chistikov, D. V., Majumdar, R., 2013. A uniformization theorem for nested word to word transductions. In: CIAA. pp. 97–108.
[12] Choffrut, C., 1977. Une caractérisation des fonctions séquentielles et des fonctions sous-sq́uentielles en tant que relations rationnelles. Theoretical Computer Science 5 (3), 325–337.
[13] Clark, J., 1999. XSL Transformations (XSLT) version 1.0, W3C recommendation.
[14] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M., 2007. Tree automata techniques and applications.
[15] Eilenberg, S., 1974. Automata, Languages, and Machines. Academic Press.
[16] Elgot, C. C., Mezei, J. E., 1965. On relations defined by generalized finite automata. IBM Journal of Research and Development 9, 47–68.
[17] Engelfriet, J., 1978. On tree transducers for partial functions. Information Processing Letters 7 (4), 170–172.
[18] Engelfriet, J., Maneth, S., 2003. Macro tree translations of linear size increase are MSO definable. SIAM Journal on Computing 32, 950–1006.
[19] Engelfriet, J., Vogler, H., 1985. Macro tree transducers. Journal of Computer and System Sciences 31 (1), 71–146.
[20] Filiot, E., Gauwin, O., Reynier, P.-A., Servais, F., 2011. Streamability of nested word transductions. In: FSTTCS. pp. 312–324.
[21] Filiot, E., Raskin, J.-F., Reynier, P.-A., Servais, F., Talbot, J.-M., 2010. Properties of visibly pushdown transducers. In: 35th International Symposium on Mathematical Foundations of Computer Science, MFCS 2010. pp. 355–367.
[22] Filiot, E., Servais, F., 2012. Visibly pushdown transducers with look-ahead. In: SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings. Vol. 7147 of Lecture Notes in Computer Science. Springer, pp. 251–263.
[23] Gauwin, O., 2009. Streaming tree automata and xpath. Ph.D. thesis, Université Lille 1.
URL http://tel.archives-ouvertes.fr/tel-00421911/
[24] Gauwin, O., Niehren, J., Tison, S., 2009. Earliest query answering for deterministic nested word automata. In: 17th International Symposium on Fundamentals of Computation Theory, FCT 2009. Vol. 5699 of LNCS. pp. 121–132.
[25] Griffiths, T. V., 1968. The unsolvability of the equivalence problem for lambda-free nondeterministic generalized machines. Journal of the ACM 15 (3), 409–413.
[26] Groups, W. X. Q. W., 2007. The XPath 2.0 standard.
[27] Gurari, E. M., Ibarra, O. H., 1983. A note on finite-valued and finitely ambiguous transducers. Theory of Computing Systems 16 (1), 61–66.
[28] Hague, M., Lin, A. W., 2011. Model checking recursive programs with numeric data types. In: 23rd International Conference on Computer Aided Verification, CAV 2011. pp. 743–759.
[29] Harju, T., Ibarra, O. H., Karhumaki, J., Salomaa, A., 2002. Some decision problems concerning semilinearity and commutation. Journal of Computer and System Sciences 65, 278–294.

[30] Harris, W. R., Jha, S., Reps, T. W., 2012. Secure programming via visibly pushdown safety games. In: Madhusudan, P., Seshia, S. A. (Eds.), Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Vol. 7358 of Lecture Notes in Computer Science. Springer, pp. 581–598.
[31] Heizmann, M., Hoenicke, J., Podelski, A., 2010. Nested interpolants. In: Hermenegildo, M. V., Palsberg, J. (Eds.), POPL. ACM, pp. 471–482.
[32] Hors, A. L., Hgaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S., 2004. Document object model (dom), W3C recommendation.
[33] Ibarra, O. H., 1978. Reversal-bounded multicounter machines and their decision problems. Journal of the ACM 25 (1), 116–133.
[34] Koch, C., Scherzinger, S., 2007. Attribute grammars for scalable query processing on XML streams. International Journal on Very Large Data Bases 16 (3), 317–342.
[35] Kumar, V., Madhusudan, P., Viswanathan, M., 2007. Visibly pushdown automata for streaming XML. In: 16th international conference on World Wide Web, WWW 2007. pp. 1053–1062.
[36] Madhusudan, P., Viswanathan, M., 2009. Query automata for nested words. In: 34th International Symposium on Mathematical Foundations of Computer Science, MFCS 2009. Vol. 5734 of LNCS. Springer Berlin / Heidelberg, pp. 561–573.
[37] Maneth, S., 2003. The macro tree transducer hierarchy collapses for functions of linear size increase. In: 23rd Conference on Foundations of Software Technology and Theoretical Computer Science, FST TCS 2003. pp. 326–337.
[38] Martens, W., Neven, F., 2003. Typechecking top-down uniform unranked tree transducers. In: 9th International Conference on Database Theory, ICDT 2003. pp. 64–78.
[39] Martens, W., Neven, F., 2004. Frontiers of tractability for typechecking simple xml transformations. In: 23th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2004. pp. 23–34.
[40] Martens, W., Neven, F., 2005. On the complexity of typechecking top-down xml transformations. Theoretical Computer Science 336 (1), 153–180.
[41] Martens, W., Neven, F., 2007. Frontiers of tractability for typechecking simple xml transformations. Journal of Computer and System Sciences 73 (3), 362–390.
[42] Megginson, D., 2004. Simple API for XML (SAX 2.0).
URL `http://www.saxproject.org/`
[43] Minsky, M. L., 1967. Finite and Infinite Machines. Prentice-Hall.
[44] Murata, M., 1999. Hedge automata: a formal model for xml schemata.
[45] Neven, F., Schwentick, T., 2002. Query automata over finite trees. Theor. Comput. Sci. 275 (1-2), 633–674.
URL `http://dx.doi.org/10.1016/S0304-3975(01)00301-2`
[46] Perst, T., Seidl, H., 2004. Macro forest transducers. Information Processing Letters 89 (3), 141–149.
[47] Plandowski, W., 1994. Testing equivalence of morphisms on context-free languages. In: Second Annual European Symposium on Algorithms, ESA 1994. pp. 460–470.
[48] Raskin, J.-F., Servais, F., 2008. Visibly pushdown transducers. In: 35th International Colloquium on Automata, Languages and Programming, ICALP 2008. Vol. 5126 of LNCS. pp. 386–397.
[49] Reynier, P.-A., Talbot, J.-M., 2014. Visibly pushdown transducers with well-nested outputs. In: Proc. 18th International Conference on Developments in Language Theory (DLT'14). Vol. 8633 of LNCS. Springer, pp. 129–141.
[50] Sakarovitch, J., de Souza, R., 2010. Lexicographic decomposition of k -valued transducers. Theory of Computing Systems 47 (3), 758–785.
[51] Schützenberger, M. P., 1976. Sur les relations rationnelles entre monoides libres. Theoretical Computer Science 3 (2), 243–259.
[52] Segoufin, L., Vianu, V., 2002. Validating streaming XML documents. In: 21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002. pp. 53–64.
[53] Servais, F., 2011. Visibly pushdown transducers. Ph.D. thesis, Université Libre de Bruxelles.
URL `http://theses.ulb.ac.be/ETD-db/collection/available/ULBetd-09292011-142239/`
[54] Staworko, S., Laurence, G., Lemay, A., Niehren, J., 2009. Equivalence of deterministic nested word to word transducers. In: 17th International Symposium on Fundamentals of Computation Theory, FCT 2009. Vol. 5699 of LNCS. pp. 310–322.
[55] Thomo, A., Venkatesh, S., Ye, Y. Y., 2008. Visibly pushdown transducers for approximate validation of streaming XML. In: 5th international conference on Foundations of information and knowledge systems, FoIKS 2008. pp. 219–238.
[56] Verma, K. N., Seidl, H., Schwentick, T., 2005. On the complexity of equational horn clauses. In: 20th International Conference on Automated Deduction, CADE 2005. Vol. 3632 of LNCS. pp. 337–352.
[57] Walmsley, P., Fallside, D. C., Oct. 2004. XML Schema Part 0: Primer Second Edition. W3C recommendation, W3C, `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`.
[58] Weber, A., 1993. Decomposing finite-valued transducers and deciding their equivalence. SIAM Journal on Computing 22 (1), 175–202.
[59] Weber, A., Klemm, R., 1995. Economy of description for single-valued transducers. Information and Computation 118 (2), 327–340.