Faculté des Sciences Appliquées
Département CoDE

Université Libre de Bruxelles

Visibly Pushdown Transducers

Frédéric SERVAIS

Thèse présentée en vue de l'obtention
du grade de Docteur en Sciences Appliquées

Sous la direction du Prof. Esteban Zimányi et co-direction du Prof. Jean-François Raskin

Visibly Pushdown Transducers

Frédéric SERVAIS

Thèse réalisée sous la direction du Professeur Esteban Zimányi et co-direction du Professeur Jean-François Raskin et présentée en vue de l'obtention du grade de Docteur en Sciences Appliquées. La défense publique a eu lieu le 26 septembre 2011 à l'Université Libre de Bruxelles, devant le jury composé de:

À mes parents

Résumé

Dans ce travail nous introduisons un nouveau modèle de transducteurs, les transducteurs avec pile à comportement visible (VPT), particulièrement adapté au traitement des documents comportant une structure imbriquée.

Les VPT sont obtenus par une restriction syntaxique de transducteurs à pile et sont une généralisation des transducteurs finis. Contrairement aux transducteurs à piles, les VPT jouissent de relativement bonnes propriétés.

Nous montrons qu'il est décidable si un VPT est fonctionnel et plus généralement s'il définit une relation k-valuée. Cependant cette classe de transducteurs n'est pas close par composition, contrairement à la classe des transducteurs finis. De plus le problème du typage des VPT par rapport aux langages "visibly pushdown" est indécidable.

Nous introduisons une sous classe des VPT, formée par les VPT bien imbriqués. Cette classe est close par composition et le typage par apport aux langages visibly pushdown est décidable.

Nous montrons ensuite que la classe des VPT est également close par 'look-ahead'. De plus les transformations fonctionnelles spécifiées par des VPT peuvent toujours être définies par des VPT déterministes avec look-ahead. Ces propriétés témoignent de la robustesse de cette classe.

Finalement nous montrons qu'il est possible de s'assurer si une transformation définie par un VPT fonctionnel peut être exécutée avec une quantité de mémoire raisonnable quelque soit le document à traiter. Par raisonnable nous entendons une quantitéde mémoire qui ne dépend pas de la taille du document, mais peut cependant dépendre du niveau d'imbrication de celui-ci. Cette quantité est en effet la quantité minimale nécessaire pour traiter ce genre de document, car par exemple pour vérifier que la structure d'un document imbriqué est correcte, une quantité de mémoire proportionnelle au niveau d'imbrication est nécessaire.

Abstract

The present work proposes *visibly pushdown transducers* (VPTs) for defining transformations of documents with a *nesting structure*. We show that this subclass of pushdown transducers enjoy good properties. Notably, we show that functionality is decidable in PTime and k-valuedness in co-NPTime. While this class is not closed under composition and its type checking problem against visibly pushdown automata is undecidable, we identify a subclass, the *well-nested* VPTs, closed under composition and with a decidable type checking problem. Furthermore, we show that the class of VPTs is closed under *look-ahead*, and that the deterministic VPTs with look-ahead characterize the functional VPTs transductions. Finally, we investigate the resources necessary to perform transformations defined by VPTs. We devise a *memory efficient* algorithm. Then we show that it is decidable whether a VPT transduction can be performed with a memory that depends on the *level of nesting* of the input document but not on its length.

Acknowledgements

My first and foremost thanks goes to my advisors Esteban Zimányi and Jean-François Raskin for their support and invaluable help in achieving this work. These years have given me the opportunity to highly value your professional and personal qualities.

A special thanks goes to the members of the jury for the time and effort they invested in carefully reviewing this work as well as for their useful feedback.

This work would not have been the same without Emmanuel Filiot. We worked together for the last couple of years. Manu let me say my most sincere 'thank you'.

This work was also made possible by the so-called french connection: Pierre-Alain and Jean-Marc from Marseille, and Olivier from Lille. Working with you all was a great pleasure. My warm thanks goes to each of you.

I owe a big thank to Boris, Serge, Ivan, Frédéric, Olivier, François, Pierre, Rafaella, Cédric, Nicolas, Gabriel, Thierry, Jean, Laurent and Laurent for their support, discussions, and all the time spent together throughout these years at the university. Thank you all!

I would also like to thank colleagues from the laboratory WIT: Mohammed, Jean-Michel, Vinciane, Aura, ... As well as colleagues from the department of informatics: Jean, Raymond, Pascaline, Véronique, Maryka, Gilles, Mahsa...

My infinite gratitude goes to my friends for being there throughout those, sometimes difficult, moments: Isa, Schou, François, Magda, Marielle, Manu, Kostas, Liv, Ivan, Marloes, Ewa, Oliver and Olivier, Sylviane, Alain, Georges, Marc, Thierry, Samir, and all those that I, unforgivably, fail to mention.

Finally I thank my parents and my family for their care and support. And last but certainly not least: Eleonora.

Contents

Chapter 1

Introduction

The present work proposes an abstract model of machines, called visibly pushdown transducers, for defining transformations of documents with a nesting structure.

Roughly speaking, a document is a sequence of symbols. The term *nesting structure* for a document refers to a structure that is organized on the basis of layers some of which are contained into the others. Special symbols, like opening and closing parenthesis, can be used to add nesting structure to a document. More generally, adding a set of opening symbols and a set of closing symbols to the set of data symbols of the document, induces a nesting structure. An opening symbol indicates the beginning of a new nesting layer (we say that we are entering an additional level of nesting), while a closing symbol indicates the end of the last opened layer. XML documents are examples of such documents: their opening and closing tags induce the nesting structure.

A transducer is a machine that defines a transformation of input documents into output documents. This relation between input and output is called a transduction. The usefulness of a class of transducers varies according to its properties.

First and foremost, performing the transformation of a document should be feasible with a reasonable amount of resources such as time and memory.

A second desirable property is the decidability of some problems related to the transducers. For example, given two transducers, a procedure for deciding whether they define the same transduction can be applied to check that a refactoring (*e.g.* minimization) is correct. The type checking problem is another example of a decision problem with useful applications. It asks, given a transducer, and an input and an output types, 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 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.

Finally, closure under some operations is desirable. Indeed, it might be easier to define a transduction as the result of some operation, such as composition or union, applied on two simpler transductions. If one can construct a transducer (of the same class) that defines the resulting transduction, this transducer inherits all the benefits of the class, such as having an efficient evaluation algorithm and the decidability of some decision problems, moreover the construction can be iterated.

The various kind of transducers studied in this thesis all rely on machines, called automata, that read documents. An automaton processes a document step by step, symbol by symbol. It starts in a predefined internal configuration, at each step it reads a symbol of the document and updates its internal configuration accordingly. After reading the whole document, the automaton internal configuration determines whether this sequence of steps, called an execution, accepts the document or not. Note that automata can be *non-deterministic*, at each step several transitions might apply. The set of documents for which there exists a sequence of steps accepting the document, is called the language of the automaton. A transducer is just an automaton that, at each step, produces a fragment of the output document. The output of the transduction is the concatenation of all the output fragments. An image of an input document is the output of an accepting execution. As the automata can be non-deterministic, an input document can have several images.

Clearly, the properties of a class of transducers depend on the properties of the class of its underlying automata. If the underlying class is not closed under some operations, like intersection or union, the corresponding class of transducers is not either. Furthermore, the undecidability of some decision problems for the automata, directly translates into the undecidability of some problems for the transducers. For example, if the equivalence or inclusion problem is undecidable for the automata, it is trivially undecidable for the transducers (as the identity transduction is definable for all classes of transducers of interest).

The class of finite state automata is one of the simplest class of automata. It enjoys excellent closure and decidability properties. It is closed for all Boolean operations and the emptiness, inclusion, equivalence, universality problems are all decidable. The derived class of transducers, the finite state transducers, enjoys reasonable properties. It is closed under union, inverse and composition, and its emptiness problem is decidable. Moreover, the type checking problem is decidable when the type of the input and output documents are defined by finite state automata. However, this class is not closed under intersection, nor under complement, and its inclusion and equivalence problems are undecidable.

Interestingly, the class of finite state transducers that defines *functional* transductions, *i.e.* each input document has at most one image, has a decidable equivalence and inclusion problem. More generally, the class of k-valued finite state transducers, *i.e.* the transducers for which each document has at most k images, has also decidable equivalence and inclusion problems. Furthermore, it is decidable whether a finite state transducer is k-valued, and a fortiori whether it is functional.

Overall the class of finite state transducers has reasonably good properties, particularly the class of k-valued transducers. However this class is not suitable for processing documents with a nesting structure. For a fact, 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. When entering a nested layer, that is when reading an opening symbol, the automaton pushes the information on the stack, while it pops it when leaving the nested layer. As documents can have arbitrarily large nesting levels, one cannot bound the size of the stacks. As a consequence, this can generally not be implemented with finite memory automata, it requires a more powerful machine.

A pushdown automaton is a finite state automaton equipped with a stack data structure. When reading an input symbol, it reads and pops the symbol on top of the stack, it pushes a sequence of symbols on the stack and it changes its internal state, which it chooses among a finite set of possible states. Pushdown automata define the class of context-free languages. It is not closed under intersection nor under complement, and the equivalence, inclusion and universality problems are all undecidable. Not surprisingly the class of pushdown transducers inherits all these weaknesses.

The class of pushdown transducers is closed under inverse and union, but it is not closed under intersection, complement and composition. Its emptiness problem is decidable, but its equivalence and inclusion problems are undecidable (note however that it is already the case for finite state transducers). When restricting to functional transductions, the situation is not better. The equivalence and inclusion are still undecidable. Therefore, while the stack is required to deal with documents with a nesting structure, the class of pushdown transducers has a limited field of application.

Visibly pushdown automata (VPA) form a subclass of pushdown automata specially tailored to deal with such nested structures. VPA operate on an structured alphabet, *i.e.* a set of symbols partitioned into call, return and internal symbols, which correspond to the opening, closing and data symbols respectively. When reading a *call* symbol the automaton must push one symbol on its stack, when reading a *return* it must pop the symbol on top of its stack, and when reading an *internal* symbol it cannot touch its stack. This restriction on the behavior of the stack is fully consistent with the use one would want to have in the setting of documents with nesting structure. Contrary to the class of pushdown automata, the class of VPA enjoys excellent closure and decision properties. This is in part due to the fact 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.* Just as for finite state automata, this class is closed under all Boolean operators, and all the important decision problems are decidable.

In this thesis, we introduce and investigate the class of visibly pushdown transducers (VPT). They are visibly pushdown automata equipped with an output mechanism. While the input alphabet is structured and drives the stack, the output alphabet is not necessarily structured. We investigate whether the excellent properties of visibly pushdown automata induce good properties for VPTs.

We first show that this class enjoys a few bad and a few good properties. It is not closed under any operation but union, and the equivalence and inclusion problems are, as a consequence of the corresponding result for finite state transducers, undecidable. Moreover, the type checking against visibly pushdown automata is also undecidable. Nevertheless, we show that it is decidable whether a transduction, defined by a visibly pushdown transducer, is functional and more generally whether it is k-valued. Furthermore, for functional transductions, we show that the equivalence and inclusion problems are decidable.

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. 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. They are visibly pushdown transducers that produces output on a structured alphabet and whose output is somehow synchronized with the stack and 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.

We then study the extension of visibly pushdown transducers with look-ahead. 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 captures 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.

Additionally, we present a memory-efficient algorithm that performs the transformation defined by a VPT. It operates in one pass over the input and uses a compact data structure in order to minimize the memory footprint. When the transduction is defined by a *deterministic* VPT the memory needed is proportional to the level of nesting of the input word and does not depend on its length. However, some functional non-deterministic transductions (such as the one that swaps the first and the last letter) might require an amount of memory that depends on the length of the input word, which can be large.

We exhibit a necessary and sufficient condition for a *non-deterministic* VPT transductions to be evaluated in one pass with a memory that is dependent on the height of the document but not on its length. We say that these transductions are height-bounded memory (HBM). We show that in the worst case the HBM transductions need a memory that is exponential in the height of the input.

Finally, we identify the class of *twinned* VPT transductions that contains all *determinizable* VPT transductions. We show that these VPT transductions can be performed, without pre-computation step, such as determinization, with the same space complexity as the *deterministic* VPT transductions, *i.e.* in space that is polynomial in the height of the input word.

Many problems remain open. Notably, the problem of deciding, given a VPT, whether there exists an equivalent deterministic one is left open. Also, while we show that the equivalence and inclusion problems are decidable for functional

VPT, we do not have a similar result for k-valued transductions. Both of these problems are decidable for finite state transducers and undecidable for pushdown transducers.

Motivations

Visibly pushdown automata have been firstly motivated by application to verification as a model of the control flow of programming languages with recursive procedure calls [AM04]. The second motivating application area is XML processing. The opening and closing tags of XML documents induce their nesting-structure. Because of their underlying processing model, VPA naturally fit the design of streaming algorithm for XML processing [Kum07], *i.e.* algorithms that proceed in one pass over the input and use a "low" amount of memory. For instance, VPA are expressive enough to capture the whole structural part of XML schema languages [PSZ11]. As the memory used is proportional to the height of the stack, and as these automata only push on opening tags and always pop on closing tags, the memory required is proportional to the level of nesting of the input XML document (generally low [BMV05]), but is independent of its length. In other words, contrary to DOM [ALH04] based processors that construct a memory-demanding, tree-like representation of the document, pushdown machines process the document as a string (or, equivalently, a stream). In [PSZ11], we have implemented a complete VPA-based XML validator. It receives as input an XML Schema [WF04] and produces an equivalent VPA-based validator. Its performances are excellent, the memory and time used are low, showing that VPA are very well-suited for efficient XML validation. VPA have also been successfully employed for querying XML documents [Cla08, MV09, GNT09].

While visibly pushdown transducers might be applied to verification, we are mainly interested in the applications to XML processing. In the setting of XML, VPT are models of XML transformations. Inspired by the effectiveness of the application of VPA to XML processing, we are interested in the possible applications of VPT for efficient *implementations* of XML transformations, as well as *static analysis* of transformations, such as type checking the transformation against input and output specifications or the static analysis of the memory required for performing a transformation.

Outline of the Document

In Chapter 2, we fix the notations and recall some elementary notions such as language and transduction.

In Chapter 3, we recall the definitions of finite state transducers, this class provides a baseline for all classes of transducers. We therefore present proofs of the results, such as the undecidability of the equivalence and inclusion problems, that are used for deriving results on VPTs. Moreover, we recall some essential constructions, such as the squaring of a transducer, that we generalize to VPT later on.

Also in Chapter 3, we present the class of pushdown transducers. The positive results, such as the decidability of the emptiness or the membership problems, directly carry over to VPTs. We present the simple proofs of undecidability, this gives some insight on the weaknesses of this class.

In Chapter 4, we present the visibly pushdown automata, they form the class of underlying automata for visibly pushdown transducers. We recall the important notion of summary used in the determinization procedure. This notion is extensively used later when investigating the extension of VPT with look-ahead. We also recall the construction to reduce a VPA, that is, to build an equivalent VPA, such that its reachable configurations are all co-accessible. We recall the complexity of the decision procedures and compare these results to the results for finite state and pushdown automata.

Visibly pushdown transducers are introduced in Chapter 5. After proving some easy results, such as the non-closure under composition and the undecidability of the type checking, we dedicate the most part of this chapter to show that functionality and k-valuedness are decidable. The decidability of the functionality is proved with a reduction to the morphism equivalence problem for context-free languages, a problem that was proved to be decidable in PTime [Pla94]. To decide k-valuedness, we use bounded-reversal counter automata [Iba78], to find witnesses with more than k images. The k-valuedness problem is therefore reduced to the emptiness problem for those counter machines. We prove that the procedure of [HIKS02] to decide this emptiness problem can be executed in co-NPTime and as a consequence, the k-valuedness problem for VPT is decidable in co-NPTime.

In Chapter 6 we introduce the class of well-nested visibly pushdown transducers. We show that it is closed under composition and that its type checking problem against VPA is decidable. Finally, as words with a nesting structure can be viewed as trees, we compare the class of wnVPT with several classes of tree transducers: top-down binary tree transducers, macro tree transducers and uniform tree transducers.

In Chapter 7, we study the resources required to perform transformation defined by VPTs. We first introduce an algorithm, called Eval, that efficiently performs VPT transductions. Then we prove that it is decidable whether a VPT transduction can be performed with height-bounded memory. We finally strengthen this last result and exhibit a class of VPT transduction that can be evaluated in quadratic height-bounded memory.

In the last chapter, Chapter 8, we introduce the visibly pushdown transducers with look-ahead. We show that the transductions definable by such machine are the same as with VPTs and that the deterministic VPT with look-ahead define exactly the functional VPT transductions. Finally, we investigate the complexity of the decision problems.

In the final conclusion, we present some important open problems and discuss the possible future work and applications.

Chronology

Chapters 5, 6, 7, 8 are based on four articles presented in international conferences and a technical report.

Chapter 5 and 6 are based on:

- J.-F. Raskin and F. Servais. *Visibly pushdown transducers.* In ICALP, volume 5126 of LNCS, pages 386–397. Springer, 2008.
- E. Filiot, J.-F. Raskin, P.-A. Reynier, F. Servais, and J.-M. Talbot. *Properties of visibly pushdown transducers.* In MFCS, volume 6281 of LNCS, pages 355–367. Springer, 2010.

Chapter 7 is based on:

- E. Filiot, J.-F. Raskin, P.-A. Reynier, F. Servais, and J.-M. Talbot. *On functionality of visibly pushdown transducers.* CoRR, abs/1002.1443, 2010.
- E. Filiot, O. Gauwin, P.-A. Reynier, and F. Servais. *Streamability of nested word transductions.* FSTTCS 2011.

And Chapter 8 is based on:

- E. Filiot, and F. Servais. *Visibly pushdown transducer with look-ahead.* SOFSEM, 2012.

An additional related work about an implementation of an XML validator based on visibly pushdown automata was presented in an international conference:

- F. Picalausa, F. Servais, E. Zimányi: *XEvolve: an XML schema evolution framework.* SAC 2011: 1645-1650

Chapter 2

Preliminaries

Alphabets and words. Let S and T be two sets. The intersection, resp. union, of S and T is denoted by $S \cap T$, resp. $S \cup T$. We write $S \uplus T$, instead of $S \cup T$, when S and T are disjoint, i.e. $S \cap T = \emptyset$. The cartesian product of S and T is $S \times T = \{(a, b) \mid a \in S \wedge b \in T\}$, and $S^n = S \times S \cdots \times S = \{(a_1, a_2, \ldots a_n) \mid \forall i \leq n : a_i \in S\}$ is the n-th power of S.

An *alphabet* Σ is a non-empty finite set, its elements are called *symbols* or *letters*. A word u over Σ is a finite sequence of letters of Σ, *i.e.* $u = a_1 \ldots a_n$, 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.

Let n be a natural number, $\Sigma^n = \{a_1 a_2 \ldots a_n \mid \forall i \leq n : a_i \in \Sigma\}$ is the set of words of length n over Σ. We pose $\Sigma^0 = \{\epsilon\}$, the set containing the empty word. The set of all words over Σ is denoted by $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$, and $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$ is the set of non-empty words.

Let $u, v \in \Sigma^*$ be two words over Σ. The *mirror image* of $u = a_1 \ldots a_n \in \Sigma^*$, denoted by u^r, is the word $a_n \ldots a_1$. 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.* $w \preceq u$, $w \preceq v$, and for all $w' \in \Sigma'$ with $w' \preceq u$ and $w' \preceq v$, then $w' \preceq w$. In particular, the word $(u \wedge v)^{-1}v$ is the word obtained from v after removing its longest common prefix with u. The *delay* between u and v is the pair of words $\Delta(u, v) = ((u \wedge v)^{-1}u, (u \wedge v)^{-1}v)$.

Morphism. Let Σ and Δ be two alphabets. A *morphism* h from Σ into Δ^* is a mapping that associates with every letter of Σ an element of Δ^*. The image of a word $u = a_1 \ldots a_n \in \Sigma^*$ is $h(u) = h(a_1 \ldots a_n) = h(a_1) \ldots h(a_n)$. For all

$u, v \in \Sigma^*$ we have $h(uv) = h(u)h(v)$. In particular, for any morphism h we have that $h(\epsilon) = \epsilon$. The image of a set of words L by a morphism h is the set $h(L) = \{h(u) \mid u \in L\}$.

Languages

Let Σ be an alphabet. A set of words over Σ, $L \subseteq \Sigma^*$, is called a *language* over Σ. The *concatenation* of 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, obtained by concatenating exactly n words of L, is defined inductively as $L_0 = \{\epsilon\}$ and $L^{i+1} = L^i \cdot L$ for all $i \geq 0$. The *star operation*, or *Kleene star*, on L is defined as $L^* = \bigcup_{n \geq 0} L^n$, and L^+ denotes the set $\bigcup_{n \geq 1} L^n$. [1] The *mirror*, resp. the *complement*, of L is defined as $L^r = \{u^r \mid u \in L\}$, resp. $\overline{L} = \Sigma^* \setminus L$. A *class* of languages $\mathcal{L}$ is a set of languages, *i.e.* $\mathcal{L} \subseteq 2^{\Sigma^*}$.

Closure Properties

We say that a class of languages $\mathcal{L}$ is *closed under* union, intersection, resp. concatenation, if for all $L_1, L_2 \in \mathcal{L}$ we have $L_1 \cup L_2 \in \mathcal{L}$, $L_1 \cap L_2 \in \mathcal{L}$, resp. $L_1 \cdot L_2 \in \mathcal{L}$. Similarly, we say that $\mathcal{L}$ is closed under complement, mirror image, resp. star operation if for all $L \in \mathcal{L}$ we have $\overline{L} \in \mathcal{L}$, $L^r \in \mathcal{L}$, resp. $L^* \in \mathcal{L}$. Finally, $\mathcal{L}$ is closed under morphism if for all $L \in \mathcal{L}$ and any morphism h, we have $h(L) \in \mathcal{L}$.

Decision Problems

We briefly recall the decision problems for languages. Let Σ be an alphabet and $\mathcal{L}, \mathcal{L}' \subseteq 2^{\Sigma^*}$ be two classes of languages.

Membership. The *membership problem* for $\mathcal{L}$ asks given a language $L \in \mathcal{L}$ and a word $u \in \Sigma^*$, whether $u \in L$ holds.

[1] Note that while the notation is the same as for denoting the n-power of a set, it does not, however, always yields an equivalent object. Indeed, the pair (a, aa) and (aa, a) are different, but concatenating a with aa or aa with a yields the same word. The context always makes it clear which operation is involved.

Emptiness, universality. The *emptiness problem* for $\mathcal{L}$ asks given a language L in $\mathcal{L}$ whether it is empty, *i.e.* whether $L = \emptyset$ holds. The *universality problem* asks whether L contains all possible words, *i.e.* whether $L = \Sigma^*$ holds.

Inclusion, equivalence, disjointness. The *inclusion problem* for $\mathcal{L}$ asks given two languages L, L' in $\mathcal{L}$ whether the first is included in the second, *i.e.* whether $L \subseteq L'$ holds. The *equivalence problem* asks if they are the same, *i.e.* whether $L = L'$ holds. The *disjointness problem* asks if their intersection is empty, *i.e.* whether $L \cap L' = \emptyset$ holds.

$\mathcal{L}'$-membership. The $\mathcal{L}'$-membership problem asks to determine whether a language of a class $\mathcal{L}$ belongs to another class $\mathcal{L}'$, *e.g.* whether a given context-free language is regular. Formally, the *$\mathcal{L}'$-membership problem* for $\mathcal{L}$ asks, given $L \in \mathcal{L}$, whether $L \in \mathcal{L}'$ holds.

Transductions

Let Σ and Δ be two alphabets. A *transduction* from Σ^* to Δ^* is a relation between words of Σ^* and words of Δ^*, i.e. it is a subset of $\Sigma^* \times \Delta^*$.

Let $R \subseteq \Sigma^* \times \Delta^*$ be a transduction. We say that v is a *transduction*, or an *image*, of u by R whenever $(u, v) \in R$. The *image* of u by R is the set of all transductions of u by R and 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*(R), is the set $\{v \in \Delta^* \mid \exists u \in \Sigma^* : (u, v) \in R\}$.

In the sequel, we denote by $\mathcal{T}$, $\mathcal{T}'$ classes of transductions, i.e. $\mathcal{T}, \mathcal{T}' \subseteq 2^{\Sigma^* \times \Delta^*}$.

Domain, Range and Expressiveness

The first natural question when investigating a class of transductions is to ask into which class of languages do the domains of a given class of transductions fall. Formally, for all $R \in \mathcal{T}$, does $dom(R) \in \mathcal{L}$ hold? A class of languages with good closure properties and decidable decision problems is necessary to answer natural questions such as the following. Given an input language L_{in}, does $L_{in} \subseteq dom(R)$ hold? *i.e.* are all words of the input language translated by the transduction?

The same question arise for the range of the transductions. For all $R \in \mathcal{T}$, does $\mathit{range}(R) \in \mathcal{L}'$ hold? Again a good class is required to answer natural questions such as: given an output language $L_{out} \in \mathcal{L}'$, whether $\mathit{range}(R) \subseteq L_{out}$ holds, *i.e.* whether all words produced by the transduction belong to the output language.

Finally, given two classes of transductions, $\mathcal{T}$ and $\mathcal{T}'$, we might want to know whether one is more expressive than the other. That is, whether all transductions of the first class belong to the second class, *i.e.* for all $R \in \mathcal{T}$, whether $R \in \mathcal{T}'$ holds.

Closure Properties

Transductions are subsets of $\Sigma^* \times \Delta^*$. As such a class of transductions can be *closed under Boolean operations*: union, intersection and complement. Given $R_1, R_2 \in \mathcal{T}$, the intersection, $R_1 \cap R_2$, is the relation $\{(u,v) \mid (u,v) \in R_1 \wedge (u,v) \in R_2\}$, the union, $R_1 \cup R_2$, is the relation $\{(u,v) \mid (u,v) \in R_1 \vee (u,v) \in R_2\}$, and the complement, $\overline{R}$, is the relation $\{(u,v) \in \Sigma^* \times \Delta^* \mid (u,v) \notin R\}$. A class $\mathcal{T}$ is closed under Boolean operations if for all $R, R_1, R_2 \in \mathcal{T}$, then $R_1 \cap R_2 \in \mathcal{T}$, $R_1 \cup R_2 \in \mathcal{T}$, and $\overline{R} \in \mathcal{T}$.

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 class $\mathcal{T}$ is closed under inverse if for all $R \in \mathcal{T}$, we have $R^{-1} \in \mathcal{T}$.

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 $R_2 \circ R_1 = \{(u,v) \mid \exists w \in {\Sigma_2}^* \wedge (u,w) \in R_1 \wedge (w,v) \in R_2\}$. The class $\mathcal{T}$ is closed under composition if for all $R_1, R_2 \in \mathcal{T}$, we have $R_2 \circ R_1 \in \mathcal{T}$.

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.* let $R \in \mathcal{T}$ and $L \in \mathcal{L}$, $R_{|L} = \{(u,v) \mid (u,v) \in R \wedge u \in L\}$. A class $\mathcal{T}$ is closed under restriction over the class of language $\mathcal{L}$ if for all $R \in \mathcal{T}$ and $L \in \mathcal{L}$, we have $R_{|L} \in \mathcal{T}$.

Decision Problems

Translation membership. The *translation membership problem* asks given a transduction $R \subseteq \Sigma^* \times \Delta^*$ and a pair of input and output word, $u \in \Sigma^*$ and $v \in \Delta^*$, whether the second word is a transduction of the first by R, *i.e.* whether $(u,v) \in R$ holds.

Type checking. The *type checking problem* for $\mathcal{T}$ against $\mathcal{L}$ asks, given an input language $L_{in} \in \mathcal{L}$, an output language $L_{out} \in \mathcal{L}$ and a transduction $R \in \mathcal{T}$, whether the images of any word of the input language belong to the output language, *i.e.* whether $R(L_{in}) \subseteq L_{out}$ holds.

Emptiness, universality. The *emptiness problem* asks whether $R = \emptyset$ holds, and the *universality problem* asks whether R is equal to $\Sigma^* \times \Delta^*$. Note that the emptiness problem for a transduction is equivalent to the problem of the emptiness of its domain. Note also that the universality of the domain does not imply universality of the transductions, such transductions with universal domain are said *total*.

Equivalence, inclusion. Two utterly important problems are the *equivalence and the inclusion problems*, they ask whether two transductions are equal, resp. included. Formally if $R_1, R_2 \in \mathcal{T}$, the equivalence problem asks whether $R_1 = R_2$ holds, while the inclusion problem asks whether $R_1 \subseteq R_2$ holds. In general those problems are undecidable, in that case, an interesting question is to identify subclasses with decidable inclusion and equivalence problems.

Functionality. A relation is *functional*, or simply a function, if all input words have at most one image. The *functionality problem* for $\mathcal{T}$ asks given $R \in \mathcal{T}$ whether for all $u \in \Sigma^*$, $|R(u)| \leq 1$ holds.

k-valuedness. Let $k \in \mathbb{N}$, the notion of k-valuedness is a slight generalization of the notion of functionality. We say that R is *k-valued* if u has at most k images by R for all $u \in \Sigma^*$. The *k-valuedness problem* for $\mathcal{T}$ asks, given $R \in \mathcal{T}$ whether for all $u \in \Sigma^*$, $|R(u)| \leq k$ holds. Note that 1-valuedness is a synonym for functionality.

$\mathcal{T}'$-membership. The *$\mathcal{T}'$-membership problem* for $\mathcal{T}$ asks, given $R \in \mathcal{T}$, whether $R \in \mathcal{T}'$ holds. For example, classes defined by non-deterministic transducers (*i.e.* machines that define transductions) may have a deterministic counterpart, these classes are usually less expressive than their non-deterministic counterpart, in that case we may asks whether a transduction is *determinizable*, that is whether there exists an equivalent deterministic transducer.

Miscellaneous

The post correspondence problem is one of the fundamental undecidable problems. It is a useful tool for deriving undecidability results.

Definition 2.0.1 (Post Correspondence Problem)**.** *Let Σ, Δ be two alphabets. An* instance *of the Post correspondence problem (PCP) is a pair of morphisms $P = (h_1, h_2)$ from Σ into Δ^*. The* Post correspondence problem *asks, given an instance P, whether there exists a non-empty word $u \in \Sigma^+$ such that $h_1(u) = h_2(u)$.*

Whenever $\Sigma = \{1, 2, \ldots, n\}$, and $h_1(i) = u_i$ and $h_2(i) = v_i$ for $i \in \Sigma$, then we may simply write the instance P as the sequence of pairs of words $(u_1, v_1) \ldots (u_n, v_n)$.

Theorem 2.0.2 ([Pos46])**.** *PCP is undecidable.*

Chapter 3

Finite State and Pushdown Transducers

Contents

Word transducers are abstract machines for specifying relations between words over an input alphabet and words over an output alphabet, *i.e.* for defining transductions. In this chapter, we present two important models of word transducers: the finite state transducers and the pushdown transducers.

These transducers are essentially automata equipped with an output mechanism. For both of these classes of transducers, we first present the underlying model of word automata and recall its properties. Then, we present the results for the transducers.

A word automaton is an abstract machine that has an internal configuration and a set of predefined transition rules. It processes words, called *input words*, generally, letter by letter, left to right, starting in a predefined initial internal configuration. When processing a letter, it chooses a transition rule compatible with its current internal configuration and with the input letter. The internal configuration is updated according to the chosen rule, this step is called a *transition*. The sequence of chosen transition rules is called a *run* of the automaton. A run is successful if the automaton ends after processing the entire input word in one of the predefined *final configurations*. A word is *accepted* by the automaton when there exists a successful run on this word. The language defined by the automaton is the set of accepted words.

A simple way to turn any word automaton into a transducer is to associate an output word with each rule. The automaton is called the *underlying* automaton of the transducer. A run of the transducers is a run of it underlying automaton, thus it is a sequence of rules. The output of a run is simply the concatenation of the output word associated with each rule of this run. The images of a word by a transducer is the set of outputs of the successful runs of the underlying automaton on this word. This defines a relation, called *transduction*, between the set of input words and the set of output words.

Finite states transducers (NFT) form one of the simplest class of machines for defining transductions. However, while simple, these machines are nonetheless powerful, and, contrarily to finite state automata, some important decision problems are undecidable. The results on NFT serve as a baseline for other, more expressive, classes of transductions. Indeed, all undecidability results for NFT do carry over to the other classes of transducers presented in this document. For closure properties, we cannot directly transfer closure or non closure results to more expressive classes. However, we will see that non-closure results do also hold for the other classes presented here.

Pushdown automata are finite state automata equipped with a stack data structure. They define the class of context-free languages which are a strict extension of regular languages. However, context-free languages do not enjoy as good closure properties as the regular languages do. Moreover all the main decision problems are undecidable with the exception of the emptiness and membership problems. It is not surprising that these weaknesses carry over to the class of pushdown transducers. For that class of machines, nearly all important decision problems are undecidable, with the notable exception of the emptiness and the translation membership problems. It is therefore worth looking for subclasses with better decidability results. The subclass formed by the deterministic pushdown transducers does not have much better properties.

We end this chapter with a comparison between the different classes introduced. In particular we summarize all closure and non-closure properties and the complexity for each decision problems.

3.1 Finite State Machines

In this section, we recall the notion of finite state automata and finite state transducers. Finite state automata are the underlying machines that "operate" transducers, in other words, transducers are automata with an output mechanism. We highlight the three main families of finite state automata: deterministic, non-deterministic and non-deterministic with ϵ-transitions. At a language level those three models have the same expressiveness, each of them characterizes the class of regular languages. However, when automata are considered as the underlying machine of transducers, these three classes of automata yield three different classes of transductions.

The complete theory about finite state automata and transducers can for example be found in [Sak09, Ber09].

3.1.1 Finite State Automata

To fix notations and terminology we recall the definition and semantics of finite state automata. We follow the standard terminology used in [HMU03]. All results and proofs can be found in any introductory book on formal languages theory.

Definition 3.1.1 (ϵ-NFA)**.** *A non-deterministic finite state automaton with ϵ-transitions (ϵ-NFA) on the alphabet Σ is a tuple $A = (Q, I, F, \delta)$ where*

- *Q is the set of states,*
- *$I \subseteq Q$ is the set of initial states,*
- *$F \subseteq Q$ is the set of final states,*
- *$\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation.*

A run ρ of A on $u \in \Sigma^*$ from state q_0 to state q_n is a sequence of transitions of the following form: $\rho = (q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2) \dots (q_{n-1}, \alpha_n, q_n)$ such that for all $i \in \{1, \dots n\}$ we have $\alpha_i \in \Sigma \cup \{\epsilon\}$ and $(q_{i-1}, \alpha_i, q_i) \in \delta$, and $u = \alpha_1 \cdot \alpha_2 \dots \alpha_n$. For any $q \in Q$, we let $\rho = \epsilon$ be a run over the empty word $\epsilon \in \Sigma^*$ from state q to

q. We write $A \models q_0 \xrightarrow{u} q_n$ when there is a run of A on u from states q_0 to state q_n. Also we may write $\rho = q_0 q_1 \dots q_n$ when the input letters are clear from the context. The set of runs of A over u is denoted by $\rho(A, u)$.

A run is accepting if $q_0 \in I$ and $q_n \in F$. A word u is accepted by A if there exists an accepting run of A on u. We denote by $\rho^{acc}(A, u)$ the set of accepting runs of A on u. The language $L(A)$ of A is the set of accepted words: $L(A) = \{u \in \Sigma^* \mid \rho^{acc}(A, u) \neq \emptyset\}$. The languages accepted by ϵ-NFA are called regular languages, the set of regular languages is denoted by REG. We denoted by $|A| = |Q| + |\delta|$ the *size* of A.

A run is a word over δ (considered as an alphabet). Let denote by $\rho(A) \subseteq \delta^*$, resp. $\rho^{acc}(A) \subseteq \delta^*$, the set of runs, resp. accepting runs, of A, that is, $\rho(A) = \bigcup_{u \in \Sigma^*} \rho(A, u)$ and $\rho^{acc}(A) = \bigcup_{u \in \Sigma^*} \rho^{acc}(A, u)$. Clearly, these two sets are regular languages over δ.

Proposition 3.1.2. *Let $A = (Q, I, F, \delta)$ be an ϵ-NFA over the alphabet Σ, then the set $\rho(A)$ and $\rho^{acc}(A)$ are regular languages over the alphabet δ.*

Some states of an ϵ-NFA are useless. A state q of an ϵ-NFA A is *accessible*, resp. *co-accessible*, if there exists a run of A from an initial state to q, resp. from q to a final state. An ϵ-NFA is *trimmed* if all its states are accessible and co-accessible. One can trim an ϵ-NFA in linear time.

The *realtime* non-deterministic finite state automata, or simply non-deterministic finite state automata, are ϵ-NFA with no ϵ-transition.

Definition 3.1.3 (NFA). *An ϵ-NFA $A = (Q, I, F, \delta)$ such that $\delta \subseteq Q \times \Sigma \times Q$ is called a realtime non-deterministic finite state automaton, or simply a non-deterministic finite state automaton (NFA).*

With each transition, NFA processes one input letter. The length of a run is therefore equal to the length of the input word and so for a given NFA $A = (Q, I, F, \delta)$ and a word u there is at most $|\delta|^{|u|}$ runs of A over u.

The class of NFA is as expressive as the class of ϵ-NFA, moreover, there exists an algorithm for removing ϵ-transitions of ϵ-NFA.

Proposition 3.1.4 (ϵ-removal). *For any ϵ-NFA one can effectively construct in* PTime *an equivalent NFA.*

Deterministic finite state automata (DFA) are NFA such that, at any time, there is at most one transition compatible with the input letter, *i.e.* the transition relation is a partial function from $Q \times \Sigma$ to Q. As a consequence there is at most one run per input word.

Definition 3.1.5 (DFA)**.** *A* NFA $A = (Q, I, F, \delta)$ *is deterministic if* $|I| = 1$ *and whenever* $(q, a, q') \in \delta$ *and* $(q, a, q'') \in \delta$ *we have* $q' = q''$*.*

Any NFA, and thus any ϵ-NFA, is equivalent to a DFA. Given an NFA, the procedure to construct an equivalent DFA is called determinization, it is done with the standard subset construction and yields an exponentially bigger automaton.

Proposition 3.1.6 (Determinization)**.** *For any* NFA A *one can effectively construct an equivalent* DFA *whose size is at most exponential in the size of the original* NFA*.*

As a consequence, these three classes of automata, ϵ-NFA, NFA and DFA, are equivalent and characterize the class of regular languages.

Finally we recall the notion of ambiguity.

Definition 3.1.7. *Let* $k \in \mathbb{N}$*, an* ϵ-NFA A *is unambiguous, resp.* k*-ambiguous, iff for all words* $u \in \Sigma^*$ *we have* $|\rho^{acc}(A, u)| \leq 1$*, resp.* $|\rho^{acc}(A, u)| \leq k$*.*

Trivially, DFA are unambiguous automata. However an unambiguous automaton may not be deterministic.

Closure properties

The class of regular languages is closed under all Boolean operations, concatenation, Kleene star, mirror image, and morphisms. The closure is effective when the language are given by ϵ-NFA. An automaton representing the complement is obtained using *determinization*, one representing the intersection is obtained with the standard *product construction*, and one that recognizes the union is build using *non-determinism.*

Let us recall the product construction for NFA as we use it several times later on. Let $A_1 = (Q_1, I_1, F_1, \delta_1)$ and $A_2 = (Q_2, I_2, F_2, \delta_2)$ be two NFT, the product of A_1 and A_2 is the NFT $A = A_1 \times A_2 = (Q_1 \times Q_2, I_1 \times I_2, F_1 \times F_2, \delta_1 \otimes \delta_2)$ where the synchronized product of the transition relations $\delta_1 \otimes \delta_2$ is defined as: $t = ((q_1, q_2), a, (q_1', q_2')) \in \delta_1 \otimes \delta_2$ if and only if $t_1 = (q_1, a, q_1') \in \delta_1$ and $t_2 = (q_2, a, q_2') \in \delta_2$. We write $t = t_1 \otimes t_2$.

There is a one-to-one correspondence between runs of A and pairs formed by a run of A_1 and a run of A_2. We have $\rho = t_0 \dots t_n \in \rho(A, u)$ if and only if there exist $\rho_1 = t_0' \dots t_n' \in \rho(A_1, u)$ and $\rho_2 = t_0'' \dots t_n'' \in \rho(A_2, u)$ with $t_i = t_i' \otimes t_i''$ for all $i \leq n$. In that case we write $\rho = \rho_1 \otimes \rho_2$. Clearly, $L(A) = L(A_1) \cap L(A_2)$.

Proposition 3.1.8 (closure properties)**.** *The class of regular languages is closed under union, intersection, complement, concatenation, Kleene star, mirror image, and morphisms. When the languages are given as* ϵ-NFA *then the closure are effective. The construction is exponential for the complement and polynomial for all other operations.*

The following proposition is a useful characterization of regular languages. It shows that any class of languages that contains the empty set, the singletons and is closed under union, Kleene star and concatenation does contain all regular languages.

Proposition 3.1.9. *The class of regular languages over* Σ *is the smallest set that contains the empty set and the languages* $\{a\}$ *for any* $a \in \Sigma$*, and that is closed under union, Kleene star, and concatenation.*

Decision problems

The main decision problems for finite state automata are decidable.

Proposition 3.1.10 (Decision problems)**.** *The following problems are decidable.*

- *The emptiness and membership problems for* ϵ-NFA *(and thus for* NFA *and* DFA*) are decidable in* PTime.
- *The universality, inclusion and equivalence problems for* ϵ-NFA *and* NFA *are* PSpace-c*. They are decidable in* PTime *for* DFA.
- *For a fixed* k*, one can check in* PTime *if an* NFA *is* k*-ambiguous.*

Let us explain how to use the product to check ambiguity of a given NFA $A = (Q, I, F, \delta)$, *i.e.* whether for all word u, there exists at most one accepting run of A over u.

The *square* of A, denoted by A^2 is the product of A with itself, *i.e.* $A^2 = A \times A = (Q \times Q, I \times I, F \times F, \delta \otimes \delta)$. Any run, resp. accepting run, of A^2 over u is in a one-to-one correspondence with pairs of runs, resp. accepting runs, of A over u. For A to be unambiguous, any accepting run of A^2 should be of the form $(q_0, q_0)(q_1, q_1) \dots (q_n, q_n)$. In other words, the trimmed automaton of A^2 must contain only states of the form (q, q). This condition is necessary and sufficient and can be checked in PTime.

3.1.2 Finite State Transducers

A finite state transducer is a finite state automaton, called the *underlying automaton*, and an output mechanism. The output is defined by a morphism that associates a word with each transition of the underlying automaton. A run of a given transducer over an input word is a run of its underlying automaton over this input word, it is a sequence of transitions. The output of a run is the image of this run by the output morphism, in other words it is the concatenation of the output of the successive transitions of this run.

In this section, we let Σ be the input alphabet and Δ be the output alphabet.

Definition 3.1.11 (Finite State Transducers). *A finite state transducer with* ϵ-transitions *(*ϵ-NFT*) from* Σ *to* Δ *is a pair* $T = (A, \Omega)$ *where* $A = (Q, I, F, \delta) \in$ ϵ-NFA *is the underlying automaton and* Ω *is a morphism from* δ *to* Δ^* *called the output.*

We denote by NFT*, respectively* DFT*, the class of realtime finite state transducers, respectively the class of deterministic finite state transducers, that is, the class of* ϵ-NFT *such that the underlying automaton is an* NFA*, respectively a* DFA.

Let $t \in \delta$, $\Omega(t) \in \Delta^*$ is the output of transition t. Let $u \in \Sigma^*$ and $\rho = t_1 \dots t_n \in \rho^{acc}(A, u)$, we say that $v = \Omega(\rho) = \Omega(t_1) \dots \Omega(t_n) \in \Delta^*$ is the output of the run and that v is an image of u. We write $T \models q \xrightarrow{u/v} q'$ (or simply $q \xrightarrow{u/v} q'$) if there is a run $\rho \in run(A, u)$ from q to q' with $v = \Omega(\rho)$. The transduction defined by T is the relation $R(T) = \{(u, v) \in \Sigma^* \times \Delta^* \mid q \xrightarrow{u/v} q' \wedge q \in I \wedge q' \in F\}$. The domain ($dom$(T)), resp. range ($range$(T)), of T is the domain, resp. range, of $R(T)$.

We say that a transduction $R \subseteq \Sigma^* \times \Delta^*$ is a finite state , resp. realtime finite state, resp. deterministic finite state, transduction if there exists an ϵ-NFT, resp. NFT, rep. DFT, T such that $R(T) = R$. We denote by $R(\epsilon\text{-}\mathsf{NFT})$, resp. $R(\mathsf{NFT})$, $R(\mathsf{DFT})$, the class of finite state , resp. realtime finite state, resp. deterministic finite state, transductions.

Example 3.1.12. *Let* $\Sigma = \{a, b\}$ *and* $\Delta = \{c\}$ *be two alphabets. The transducer* T_1*, depicted in Figure 3.1 (left), defines the following transduction:*

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

The initial states, here q_1 *only, are depicted with an incoming arrow. The final states,* q_1 *and* q_2*, are depicted with a double circle. The edges of the graph*

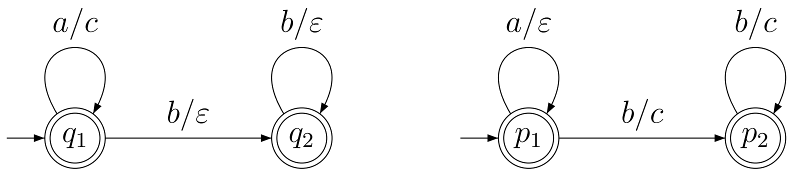


Figure 3.1: Two finite state transducers: T_1 (left) and T_2 (right).

represent the transitions. Each edge is labelled with the input and output of the corresponding transition, that is, a label a/bc represents a transition t with input letter a and output $\Omega(t) = bc$.

The transducer T_2, depicted in Figure 3.1 (right), has the same underlying automaton and defines the transduction:

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

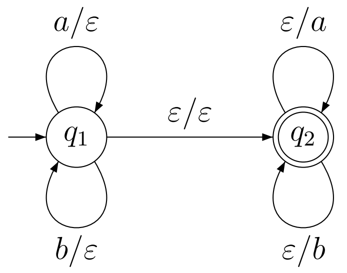


Figure 3.2: A finite state transducer T_∞ on $\Sigma = \Delta = \{a, b\}$

Example 3.1.13. *Let $\Sigma = \Delta = \{a, b\}$. The transducer T_∞, depicted in Figure 3.2, defines the universal transduction over Σ:*

$$R(T_\infty) = \Sigma^* \times \Delta^*$$

It first reads the input word with no output (state q_1), and then it switches to state q_2 and produces the output word with ϵ-transitions.

In contrast with what happens with the three classes ϵ-NFA, NFA, and DFA, that all define the same class of regular languages, here the three classes of transducers, ϵ-NFT, NFT and DFT, define a strict hierarchy of classes of transductions. Indeed, there can be an infinite number of image of a word u by an ϵ-NFT, for example the ϵ-NFT of Example 3.1.13 maps any input word to any word of Δ^*. On the other hand, for a given NFT T and a word u, there is at most at most $n = |\delta|^{|u|}$ runs of T over u (where δ is its transition relation), and therefore at most n images of u. Finally, when the transducer is a DFT, there is at most one run and therefore one image per input word u. Therefore we have: $\mathsf{R}(\mathsf{DFT}) \subsetneq \mathsf{R}(\mathsf{NFT}) \subsetneq \mathsf{R}(\epsilon\text{-}\mathsf{NFT})$.

Sometimes proofs and constructions gain in readability by using a slightly more general definition of ϵ-NFT. The generalized machines differ on the definition of the output of a transition. Their output function associates with each transition a finite set of outputs, that is a finite subset of Δ^*, instead of, for ϵ-NFT, a single output. In this document, we call these transducers *generalized* ϵ-NFT.

Definition 3.1.14. *A generalized ϵ-NFT T is a pair $T = (A, O)$ where $A = (Q, I, F, \delta)$ is an ϵ-NFA and O is a function from δ to finite subsets of Δ^*.*

A run of T is a run of the underlying automaton A. The set of outputs of a run $\rho = t_1 \dots t_n \in \delta^*$ is $O(t_1) \cdot O(t_2) \cdot \dots \cdot O(t_n)$. All notations and notions carry over from ϵ-NFT. In some cases, it simplifies the notations to write $\delta(q, a, q')$ instead of $O((q, a, q'))$ to denote the set of outputs.

The class of transductions definable by generalized ϵ-NFT is exactly the class of transductions defined by ϵ-NFT. We informally present this construction. Intuitively, it amounts to split each state according to the number of possible outputs its incoming transitions have. Let n be the maximum number of outputs of the incoming transitions of state q (that is, an incoming transition has n outputs, and no incoming transition has more than n outputs), then q is split into the states $(q, 1), \dots (q, n)$. An incoming transition with $m \leq n$ outputs is transformed into m transitions with one output each and targeting state $(q, 1)$ to (q, m). All outgoing transitions of q are duplicated for each state (q, i). Clearly this construction is polynomial in the size of T.

Proposition 3.1.15. *Given a generalized ϵ-NFT one can effectively construct in polynomial time an equivalent ϵ-NFT.*

The domain and range of an ϵ-NFT $T = (A, \Omega)$ are regular languages. Indeed, the domain of T is the language of its underlying automaton A. The range is the image of the set of its accepting runs by the morphism Ω. The set of accepting runs being regular (Proposition 3.1.2) and the set of regular languages being closed under morphism (Proposition 3.1.8), the range is regular.

Proposition 3.1.16 (Domain and Range). *Let $T \in \epsilon$-NFT, we have $dom(T) \in \mathsf{REG}$ and $range(T) \in \mathsf{REG}$. Moreover, on can effectively compute in polynomial time, an ϵ-NFT that represents $dom(T)$, resp. $range(T)$.*

Closure Properties

Union. Closure under union can be obtained with non-determinism, therefore the classes of ϵ-NFT and NFT are closed under union. Clearly, the class of DFT is not since any word has at most one image, which is not true in general when taking the union of two deterministic finite state transductions.

Inverse. Trivially, the classes of NFT and DFT are not closed under inverse. Indeed, any word has a finite number of images, but an infinite number of words can be mapped (by an NFT or a DFT) onto the same word, for example the transduction that maps any input word onto the empty word. This is not the case for ϵ-NFT and in fact it is easy to show that the class of ϵ-NFT is closed under inverse. The idea is to simply swap the input with the output of each transition. Let $T = (A, \Omega)$ be an ϵ-NFT with $A = (Q, I, F, \delta)$, wlog we can suppose that for each $t \in \delta$ we have $\Omega(t) \in \Sigma \cup \{\epsilon\}$. Let $T^{-1} = (A', \Omega')$ be an ϵ-NFT where $A' = (Q, I, F, \delta')$ and δ' is defined as follows. For each $t = (q, a, q') \in \delta$, where $a \in \Sigma \cup \{\epsilon\}$ we pose $t' = (q, \Omega(t), q') \in \delta'$ and $\Omega'(t') = a$. One can easily check that $R(T)^{-1} = R(T^{-1})$.

Composition. The classes of ϵ-NFT, NFT and DFT are closed under composition. Let us show how to proceed in the case of NFT. Let $T_1 = (A_1, \Omega_1)$ be an NFT from Σ_1 to Σ_2 and $T_2 = (A_2, \Omega_2)$ an NFT form Σ_2 to Δ, where $A_i = (Q_i, I_i, F_i, \delta_i)$ (for $i \in \{1, 2\}$). We define the generalized NFT $T = (A, O)$ with $A = (Q, I, F, \delta)$ where $Q = Q_1 \times Q_2$, $I = I_1 \times I_2$, $F = F_1 \times F_2$ and δ is defined as follows. For each transition $t_1 = (q_1, a, q_1') \in \delta_1$ with $v = \Omega_1(t_1) \in \Sigma_2$, we add the following transitions to T:

$$\delta((q_1, q_2), a, (q_1', q_2')) = \{w \in \Delta^* \mid T_2 \models q_2 \xrightarrow{v/w} q_2'\}$$

It is routine to check that $R(T) = R(T_2) \circ R(T_1)$. By Proposition 3.1.15, one can construct an NFT equivalent to a given generalized NFT in polynomial time,

	T^{-1}	$\overline{T}$	$T_1 \cup T_2$	$T_1 \cap T_2$	$T_1 \circ T_2$
ϵ-NFT	yes	no	yes	no	yes
NFT	no	no	yes	no	yes
DFT	no	no	no	no	yes

Table 3.1: Closure Properties for ϵ-NFT, NFT and DFT.

therefore, one can effectively construct T in polynomial time.

Non-closure

While the class of regular languages is closed under pretty much any operation, it is not the case for finite state transductions. None of the three classes of finite state transductions is closed under intersection nor under complement.

Intersection. Consider the transducers T_1 and T_2 of Example 3.1.12. 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 regular. Therefore this transduction is not regular. As T_1 and T_2 are deterministic this shows the non-closure for the three classes of transducers.

Complement. The class of NFT and DFT are not closed under complement for the same reason that they are not closed under inverse: any word has a finite number of images, therefore any word in the complementary relation has an infinite number of images, and can therefore not be represented has an NFT or DFT. Non closure under complement for ϵ-NFT is a direct consequence of the fact that it is closed under union but not under intersection.

The closure properties are summarized in Table 3.1 and in the following proposition.

Proposition 3.1.17 (Closure). *The class of* ϵ-NFT *is closed under union, composition and inverse. The class of* NFT *is closed under union and composition. The class of* DFT *is closed under composition. All closures are effective and can be computed in* PTIME.

Decision Problems

The relation defined by T is empty if and only if its domain is empty. The membership of a pair $(u, v) \in \Sigma^* \times \Delta^*$ can be tested by testing the membership of v in the image of u by T. A NFA representing the image of u by T can be computed in linear time.

Proposition 3.1.18. *The emptiness and membership problems for ϵ-NFT are decidable in* PTime.

The type checking problem for ϵ-NFT against NFA is also decidable. It simply amounts to testing the inclusion of the image of the input language by the transducer (a regular language) into the output language, this can be done in PSpace. As there is a DFT that implements the identity, the problem is as hard as the inclusion problem for NFA which is PSpace-c. When the output language is given as a DFA the procedure above is in PTime.

Proposition 3.1.19. *The type checking problem for ϵ-NFT, NFT, or DFT against NFA, resp. DFA, is* PSpace-c*, resp. decidable in* PTime.

For non-deterministic ϵ-NFT the equivalence, inclusion and universality problems are all undecidable. For NFT the universality is trivially false, while equivalence and inclusion are undecidable. We present a proof, first published in 1968 by Griffiths, it is an elegant reduction of the PCP.

Theorem 3.1.20 ([Gri68]). *The equivalence and inclusion problems are undecidable for ϵ-NFT and NFT. The universality problem is undecidable for ϵ-NFT.*

Proof. We first prove the undecidability of the equivalence and inclusion problems for NFT.

Let Σ and Δ be two alphabets, g, h two morphisms from Σ into Δ^*. Consider the PCP instance $P = (g, h)$ (see Definition 2.0.1) and let construct two NFT T and T_M such that P has a solution if and only if T and T_M are equivalent if and only if T_M is included in T.

Let M be the maximal length of the images of the elements of Σ by g and h, that is, $M = \max\{\max(|g(a)|, |h(a)|) \mid a \in \Sigma\}$.

First, let us consider the relation R_M that associates any word $u \in \Sigma^*$ with any word of length at most $M|u|$, that is:

$$R_M = \{(u, v) \mid |v| \leq M|u|\}$$

We define a generalized NFT T_M that implements the relation R_M. Let $T_M = (A_M, O_M)$ with $A_M = (Q_M, I_M, F_M, \delta_M)$, where $Q_M = I_M = F_M = \{q\}$, and for any $a \in \Sigma$ we set $\delta(q, a, q) = \{v \mid |v| \leq M\}$ (recall that we write $\delta(q, a, q)$ for $O((q, a, q))$. Clearly $R(T_M) = R_M$.

Second, consider the relation R_g such that the images of any word $u \in \Sigma^*$ are all the words of length at most $|u|M$ but $g(u)$, that is:

$$R_g = \{(u, v) \in \Sigma^* \times \Delta^* \mid |v| \leq |u|M \wedge v \neq g(u)\}$$

and similarly,

$$R_h = \{(u, v) \in \Sigma^* \times \Delta^* \mid |v| \leq |u|M \wedge v \neq h(u)\}$$

We construct two generalized NFT T_g and T_h such that $R(T_g) = R_g$ and $R(T_h) = R_h$. Let us show the construction for $T_g = (A_g, O_g)$ where $A_g = (Q_g, I_g, F_g, \delta_g)$. Let $Q_g = \{q_0, q_-, q_=, q_+\}$, $I_g = \{q_0\}$, $F_g = \{q_-, q_=, q_+\}$ and δ_g is defined as follows. Consider $a \in \Sigma$ and let $g(a) = a_1 \dots a_l \in \Sigma^*$. We add the following transitions to T_g:

$$\begin{array}{lcll}
\delta(q_0, a, q_0) & = & \{a_1 \dots a_l\} & \\
\delta(q_0, a, q_-) & = & \{v \in \Delta^* \mid |v| < l\} & \\
\delta(q_0, a, q_=) & = & \{v \in \Delta^* \mid |v| = l \wedge v \neq g(a)\} & \\
\delta(q_0, a, q_+) & = & \{v \in \Delta^* \mid l < |v| \leq M\} & \\
\delta(q_=, a, q_=) & = & \{v \in \Delta^* \mid |v| = l\} & \\
\delta(q_-, a, q_-) & = & \{v \in \Delta^* \mid |v| \leq l\} & \\
\delta(q_+, a, q_+) & = & \{v \in \Delta^* \mid l \leq |v| \leq M\} & \\
\delta(q, a, q') & = & \emptyset & \text{otherwise}
\end{array}$$

Intuitively, the semantics of this automata is as follows. Suppose the automaton has read the word u, there are three possible cases: (i) it is in state q_0, then the output up to now is $g(u)$, however q_0 not being a final state the automaton must, at some point, switch to q_-, q_+ or $q_=$, in order to accept the current transduction, this is necessary as $(u, g(u))$ should not belong to $R(T_g)$; (ii) it is in state q_-, resp. q_+, it means the output word of this run is shorter, resp. longer, than $g(u)$ and the transducer ensures (by, from now on, on input b outputting words of length shorter, resp. longer, than the length of $g(b)$) that whatever sequel of the input is, this property will hold on the whole run; (iii) when T'_g is in state $q_=$, it means the output word up to now, as a length equal to the length of $g(u)$, but both words, $g(u)$ and the output, are different, therefore

	emptiness / membership	type checking (vs NFA)	equivalence / inclusion	universality
ϵ-NFT	PTIME	PSPACE-C	undec	undec
NFT	PTIME	PSPACE-C	undec	-
DFT	PTIME	PSPACE-C	PTIME	-

Table 3.2: Decision problems for ϵ-NFT, NFT and DFT.

whatever the sequels of the input and the output are, the transduction belongs to R_u. A careful (but easy) analysis shows that this separation of cases, implies that $R(T_g) = R_g$. A complete and formal proof of the correctness of this construction is given in [Gri68].

Consider the NFT obtained as the union of both transducers: $T = T_g \cup T_h$. It recognizes the relation that translates any word $u \in \Sigma^*$ into any word $w \in \Delta^*$ provided $w \neq g(u)$ or $w \neq h(u)$, that is:

$$R(T) = \{(u, v) \in \Sigma^* \times \Delta^* \mid (v \neq g(u) \vee v \neq h(u)) \wedge |v| \leq |u|M\}$$

Clearly, T is equivalent to T_M if and only if T_M is included into T if and only if P has a no solution. This completes the proof of the undecidability of the equivalence and inclusion problems of NFT.

The undecidability of these two problems, trivially, carries over to the class of ϵ-NFT. The proof of the undecidability of the universality is very similar. It uses the unbounded version of the relations R and R_M, that is, let the

$$R_\infty = \Sigma^* \times \Delta^*$$

and

$$R' = \{(u, v) \in \Sigma^* \times \Delta^* \mid v \neq g(u) \vee v \neq h(u)\}$$

It is not difficult to construct two ϵ-NFT, T_∞ and T', that implements these relations. Then, T' is universal if and only if T_∞ is included into T' if and only if P has a no solution. □

The results concerning the decision problems for ϵ-NFT, NFT and DFT are summarized in Table 3.2. As a consequence of the undecidability of the inclusion and equivalence, one has to look for subclasses of NFT with decidable equivalence and/or inclusion problems. The class of DFT, but also the class formed by the functional and k-valued finite state transductions, presented in the next section, are example of such classes.

Functional NFT

The class of functional transductions defined by NFT enjoys an important property. For that class the equivalence and the inclusion problems are decidable.

Transductions defined by DFT are trivially functional. However some NFT that are not deterministic define functional transductions, furthermore, some functional NFT definable transductions are not definable by a deterministic NFT. For example, the following functional transduction is definable by a NFT but it is not *determinizable*:

$$R_{swap} = \{(ua, au) \mid u \in \Sigma^*\} \cup \{(ub, bu) \mid u \in \Sigma^*\}$$

In the same spirit, transductions defined by unambiguous NFT are obviously functional. However, some NFT might be ambiguous but still may define functional transduction. In that case, clearly all accepting runs over a same input must produce the same output. However, unambiguous NFT are more powerful than DFT, it has been proved [EM65] that all functional NFT transductions can be defined by an unambiguous NFT.

Decidability of functionality for NFT was first published by Schutzenberger [Sch75]. His proof relies on a pumping argument. It shows that if the transducer is not functional then there exists an input word, whose size is polynomial in the size of the transducer, that has two different associated output words. This pumping argument shows membership in NPTime. Gurari and Ibarra showed that functionality is decidable in PTime [GI83]. They proceed by a reduction to the emptiness of counter automata.

Recently, Béal et al., presented an elegant and direct construction to decide in PTime functionality of NFT [BCPS03]. This construction is based on the squaring construction for deciding ambiguity of NFA (Proposition 3.1.10). We recall briefly their procedure.

Let $T = (A, \Omega)$ be an NFT. Suppose there are two different runs for a given input word, say u. If T is functional both of these runs must produce the same output. However, they may produce their output at different "speed", *e.g.* the first run may produce its output faster at first then decelerates, while the second run may start outputting slowly but catches up afterwards. After reading a prefix of u the first run is in a state q has output v, while the second run is in state p and produced w. The delay $\Delta(v, w)$ is the difference between these outputs, it is defined as the pair $((v \wedge w)^{-1}v, (v \wedge w)^{-1}w)$, *i.e.* it is the pair of output words from which the longest common prefix has been removed. If there exists

z with $p \xrightarrow{z/.} p_f$ and $q \xrightarrow{z/.} q_f$, where $p_f, q_f \in F$, we say that the pair of states (p, q) is co-accessible in T. In that case, if T is functional, then necessarily either v is a prefix of w or w is a prefix of v, and in that case the first or the second component of the delay must be the empty word ϵ.

The procedure of Béal et al. is based on the following observation. If T is functional then there exists a unique delay for each pair of states p, q (provided the pair is co-accessible). Indeed, let $u_1, u_2, u_3 \in \Sigma^*$ such that

$$\begin{array}{ll} q_0 \xrightarrow{u_1/v_1} q & q_0' \xrightarrow{u_2/v_2} q \\ p_0 \xrightarrow{u_1/w_1} p & p_0' \xrightarrow{u_2/w_2} p \\ q \xrightarrow{u_3/v_3} q_f & p \xrightarrow{u_3/w_3} p_f \end{array}$$

for some $p_0, q_0, p_0', q_0' \in I$, $p_f, q_f \in F$, and $p, q \in Q$. If $\Delta(v_1, w_1) \neq \Delta(v_2, w_2)$ then T is not functional. Indeed, either $v_1v_3 \neq w_1w_3$ or $v_2v_3 \neq w_2w_3$. Therefore the *delay* at each pair of accessible and co-accessible states in the square of A should be unique. Moreover if both states are final, then the delay must be null.

The square of T is an NFT $T^2 = (A^2, \Omega')$ whose underlying automaton is the square of the underlying automaton of T and whose output Ω' is defined as follows. The output alphabet is $O_T^2 = O_T \times O_T$ where $O_T = \Omega(\delta)$ is the set of output words of each transition, each word is considered as a single letter. The output of a transition $t = t_1 \otimes t_2$ is the pair $\Omega'(t) = (\Omega(t_1), \Omega(t_2))$.

The decision procedure first constructs the square of T, trims it, and associates a null delay (*i.e.* $\Delta(p_0, q_0) = (\epsilon, \epsilon)$) with each initial state. If $(x_1, x_2) \in \Delta^* \times \Delta^*$ is the delay at state (q_1, p_1) and there is a transition $(q_1, p_1) \xrightarrow{a/(y_1,y_2)} (q_2, p_2)$, then the delay at (q_2, p_2) is $\Delta(x_1y_1, x_2y_2)$ (this can also be written $\Delta((x_1, x_2)(y_1, y_2)))$. This delay must be unique for each state of the square, this can be verified by a breadth first traversal of the graph of the square, *i.e.* the complexity of the procedure is linear in the size of the square.

Let denote by $\Delta(p, q)$ the unique delay computed for state (p, q), we can show that if for all transitions in the square $(p, q) \xrightarrow{u/(y_1,y_2)} (p', q')$ we have $\Delta(p', q') = \Delta(\Delta(p, q)(y_1, y_2))$, and if $p', q' \in F$ the delay is (ϵ, ϵ), then T is functional.

Theorem 3.1.21 (Functionality). *It is decidable in* PTIME *whether an* NFT *is functional.*

A procedure deciding whether a transducer is functional yields a procedure for deciding equivalence of functional transducers. One first has to check whether the domains of the transducers are equivalent. Then, one verified that the union of the transducers is functional.

Theorem 3.1.22 (Equivalence of functional)**.** *The equivalence and inclusion problems for functional* NFT *are* PSPACE-C.

Proof. Let T_1, T_2 be two functional NFT. $R(T_1) \subseteq R(T_2)$ if and only if the following two conditions are met: (i) $dom(T_1) \subseteq dom(T_2)$, and (ii) $R(T_1) \cup R(T_2)$ is functional.

The domains are represented by the underlying automata which are NFA. By Proposition 3.1.10, testing the inclusion of NFA is PSPACE-C. By Proposition 3.1.17, one can construct in PTIME an NFT T that implements the union and such that $|T| = |T_1| + |T_2|$. By Proposition 3.1.21 testing functionality of T is in PTIME. This concludes the proof of the decidability of the inclusion.

Testing the equivalence can be done simply by testing mutual inclusions, that is $R(T_1) = R(T_2)$ if and only if $R(T_1) \subseteq R(T_2)$ and $R(T_2) \subseteq R(T_1.)$. □

Deterministic Transductions

A sub-sequential finite state transducer is a deterministic finite state transducer that has the additional capacity to output a word after reading the last symbol of the word.

Definition 3.1.23. *A sub-sequential transducer T from Σ to Δ is a pair (T', Ω_f) where $T' = (A, \Omega)$ is a* DFT *and Ω_f is a morphism from F to Δ.*

The transduction defined by T is:

$$R(T) = \{(u, vw) \mid T' \models q_0 \xrightarrow{u,v} q_f \wedge q_0 \in I, \wedge q_F \in F \wedge \Omega_f(q_f) = w\}$$

Strangely, not all transductions definable by sub-sequential transducers are definable by DFT. For example the following transduction is sub-sequential but not deterministic:

$$R_{odd} = \{(u, ua) \mid u \in \Sigma^* \wedge |u| \text{ is odd}\} \cup \{(u, ub) \mid u \in \Sigma^* \wedge |u| \text{ is even}\}$$

Obviously, all deterministic NFT transductions are sub-sequential. In fact these notions are equivalent when one assumes that every word terminates with a special end of the word symbol $\$ \notin \Sigma$. Sub-sequential transductions correspond exactly to the transductions that can be performed with an amount of memory that is independent of the input word (See Chapter 7).

As we have seen above, not all functional NFT transductions can be determinized nor defined by a sub-sequential transducer. However, it is decidable whether a functional NFT transduction is determinizable, resp. is equivalent to a sub-sequential transducer.

Sub-sequentializable NFT are characterized by the so called *twinning property* [Cho77], which is decidable in PTIME [WK95]. Intuitively, the twinning property requires that two runs on the same input cannot have arbitrary large difference between their outputs. We recall the twinning property as it is worded in [BC02].

Definition 3.1.24 (Twinning property for NFT of [BC02])**.** *Let $T = (Q, I, F, \delta)$ be a trimmed* NFT*. T satisfies the twinning property if for all $q_0, q_0' \in I$, for all $q, q' \in Q$, for all words $u_1, v_1, w_1, u_2, v_2, w_2 \in \Sigma^*$, if:*

$$q_0 \xrightarrow{u_1/v_1} q \xrightarrow{u_2/v_2} q \qquad q_0' \xrightarrow{u_1/w_1} q' \xrightarrow{u_2/w_2} q'$$

Then either $v_2 = w_2 = \epsilon$, or the following holds:

(i) $|v_2| = |w_2|$

(ii) $v_1(v_2)^\omega = w_1(w_2)^\omega$

Note that in contrast to the condition for functionality, the twinning property does not ask the pair of states (q, q') to be co-accessible (but both of these states must be co-accessible).

We slightly rephrase the twinning property to better fit our need in Chapter 7.

Definition 3.1.25 (Twinning Property for NFT (delay))**.** *Let $T = (Q, I, F, \delta)$ be a trimmed* NFT*. T satisfies the twinning property (delay) if for all $q_0, q_0' \in I$, for all $q, q' \in Q$, for all words $u_1, v_1, w_1, u_2, v_2, w_2 \in \Sigma^*$, if:*

$$q_0 \xrightarrow{u_1/v_1} q \xrightarrow{u_2/v_2} q \qquad q_0' \xrightarrow{u_1/w_1} q' \xrightarrow{u_2/w_2} q'$$

Then $\Delta(v_1, w_1) = \Delta(v_1v_2, w_1w_2)$.

The two definitions are equivalent, as shown by the next lemma.

Lemma 3.1.26. *Definitions 3.1.24 and 3.1.25 are equivalent.*

Proof. First suppose that Definition 3.1.24 holds. If $v_2 = w_2 = \epsilon$, then clearly $\Delta(v_1, w_1) = \Delta(v_1v_2, w_1w_2)$. Otherwise $|v_2| = |w_2|$ and $v_1(v_2)^\omega = w_1(w_2)^\omega$. Then necessarily $v_1 \preceq w_1$ (i.e. v_1 is a prefix of w_1) or $w_1 \preceq v_1$. Wlog suppose that $v_1 \preceq w_1$, i.e. $w_1 = v_1 w_1'$ for some w_1'. Therefore $\Delta(v_1, w_1) = (\epsilon, w_1')$, and $\Delta(v_1v_2, w_1w_2) = \Delta(v_2, w_1'w_2)$.

We now prove that $\Delta(v_2, w_1'w_2) = (\epsilon, w_1')$. Since $v_1(v_2)^\omega = w_1(w_2)^\omega$, we have $(v_2)^\omega = w_1'(w_2)^\omega$. Therefore $v_2w_1'(w_2)^\omega = v_2^\omega = w_1'(w_2)^\omega$. Since $|v_2| = |w_2|$, we get $v_2w_1' = w_1'w_2$, from which we have $\Delta(v_2, w_1'w_2) = \Delta(v_2, v_2w_1') = (\epsilon, w_1')$.

Conversely, suppose that Definition 3.1.25 holds and $v_2w_2 \neq \epsilon$. Since $\Delta(v_1, w_1) = \Delta(v_1v_2, w_1w_2)$, we have either $v_1 \preceq w_1$ or $w_1 \preceq v_1$. Wlog suppose that $v_1 \preceq w_1$, i.e. $w_1 = v_1w_1'$ for some w_1'. Therefore we have:

$$\Delta(v_1, w_1) = (\epsilon, w_1') = \Delta(v_1v_2, w_1w_2) = \Delta(v_2, w_1'w_2)$$

Consequently, $v_2 \preceq w_1'w_2$, and in particular, $w_1'w_2 = v_2w_1'$, which gives us the following series of equalities:

$$w_1(w_2)^\omega = v_1w_1'(w_2)^\omega = v_1v_2w_1'(w_2)^\omega = v_1(v_2)^2w_1'(w_2)^\omega = \cdots = v_1(v_2)^\omega$$

□

The twinning property characterizes the sub-sequentializable transducers.

Proposition 3.1.27 ([Cho77]). *Let T be an* NFT*, T satisfies the twinning property if and only if it is sub-sequentializable.*

Let us explain briefly the idea behind the twinning property. Suppose the twinning property does not hold for a NFT T. Then on input words of the form $u_1u_2^n$ the two possible output are $v_1v_2^n$ and $w_1w_2^n$. The delay $\Delta(v_1v_2^n, w_1w_2^n)$ increases with n. This delay is the output words of each run that can not be produced until the ambiguity is lifted, *i.e.* until a symbol is read that is incompatible with either the first or the second run. This delay must be stored and can be arbitrarily large. Therefore T' can not be sub-sequential as sub-sequential implies a finite memory (one may use a simply pumping argument).

Let T be an NFT that satisfies the twinning property, an equivalent sub-sequential transducer T_d is built as follows. Its states are set of pairs $(q, w) \in Q \times \Delta^*$ where q is a state of T and w is the output that T_d has not produced yet, but that T would have produced with a run in q, *i.e.* its the delay associated with state q. Suppose T_d is in state P and it reads a letter a. To compute the next state P' one first define the set P'':

$$P'' = \{(q, vw) \mid \exists (p, v) \in P \wedge T \models p \xrightarrow{a/w} q\}$$

Let $x \in \Delta^*$ be the longest common prefix of words u appearing in P'', *i.e.* x is the longest common prefix of the words in $\{u \mid (p,u) \in P''\}$. The next state P' is:

$$P' = \{(q, x^{-1}u) \mid \exists (q,u) \in P''\}$$

and the output of the transition is x. The initial state is $I_d = \{(i,\epsilon) \mid i \in I\}$ where I is the initial states of T. The set of finial states is the set P such that there is at least one pair (q,u) with $q \in F$, where F are the final states of T.

This construction is based on the fact that, for sub-sequentializable NFT, the delay u can be bounded. Indeed, on can show that if the twinning property holds, then for any $(p,u) \in P$ and P accessible in T_d from the initial state I_d, then $|u| \leq 2Mn^2$. where n is the number of states of T and M is the maximal length of the output of the transition of T. Therefore, in that case, the construction above defines a sub-sequential transducer. Note that the output morphism on final states, Ω_f, of T_d (that differentiate sub-sequential and deterministic) is needed as it might be the case that a final state has a pair $(q,u) \in F \times \Delta^*$ with u non empty, in that case the word u must be produced.

Finally, one can decide in PTime whether the twinning property holds for a given NFT. Deciding if T is determinizable is done similarly [Ber09].

Theorem 3.1.28. *It is decidable in* PTime *whether a* NFT *is determinizable, resp. sub-sequentializable.*

k-valued NFT

Finally note that all results about functional NFT transductions generalized to k-valued transductions. Notably, it is decidable whether a NFT transduction is k-valued and their equivalence and inclusion problems are decidable

Theorem 3.1.29 (k-valuedness)**.** *[GI83, Web89, SdS08] It is decidable in* PTime *whether an* NFT *T is finite valued, that is, whether there exists a $k \in \mathbb{N}$ such that T is k-valued. Let $k \in \mathbb{N}$ be fixed. It is decidable in* PTime *whether an* NFT *is k-valued.*

Theorem 3.1.30 (Equivalence of k-valued)**.** *[IK86, Web88] The equivalence and inclusion problems for k-valued* NFT *are* PSpace-c.

3.2 Pushdown Machines

Pushdown automata are finite state automata equipped with a stack, they characterize the class of context-free languages. We first recall that the class of context-free languages does not enjoy as good properties as regular languages do. They are not closed under various operations and some important decision problems are undecidable.

Pushdown transducers are pushdown automata with an output mechanism. We show that the weaknesses of pushdown automata carry over to pushdown transducers. Closure properties are scarce and nearly all decision problems are undecidable, with the notable exception of the emptiness and membership problems.

All results can be found in standard textbooks such as [Har78, ABB97, Ber09, HMU03].

3.2.1 Context-Free Grammars

We recall the basic notions of context-free grammar, derivation, associated language, normal forms, and some basic properties. Notably, we recall that, while constructions in standard textbooks are often exponential, pretty much everything (with regards to normal forms) can be done in *polynomial* time.

Definition 3.2.1 (CFG)**.** *A* context-free grammar *is a tuple* (V, T, P, S) *where* V *is the set of* variables*,* T *is the set of* terminals*,* $S \in V$ *is the start symbol, and* P *is the set of* productions*. Productions, also called rules, are of the form* $A \to w$ *where* $A \in V$ *and* $w \in (T \cup V)^*$*.*

The derivation of G, denoted by $\Rightarrow$, is the binary relation defined as follows. For all $\alpha, \beta \in (T \cup V)^*$, $A \in V$ and $A \to \gamma$ a production of G, we have $\alpha A \beta \Rightarrow \alpha\gamma\beta$. The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. The language of G is the set $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. The class of languages defined by CFG is called the class of context-free languages. The size of a rule $A \to w$ is $|w| + 1$, the size $|G|$ of G is the sum of the size of its rules.

Example 3.2.2. *The grammar* $G_{ab}(\{S\}, \{a, b\}, S, P)$ *where* $P = \{S \to aSb, S \to \epsilon\}$ *defines the language* $L(G_{ab}) = L_{ab} = \{a^n b^n \mid n \geq 0\}$*.*

Another interesting example shows that a grammar can be used to represent a word in a compact way.

Example 3.2.3. *The grammar $G_8 = (\{S, A_1, A_2, A_3\}, \{a\}, S, P)$ where $P = \{S \to A_1A_1, A_1 \to A_2A_2, A_2 \to A_3A_3, A_3 \to a\}$ defines the language containing the single word a^8.*

More generally, the grammar $G_{2^k} = (\{S, A_1, \ldots, A_k\}, \{a\}, S, P)$ where $P = \{A_{i-1} \to A_iA_i \mid 1 < i \leq k\} \cup \{S \to A_1A_1, A_k \to a\}$, generates the language containing the single word a^k. Note that, for $n = 2^k$, the size of $|G_n| = O(\log n)$.

We recall the Chomsky, Greibach, and quadratic Greibach normal forms. Note that the definitions might slightly differ from one book to another. Some books rule out the empty word from being in the language. We follow the definitions of [Har78, ABB97].

Definition 3.2.4 (Chomsky Normal Form). *A context-free grammar $G = (V, T, P, S)$ is in* Chomsky normal form *if all production rules are in one of the following form: (i) $A \to BC$, (ii) $A \to a$, or (iii) $S \to \epsilon$, where $A, B, C \in V$, $B, C \neq S$ and $a \in T$.*

The Greibach normal form requires that each production, but the empty one, produces a terminal at its leftmost position. The quadratic Greibach normal form imposes an additional constraint on the length of the right-hand side.

Definition 3.2.5 (Greibach Normal Forms). *A context-free grammar $G = (V, T, P, S)$ is in* Greibach normal form *if all production rules are in one of the following form: (i) $A \to a\alpha$, or $S \to \epsilon$ where $A \in V$, $\alpha \in V^*$, and $a \in T$.*

It is in quadratic Greibach normal form, also called 2-standard form, if it is in Greibach normal form and for any production $A \to a\alpha$ we have $|\alpha| \leq 2$.

Each one of these normal forms captures exactly the context-free languages. Furthermore, one can transform a grammar back and forth (between any two of these forms) in *polynomial* time.

Theorem 3.2.6. *[Har78, ABB97, HMU03] Given a* CFG *one can construct in polynomial time an equivalent grammar in Chomsky, resp. Greibach, resp. quadratic Greibach normal form.*

3.2.2 Pushdown Automata

A pushdown automaton (PA) is a finite state automaton equipped with a stack. The automaton chooses its next transition depending on its current state, the input letter and the symbol on top of the stack. In a transition, the automaton reads the input letter, replaces the symbol on top of the stack by a sequence of symbols from the stack alphabet, and changes its current state.

In its most general form, the automaton can be non-deterministic, that is for a given input letter, a stack symbol and a state, several transition rules may apply. Moreover, epsilon transitions are allowed, *i.e.* transition that do not read nor consume the input letter, but may change the current state and modify the stack.

The automaton starts its computations in one of the predefined initial states and its stack contains a single symbol called the bottom of the stack symbol, denoted by Z_0. A computation is a sequence of compatible transitions. In this document PA accepts a computation if it terminates in one of the predefined *final states*. It is well-known that the class of languages accepted by non-deterministic PA is the same if the automata accept by empty stack or by empty stack *and* final states.

We follow the convention of the previous section and define pushdown automata with, resp. without, ϵ-transitions, and then the deterministic variant.

Definition 3.2.7 (ϵ-NPA). *A non-deterministic pushdown automaton with ϵ-transition (ϵ-NPA) on the alphabet Σ is a tuple $A = (Q, I, F, \Gamma, \delta)$ where*

- *Q is the set of states,*
- *$I \subseteq Q$ is the set of initial states,*
- *$F \subseteq Q$ is the set of final states,*
- *Γ is the stack alphabet such that $Z_0 \in \Gamma$,*
- *$\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \Gamma^* \times Q$ is the transition relation.*

A *configuration* of A is an element of $Q \times \Gamma^*$. A *run* of A on a word $u \in \Sigma^*$ from configuration (q_0, σ_0) to (q_n, σ_n) is a sequence of transitions $\rho = t_1 \dots t_n \in \delta^*$ such that there exists a sequence of configurations $(q_0, \sigma_0) \dots (q_n, \sigma_n)$ and for all $i \in \{1, \dots n\}$, there exist $\sigma'_i, \sigma''_i \in \Gamma^*$, with $t_i = (q_{i-1}, \alpha_i, \gamma, \sigma'_i, q_i)$, $\sigma_{i-1} = \gamma\sigma''_i$ and $\sigma_i = \sigma'_i\sigma''_i$. We write $A \models (q_0, \sigma_0) \xrightarrow{u} (q_n, \sigma_n)$ when there is such a run, or simply $q_0 \xrightarrow{u} q_n$ when A and σ_0, σ_n are clear from the context. A run is *accepting* if $q_0 \in I$, $\sigma_0 = Z_0$ and $q_n \in F$. A word u is *accepted* by A if there exists an accepting run of A on u. The *language* $L(A)$ of A is the set of accepted words: $L(A) = \{u | (q_0, Z_0) \xrightarrow{u} q_n \wedge q_0 \in I \wedge q_n \in F\}$. The set of runs, resp. accepting runs, of A over u is denoted by $\rho(A, u)$, $\rho^{acc}(A, u)$, the set of all runs, resp. accepting runs, of A is denoted by $\rho(A)$, $\rho^{acc}(A)$.

Proposition 3.2.8. *Let A be a ϵ-*NPA*, the sets $\rho(A)$ and $\rho^{acc}(A)$ are context-free languages, and one can construct in* PTime *ϵ-*NPA *that recongnize them.*

The class of pushdown automata exactly characterizes the class of context-free languages. However one has to be careful in order to obtain a *polynomial* time equivalence between PA and CFG. Indeed, the standard construction that transforms a PA into a CFG produces, in some cases, an exponentially larger CFG. However a small modification of this procedure yields a PTime algorithm [HMU03].

Proposition 3.2.9. *Given a ϵ-*NPA*, resp.* CFG*, one can construct in* PTime *an equivalent* CFG*, resp. ϵ-*NPA*.*

As for NFA, we define the real-time and the deterministic pushdown automata.

Definition 3.2.10 (NPA, DPA)**.** *An ϵ-*NPA *$A = (Q, I, F, \Gamma, \delta)$ is a realtime non-deterministic pushdown automaton, or simply a* realtime pushdown automaton *(*NPA*) if $\delta \subseteq Q \times \Sigma \times \Gamma \times \Gamma^* \times Q$.*

It is deterministic *(*DPA*), if $|I| = 1$ and when $(p, a, \gamma, \sigma, q) \in \delta$ and $(p, a, \gamma, \sigma', q') \in \delta$ then $q = q'$ and $\sigma = \sigma'$.*

The class of realtime pushdown automata, also characterizes the context-free languages. Note, however, that in order to accept all context-free languages, an NPA must accept by final state. Otherwise, when accepting by empty stack one has to at least remove the Z_0 symbol and therefore, in that case with no ϵ-transition an NPA can not recognize the empty word.

There is a PTime procedure that removes ϵ-transitions from ϵ-NPA. One way to get rid of the ϵ-transitions is to, first, transform the ϵ-NPA into a CFG, then transform this CFG into a CFG in Greibach normal form and finally, transform back this grammar into an NPA which is by construction, in that case, a realtime automaton.

Proposition 3.2.11 (ϵ-removal)**.** *For any ϵ-*NPA *one can effectively construct in* PTime *an equivalent* NPA*.*

The class of languages defined by DPA does not contain all context-free languages. For example no DPA accept the context-free language $L_r = \{ww^r | w \in \Sigma^*\}$. Indeed, an NPA that recognize L_r has to guess when it reaches the end of w but it cannot guarantee the correctness of its guess until reaching the end of the whole word. This can clearly not be defined by a DPA. We denote by dCFL the class of deterministic CFL, the languages that can be recognized by DPA.

Closure Properties

The class of context-free languages enjoys some reasonable closure properties.

Proposition 3.2.12 (Closure). *The class of context-free languages is closed under union, concatenation, star operation, mirror image, morphism and intersection with regular languages. Moreover, when the* CFL *are represented by* ϵ-NPA*, resp.* NPA*, (and the regular language by a* NFA*) one can construct in* PTime *an* ϵ-NPA*, resp.* NPA*, that represents the* CFL *obtained by application of the given operator.*

The class of deterministic CFL *is closed under complement and intersection with regular languages. When the* dCFL *are given as* DPA *closure are effective in* PTime*.*

Closure properties are useful to build from simple languages more complex ones. In the next example we use closure under concatenation to construct, for a given natural number n, an NPA that accepts a unique word of length n. The size of this NPA is polynomial in $\log n$. This construction is used later on for proving a hardness result.

Example 3.2.13 (Compact NPA). *Let $n \in \mathbb{N}$ a natural number. One can construct an* NPA *A_n whose size is polynomial in $\log n$ and such that $L(A) = \{a^n\}$.*

If n is a power of 2 then G_n from Example 3.2.3 is a CFG *whose size is proportional to $\log n$, by Proposition 3.2.9, there exists an equivalent* NPA *whose size is polynomial in the size of G_n.*

Let $n = \sum_{i=0}^{k} 2^{b_i}$ (the binary decomposition of n) with $k \leq \log n$, $b_0, \dots, b_k \in \mathbb{N}$, and $b_i \neq b_j$ for all $i \neq j$. Consider the grammar $G_n = G_{n_0} \cdot G_{n_1} \cdots G_{n_k}$ where $n_i = 2^{b_i}$. By Proposition 3.2.12, one can construct G_n in PTime *with regards to the sum of the size of the grammars G_{n_i}, that is in* PTime *with regards to $\log n$. Finally one can transform G_n in* PTime *into an* NPA *A_n.*

Context-free languages are not closed under intersection, and, contrarily to the class of deterministic CFL, it is neither closed under complement. Deterministic CFL are not closed under union as they are closed under complement but not under intersection.

Proposition 3.2.14 (Non-closure). *The class of context-free languages and deterministic* CFL *are not closed under intersection, The class of context-free languages is not closed under complement. The class of deterministic* CFL *is not closed under union, Kleene star, concatenation, morphism and mirror image.*

Decision Problems

The membership and emptiness problems are decidable for ϵ-NPA, and so also for NPA and DPA. The class of DPA enjoys the privilege to be one of the rare classes of languages with an undecidable inclusion problem but a decidable equivalence one.

Proposition 3.2.15. *The membership and emptiness problems for* ϵ-NPA*,* NPA *and* DPA *are decidable in* PTime*. The equivalence of* DPA *is decidable [Sén97].*

Many problems are undecidability for NPA. The regularity problem is one of these undecidable problems, it asks whether a context-free language is regular.

The undecidability of the universality problem for NPA easily implies the undecidability of several other problems. We now present this simple proof.

Proposition 3.2.16. *The disjointness, universality, inclusion and regularity problems are undecidable for* ϵ-NPA*,* NPA *and* DPA*. The equivalence problem for* ϵ-NPA *and* NPA *is undecidable. It is undecidable whether an* ϵ-NPA *or a* NPA *is equivalent to some* DPA*.*

Proof. Let $P = ((u_1, v_1) \dots (u_n, v_n))$ be an instance of PCP. Let $A \in \mathsf{PA}$ be a universal automaton ($L(A) = \Sigma^*$). Let $L_1 = \{u_{i_1} \dots u_{i_k} \# i_k i_{k-1} \dots i_1 \mid 0 < i_j \leq n\}$, and $L_2 = \{v_{i_1} \dots v_{i_k} \# i_k i_{k-1} \dots i_1 \mid 0 < i_j \leq n\}$, these languages are context-free. Let $A_1 \in \mathsf{DPA}$ and $A_2 \in \mathsf{DPA}$ such that $L(A_1) = L_1$ and $L(A_2) = L_2$.

The automaton $B = \overline{A_1} \cup \overline{A_2}$ recognizes any word except words that are in $L(A_1) \cap L(A_2)$, i.e. words that are solution of P. B is a non-deterministic pushdown automaton. P has no solution iff B is equivalent to A iff B is universal iff $B \subseteq A$ iff B is regular iff B is a dCFL. ☐

Simple Pushdown Automata

In this document, we call *simple pushdown automaton* a realtime pushdown automaton whose stack behavior is restricted in the following manner. At each transition, the automaton either pushes one *single* symbol on the stack, or it pops (and reads) the symbol on top of the stack, or it does not touch nor read the stack.

Definition 3.2.17. *A* simple pushdown automata *is a pushdown automata* $A = (Q, I, F, \Gamma, \delta)$ *such that for all* $(q, a, \gamma, \sigma, q') \in \delta$ *we either have:*

- *pop:* $\sigma = \epsilon$

- *push:* $\sigma = \gamma\gamma''$ *and for all* $\gamma' \in \Gamma$ *we have* $(q, a, \gamma', \gamma'\gamma'', q') \in \delta$
- *internal:* $\sigma = \gamma$ *and for all* $\gamma' \in \Gamma$ *we have* $(q, a, \gamma', \gamma', q') \in \delta$

The 'for all' conditions on push and internal operations ensure the stack is not read in this transition, that is, the transition does not depend on the symbol on the top of the stack.

Simple pushdown automata captures all context-free languages. The simplest proof of this fact can be found in [BG06]. Let G be a context-free grammar, wlog one can suppose that G is in quadratic Greibach normal form. The application of the classical CFG to NPA transformation produces a real-time pushdown automaton. This automaton (due to the fact that G is in quadratic Greibach normal form) is such that, at each transition, it pops the top of the stack and pushes at most 2 symbols. It is then easy to construct a simple pushdown automaton by storing the current top of the stack symbol in the current state.

Proposition 3.2.18. *[BG06] Given an* NPA*, one can effectively construct in* PTime *an equivalent simple pushdown automaton.*

Parikh Image

The Parikh image of a word u over an ordered alphabet $\Sigma = \{a_1, \ldots, a_n\}$ is a vector of natural number, $\mathsf{parikh}(u) = (i_1, \ldots i_n)$, representing the number of occurrences in u of each letter of Σ, *i.e.* i_j is the number of occurrences in u of the letter a_j. The Parikh image of a language $L \subseteq \Sigma^*$ is the set containing the Parikh images of the words of the language, $\mathsf{parikh}(L) = \{\mathsf{parikh}(u) \mid u \in L\} \subseteq \mathbb{N}^n$.

Parikh images are a useful tool for deciding some properties on languages. For example, a language is empty if and only if its Parikh image of is empty, it is in same cases easier to test the latter condition than the former.

The Parikh image of a context-free language is a semi-linear set. Semi-linear sets can be represented by existential Presburger formulae. Let us recall what are those formulae.

Definition 3.2.19. Existential Presburger formulae *are defined by the following grammar:*

$$t ::= 0 \mid 1 \mid x \mid t_1 + t_2 \quad \phi ::= t_1 = t_2 \mid t_1 > t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x \cdot \phi_1$$

Proposition 3.2.20 ([Pap81])**.** *Satisfiability of existential Presburger formulae is NP-complete.*

A recent result shows that one can construct in linear time an existential Presburger formula that represents the Parikh image of a given context-free grammar [VSS05]. Therefore, by Proposition 3.2.9, one can construct in polynomial time this Parikh image if the context-free language is given as an ϵ-NPA.

Proposition 3.2.21. *[VSS05] Given an ϵ-*NPA *A, one can compute an existential Presburger formula for the Parikh image of $L(A)$ in* PTime.

3.2.3 Pushdown Transducers

We now turn to pushdown transducers and show that the undecidability results of pushdown automata induce undecidability of most problems for the corresponding class of transducers. As for finite state transducers, we look into the three classes induced by the three classes of automata: deterministic, realtime and non-deterministic with ϵ-transitions.

Definition 3.2.22 (Pushdown Transducers)**.** *A* non-deterministic *(ϵ-*NPT*), resp.* realtime *(*NPT*), resp.* deterministic *(*DPT*), pushdown transducer from Σ to Δ is a pair $T = (A, \Omega)$ where $A = (Q, I, F, \Gamma, \delta)$ is an ϵ-*NPA*, resp.* NPA*, resp.* DPA*, and Ω is the output morphism from δ to Δ^*.*

Let $t \in \delta$, $\Omega(t) \in \Delta^*$ is the output of the transition t. Let $u \in \Sigma^*$ and $\rho = t_1 \dots t_n \in \rho^{acc}(A, u)$, we say $v = \Omega(\rho) = \Omega(t_1) \dots \Omega(t_n) \in \Delta^*$ is the output of the run and that v is an image of u. We write $T \models (q_0, \sigma) \xrightarrow{u/v} (q_n, \sigma')$ (or simply $(q_0, \sigma) \xrightarrow{u/v} (q_n, \sigma')$) if there is a run $\rho \in \rho(A, u)$ from (q_0, σ) to (q_n, σ') with $v = \Omega(\rho)$. The transduction defined by T is the relation $R(T) = \{(u, v) \mid \exists (q_0, Z_0) \xrightarrow{u/v} (q_n, \sigma) \land q_0 \in I \land q_n \in F\}$.

The domain of T is the language of its underlying automaton, i.e. it is a context-free language. The range of T is the image by Ω of the language (over the alphabet δ) of the accepting runs of A, $\rho^{acc}(A)$. This language is a context-free language (Proposition 3.2.8). As CFL are closed under morphism (Proposition 3.2.12), the range of an ϵ-NPT is a context-free language.

Proposition 3.2.23. *The domain and range of any ϵ-*NPT *are context-free languages and one can compute in* PTime *an ϵ-*NPA *that represents it.*

Closure Properties

There are few operations under which the various classes of pushdown transducers are closed.

Union. Non-determinism induces closure under union for the classes ϵ-NPT and NPT, while the class of DPT is not closed under union as the result might not be functional.

Inverse. The class of ϵ-NPT is closed under inverse. Indeed, without loss of generality we can suppose that the output of any transition of an ϵ-NPT, $T = (A, \Omega)$ is either the empty word or a single letter, that is, $\Omega : \delta \to \Delta \cup \{\epsilon\}$. Therefore, to get the inverse, we just swap the output with the input.

The classes of NPT and DPT are not closed under inverse, as any word has only a finite number of images.

Proposition 3.2.24. *The class of ϵ-NPT is closed under inverse. The class of ϵ-NPT and NPT are closed under union.*

The pushdown transducers are not closed under any of the other operations we considered for finite state transducers.

Composition. To show non-closure under composition, consider the following deterministic pushdown transductions:

$$R_1 = \{(a^n b^n c^m, a^n b^n c^m) \mid n, m \geq 0\}$$

$$R_2 = \{(a^n b^m c^m, a^n b^m c^m) \mid n, m \geq 0\}$$

we have:

$$R_2 \circ R_1 = \{(a^n b^n c^n, a^n b^n c^n) \mid n \geq 0\}$$

which is not a pushdown transductions as neither its domain nor its range are context-free.

Intersection. Non closure under intersection can be deduced directly from the fact that context-free languages are not closed under intersection and that the domains of pushdown transducers are context-free languages.

Complement. For ϵ-NPT and NPT, non closure under complement is a consequence of the fact that their domains are context-free languages which are not closed under complement. While for DPT, we can invoke the same argument as for inverse: each word has only a finite number of images.

We summarize closure properties for pushdown transducers in Table 3.3.

	T^{-1}	$\overline{T}$	$T_1 \cup T_2$	$T_1 \cap T_2$	$T_1 \circ T_2$
ϵ-NPT	yes	no	yes	no	no
NPT	no	no	yes	no	no
DPT	no	no	no	no	no

Table 3.3: Closure Properties for ϵ-NPT, NPT and DPT.

Decision Problems

All problems we consider but membership and emptiness are undecidable for pushdown transducers. The undecidability results for pushdown transducers can be easily derived from undecidability of related problems for pushdown automata. We start with the decidable problems and then turn to undecidability results.

Emptiness, membership. A pushdown transducer is empty if its domain is empty. Therefore deciding emptiness reduces to deciding emptiness of context-free languages.

Deciding membership, that is, given a pushdown transducer T and a pair of words u, v, whether $v \in T(u)$ can be done by computing (in PTIME) a pushdown automaton representing the image of u by T (product of T with u) and testing the membership of v in $T(u)$, which can be done in PTIME according to Proposition 3.2.15.

Proposition 3.2.25. *The emptiness and membership problems for ϵ-*NPT*, and thus for* NPT *and* DPT*, are decidable in* PTIME.

Type checking. Recall that the type checking problem asks given an input language L_{in}, an output language L_{out}, and a transducers T whether $T(L_{in}) \subseteq L_{out}$ holds. The identity relation ($T_{id} = \{(u, u) \mid u \in \Sigma^*\}$) can be defined by a DPT, therefore the undecidability of the type checking problem for DPT against DPA is a direct consequence of the undecidability of the inclusion problem for DPA. Clearly, it is also undecidable for ϵ-NPT and NPT. Note that this argument also holds for the type checking of DFT against DPA.

However, when the input and output languages are regular languages given as finite state automata, the type checking problem is decidable. Indeed, given an input and an output language L_{in} and L_{out} and a pushdown transducer T, then, T type checks against L_{in} and L_{out} if and only if the image of L_{in} by T is included into L_{out}. The image of L_{in} by T is a context-free language and,

	emptiness membership	type checking (vs NFA)	equivalence / inclusion	universality
ϵ-NPT	PTime	undec	undec	undec
NPT	PTime	undec	undec	-
DPT	PTime	undec	dec[Sén99] / undec	-

Table 3.4: Decision problems for ϵ-NPT, NPT and DPT.

by Proposition 3.2.23, one can compute in PTime a pushdown automata that represents it. Finally, the inclusion of a pushdown automata in a finite state automata can be decided in ExpTime (Proposition 3.1.10).

Proposition 3.2.26 (Type Checking)**.** *The type checking problems for* DPT*,* ϵ-NPT *and* NPT*, against* DPA*, resp.* ϵ-NPA *or* NPA*, are undecidable.*

Equivalence, inclusion, universality. As a consequence of the undecidability of the inclusion problem for DPA, the inclusion problem is undecidable for DPT, and therefore it is also undecidable for ϵ-NPT and NPT.

The equivalence problem is undecidable for ϵ-NPT and NPT as this is already the case for NFT. On the contrary, the equivalence problem for DPT has been shown to be decidable by Senizergues[Sén99].

Finally, the universality problem is undecidable for ϵ-NPT as it is for ϵ-NFT, and is always false for NPT and DPT as, in these cases, a word has only a finite number of images.

Proposition 3.2.27 (Equivalence, inclusion, universality)**.** *The inclusion problem for* DPT*, and thus also for* ϵ-NPT *and* NPT*, is undecidable. The equivalence problem for* ϵ-NPT *and* NPT*, is undecidable, but decidable for* DPT*. The universality problem is undecidable for* ϵ-NPT *and trivially false for* NPT *and* DPT*.*

The results on decision problems for ϵ-NPT, NPT an DPT are summarized in Table 3.4.

Functional and k-valued transductions

In the case of finite state transducers, restricting functional, or more generally k-valued, transductions yields decidability of the equivalence and inclusion problems. We recall here that functionality is undecidable for pushdown transducers and therefore so is k-valuedness. Moreover we recall that the inclusion and equivalence problem are undecidable even when we restrict to functional pushdown transducers.

Proposition 3.2.28. *The k-valuedness problem for* NPT *is undecidable, and in particular so is the functionality problem.*

Proof. Let $P = ((u_1, v_1), \dots (u_n, v_n))$ an instance of PCP. The following relations are two pushdown transductions:

$$R_1 = \{(u_{i_1} \dots u_{i_k} \# i_k i_{k-1} \dots i_1, 0) \mid 0 < i_j \leq n\}$$

$$R_2 = \{(v_{i_1} \dots v_{i_k} \# i_k i_{k-1} \dots i_1, 1) \mid 0 < i_j \leq n\}$$

Therefore $R = R_1 \cup R_2$ is a pushdown transduction. R is functional if and only if P has no solution. □

As a consequence of the undecidability of the equivalence and inclusion problems for pushdown automata, it is also undecidable whether two functional pushdown transducers are equivalent, resp. included.

Proposition 3.2.29 (Equivalence and inclusion)**.** *The equivalence and inclusion problems are undecidable for* functional *pushdown transducers.*

While the equivalence problem for *deterministic* pushdown transducers is decidable, it is undecidable whether an NPT is determinizable, that is, whether there exists an equivalent DPT. This is a direct consequence of the undecidability of deciding whether a NPA is equivalent to some DPA (Proposition 3.2.16).

Proposition 3.2.30 (Determinization problem)**.** *It is undecidable whether a given* NPT *is equivalent to some* DPT*.*

The same observation holds for NFT. It is undecidable to check whether a NPT is equivalent to some NFT, as this is already the case for the corresponding automata (Proposition 3.2.16).

Proposition 3.2.31 (Regularity problem)**.** *It is undecidable whether a given* NPT *is equivalent to some* NFT*.*

	$\overline{L}$	$L_1 \cup L_2$	$L_1 \cap L_2$	$h(L)$	L^r	L_1L_2	$L \cap \mathsf{REG}$
REG	yes	yes	yes	yes	yes	yes	yes
CFL	no	yes	no	yes	yes	yes	yes
dCFL	yes	no	no	no	no	no	yes

Table 3.5: Closure under complement, union, intersection, morphism, mirror image, concatenation, and intersection with a regular language for the classes REG, CFL and dCFL.

	emptiness / membership	equivalence / inclusion	universality
ϵ-NFA	PTime	PSpace-c	PSpace-c
NFA	PTime	PSpace-c	PSpace-c
DFA	PTime	PTime	PTime
ϵ-NPA	PTime	undec	undec
NPA	PTime	undec	undec
DPA	PTime	dec/undec	dec

Table 3.6: Decision problems for ϵ-NFA, NFA, DFA, ϵ-NPA, NPA and DPA.

3.3 Conclusion

We summarized in Table 3.5 the closure properties and in Table 3.6 decision problems for the languages and automata introduced in this chapter. The difference between finite state and pushdown automata is considerable. Adding a stack to a class of finite state automata makes it loose the Boolean closure and makes the equivalence (except for DPA), inclusion and universality problems undecidable.

The results for the closure properties and the decision problems for transducers are summarized in Table 3.7 and Table 3.8. The main differences between finite state and pushdown transducers lie in the closure under composition, the decidability of functionality (and more generally k-valuedness), the decidability of the equivalence of functional transducers and the decidability of determinizability.

Clearly, the class of functional (and more generally k-valued) finite state transductions has the most interesting properties: it is closed under composition and the equivalence and inclusion problems are decidable. Moreover, unambigu-

	T^{-1}	$\overline{T}$	$T_1 \cup T_2$	$T_1 \cap T_2$	$T_1 \circ T_2$
ϵ-NFT	yes	no	yes	no	yes
NFT	no	no	yes	no	yes
DFT	no	no	no	no	yes
ϵ-NPT	yes	no	yes	no	no
NPT	no	no	yes	no	no
DPT	no	no	no	no	no

Table 3.7: Closure under inverse, complement, union, intersection, and composition for ϵ-NFT, NFT, DFT, ϵ-NPT, NPT and DPT.

	emptiness / membership	equivalence / inclusion	universality	functionality/ determinizable
ϵ-NFT	PTIME	undec	undec	PTIME/ PTIME
NFT	PTIME	undec	-	PTIME/ PTIME
functional NFT	PTIME	PSPACE-C	-	- / PTIME
DFT	PTIME	PTIME	-	- / -
ϵ-NPT	PTIME	undec	undec	undec / undec
NPT	PTIME	undec	-	undec / undec
functional NPT	PTIME	undec	-	- / undec
DPT	PTIME	dec/undec	-	- / -

Table 3.8: Decision problems for ϵ-NFT, NFT, DFT, ϵ-NPT, NPT and DPT.

ous transducers are all functional, and any functional, resp. k-valued, finite state transduction can be defined by an unambiguous, resp. k-ambiguous, finite state transducer. Finally, deterministic finite state transductions are also functional and it is decidable whether a functional NFT is determinizable.

As a conclusion, note that many undecidability results for pushdown transducers are a direct consequence of undecidability results for pushdown automata. However, some fundamental undecidability results already appear for finite state transductions: the equivalence and inclusion problems are undecidable. To avoid this undecidability, one has to restrict to functional or k-valued transductions.

Chapter 4

Visibly Pushdown Automata

Contents

Introduced by R. Alur and P. Madhusudan [AM04], visibly pushdown automata (VPA) form an interesting subclass of pushdown automata. They characterize the class of visibly pushdown languages (VPLs). The VPLs form a strict subclass of context-free languages but strictly extends regular languages. This class of languages is attractive because, unlike the class of (deterministic) context-free languages, it enjoys good closure properties. Moreover, all the relevant decision problems are decidable for VPAs.

We call *structured alphabet* an alphabet that is partitioned into three sets, the call symbols, the return symbols and the internal symbols. A VPA is a pushdown automata that operates on a structured alphabet. Its behavior is constrained by the type of the input letter. When reading a call the automaton must push a single symbol on its stack. When reading a return it must pop the symbol on top of its stack and it can use this symbol for choosing the transition. Finally, when reading an internal the automaton cannot touch nor read its stack.

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 time. This enables the important product construction and yields closure under intersection.

Furthermore, for any run, the stack symbol popped when reading a return is the symbol that was pushed when reading its matching call, which is the same for all runs. By taking advantage of this observation it is possible, unlike pushdown automata, to determinize VPA. The determinization yields closure under complement.

Closure under union is obtained thanks to non-determinism. Therefore VPLs are closed under all Boolean operations. This coupled with a decidable emptiness problem provides decision procedures for all others relevant decision problems. Testing inclusion is obtained by testing the emptiness of the intersection between the first language and the complement of the second. Testing equivalence can be done by testing mutual inclusion. Lastly, testing universality can be done by testing emptiness of the complement.

We now formally define VPA, then we recall all closure properties and the main decision procedures. We also show that visibly pushdown automata can be reduced, that is, can be transformed into an equivalent VPA such that all accessible configurations are co-accessible, in polynomial time. 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 [AM09].

4.1 Definitions

Visibly pushdown automata are pushdown automata and, as such, they do not need a new definition. However the restriction imposed to the use of the stack is better enforced with a slightly modified definition. Moreover, constructions like product and determinization are better explained with a streamlined, ad-hoc definition. We first recall the notion of structured alphabet.

4.1.1 Structured alphabet and nesting

A structured alphabet, Σ, is a triple $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$, where Σ_c, Σ_i, and Σ_r, are disjoint alphabet 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 $u = \epsilon$,

(*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*) $u = vw$ where v and w 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 $L_{wn}(\Sigma)$, or simply L_{wn} when Σ is clear from the context.

Let $u \in L_{wn}(\Sigma)$, the *height*, or *nesting level*, of u, denoted by $h(u)$, is defined inductively as follows:

(*i*) $h(\epsilon) = 0$,

(*ii*) $h(a) = 0$ with $a \in \Sigma_i$,

(*iii*) $h(cvr) = h(v) + 1$

(*iv*) $h(vw) = max(h(v), h(w))$

A word u over the structured alphabet Σ can be uniquely decomposed as $u = w_0 r_1 w_1 r_2 \dots r_m w_m c_1 w_{m+1} c_2 w_{m+2} \dots c_n w_{m+n}$, where, for all $0 \leq i \leq m + n$, $w_i \in L_{wn}(\Sigma)$, for all $1 \leq j \leq m$, $r_j \in \Sigma_r$ are the *unmatched returns* and for all $1 \leq k \leq n$, $c_k \in \Sigma_c$ are the *unmatched calls.*

Example 4.1.1. *The word $u = c_0 c_1 a a r_1 c_2 a r_2 r_0 c_3 r_3$ is well-nested. For all $0 \leq i \leq 3$, the matching return of the call c_i is the return r_i. The height $h(u)$ of u is 2.*

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.

4.1.2 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 corresponds 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 on its stack (and cannot read the stack). A call transition is defined by a tuple $(q, c, \gamma, q') \in Q \times \Sigma_c \times \Gamma \times Q$, where q is the current state, c the input symbol (which must be a call symbol), γ is the stack symbol pushed on the stack, and q' is the next state.

A *return transition* occurs when the automaton reads a call 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 $\perp$. A return transition is defined by a tuple $(q, r, \gamma, q') \in Q \times \Sigma_r \times (\Gamma \cup \{\perp\}) \times Q$, where q is the current state, r the input symbol (which must be a return symbol), γ is the stack symbol on top of the stack, and q' is the next state.

An *internal transition* occurs when the automaton reads an internal symbol. In that case it cannot touch nor read its stack. An internal transition is defined by a tuple $(q, i, q') \in Q \times \Sigma_i \times Q$, where q is the current state, i the input symbol, and q' is the next state.

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 4.1.2 (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 the current state and $\sigma \in \perp \cdot \Gamma^*$ is the current 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 transition $\rho = t_1 t_2 \dots t_l \in \delta^*$ (where δ^* denotes the set of finite sequences of elements of δ) 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 $\sigma_{k-1} = \sigma_k$.

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 is *accepting* if $q_0 \in I$, $\sigma_0 = \perp$, and $q_l \in F$. A word w is *accepted* by A if there exists an accepting run of A over w. $L(A)$, the *language* of A, is the set of words accepted by A. A language L over Σ is a *visibly pushdown language* if there is a VPA A over Σ such that $L(A) = L$. The set of runs, resp. accepting runs, of A over $u \in \Sigma^*$ is denoted by $\rho(A, u)$, resp. $\rho^{acc}(A, u)$. The set of runs, resp. accepting runs, of A is denoted by $\rho(A)$, resp. $\rho^{acc}(A)$. Clearly, those sets are visibly pushdown languages.

Proposition 4.1.3. *Let $A = (Q, I, F, \Gamma, \delta)$ be a* VPA*, $\rho(A)$ and $\rho^{acc}(A)$ are visibly pushdown languages over the alphabet δ.*

Definition 4.1.4 (Deterministic VPA)**.** *A* VPA *$A = (Q, I, F, \Gamma, \delta)$ is deterministic if the following conditions hold: (i) $(q, a, \gamma, q'), (q, a, \gamma', q'') \in \delta_c$ 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*.*

4.1.3 Examples

We now present some examples of visibly pushdown languages and automata.

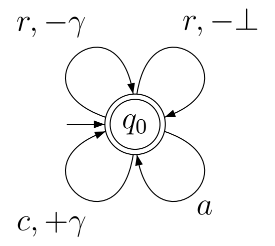


Figure 4.1: VPT A_1 on $\Sigma_c = \{c\}$, $\Sigma_i = \{a\}$, and $\Sigma_r = \{r\}$.

Example 4.1.5. *The deterministic* VPA $A_1 = (Q, I, F, \gamma, \delta)$ *is represented in Figure 4.1, where* $Q = I = F = \{q_0\}$*, and* $\Gamma = \{\gamma\}$*. It recognizes the universal language over* $\Sigma = (\{c\}, \{a\}, \{r\})$*, i.e.* $L(A_1) = \Sigma^*$*.*

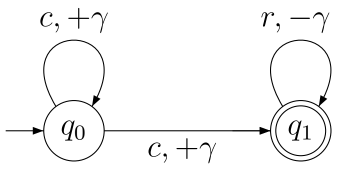


Figure 4.2: VPA A_2 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Example 4.1.6. *The deterministic* VPA $A_2 = (Q, I, F, \gamma, \delta)$ *is represented in Figure 4.2, where* $Q = I = F = \{q_0\}$*, and* $\Gamma = \{\gamma\}$*. It recognizes the non regular language* $\{c^n r^m \mid n \geq m\}$*.*

Example 4.1.7. *The deterministic* VPA $A_3 = (Q, I, F, \gamma, \delta)$ *is represented in Figure 4.3, 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 4.1.8. *The deterministic* VPA $A_A = (Q, I, F, \gamma, \delta)$ *is represented in Figure 4.4, where* $Q = \{q_0, q_1\}$*,* $I = \{q_0\}$*,* $F = \{q_0\}$*, and* $\Gamma = \{\gamma_0, \gamma\}$*. It recognizes the language of well-nested words over* $\Sigma = (\{c\}, \emptyset, \{r\})$*, we have* $L(A_4) = L_{wn}(\Sigma)$*.*

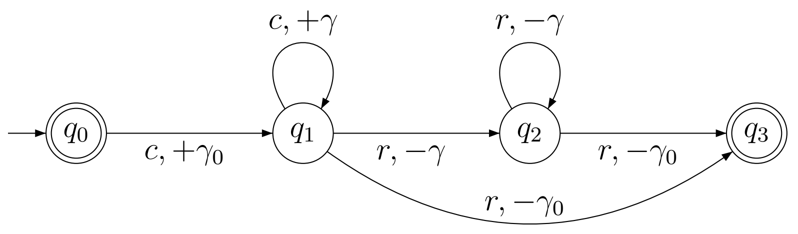


Figure 4.3: VPA A_3 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

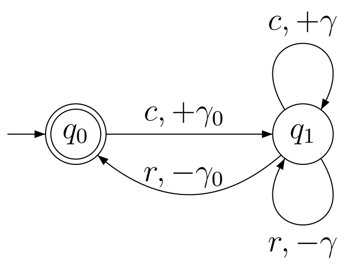


Figure 4.4: VPA A_4 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Example 4.1.9. *Consider the context-free language* $L = \{a^n b^n c^m a^m \mid n, m \geq 0\}$. *This language is not a* VPL *for any partition of* Σ.

4.2 Closure Properties

The closure properties of the class of VPLs are obtained by various constructions on VPA. We recall the three main constructions: union, product and determinization.

Union. The union of A and B, noted $A \cup B$, is obtained using non-determinism. The automaton for the union non-deterministically chooses a run of A or B. It amounts to simply take the disjoint union of the components (set of states, stack alphabet, and transition relation) of the two VPA.

Product. The intersection is obtained via a product construction. 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 popped or pushed together. 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 components (set of states and stack alphabet) of the two VPA, except for the transition relation which is obtained by a synchronized product.

Definition 4.2.1 (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*:*

$$A_1 \times A_2 = ((Q_1 \times Q_2), (I_1 \times I_2), (F_1 \times F_2), (\Gamma_1 \times \gamma_2), (\delta_1 \otimes \delta_2))$$

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: $((q_1, q_2), a, (\gamma_1, \gamma_2), (q_1', q_2')) \in \delta_1 \otimes \delta_2 \quad \textit{iff} \begin{cases} (q_1, a, \gamma_1, q_1') \in \delta_1 \\ (q_2, a, \gamma_2, q_2') \in \delta_2 \end{cases}.$

returns on $\perp$: $((q_1, q_2), a, \perp, (q_1', q_2')) \in \delta_1 \otimes \delta_2 \quad \textit{iff} \begin{cases} (q_1, a, \perp, q_1') \in \delta_1 \\ (q_2, a, \perp, q_2') \in \delta_2 \end{cases}.$

internals: $((q_1, q_2), a, (q_1', q_2')) \in \delta_1 \otimes \delta_2 \quad \textit{iff} \begin{cases} (q_1, a, q_1') \in \delta_1 \\ (q_2, a, q_2') \in \delta_2 \end{cases}.$

Each transition of the product $A_1 \times A_2$ corresponds to a pair formed by a transition of A_1 and a transition of A_2. Furthermore, any run ρ of $A_1 \times A_2$ 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 VPAs is the intersection of the languages accepted by these automata.

Determinization. Unlike pushdown automata, VPA are determinizable. The construction is based on the notion of summaries and the observation that a run of a VPA over a well-nested word starts and finishes with the same stack content. The summary of a run starting in q and ending in q' over a well-nested word w is the pair (q, q'). The determinization maintains sets of summaries, that allows one to simulates all possible runs of the VPA on the input word.

Let $A = (Q, I, F, \Gamma, \delta)$ be a VPA and $B = (Q', I', F', \Gamma', \delta')$ be the VPA obtained by the determinization procedure. Let us explain how B works.

Any word u can be decomposed as $u = w_0 c_1 w_1 c_2 w_2 \dots c_n w_n$, where w_0 is a word with no unmatched call, c_i are the unmatched calls and, for all $i > 0$, w_i is a well-nested words. Let the following run be the unique run of B over u:

$$S_0 \xrightarrow{w_0} S'_0 \xrightarrow{c_1, +(c_1, S'_0)} S_1 \xrightarrow{w_1} S'_1 \xrightarrow{c_2, +(c_2, S'_1)} S_2 \cdots S_n \xrightarrow{w_n} S'_n$$

After reading u, the state $S = S'_n$ of B is the set of summaries (q, q') such that $(q, \perp) \xrightarrow{w_n} (q', \perp)$ and such that there exists $q_0 \in I$ with

$$(q_0, \perp) \xrightarrow{w_0 c_1 w_1 c_2 w_2 \dots c_n} q$$

i.e. q is reachable from an initial state by reading $w_0 c_1 w_1 c_2 w_2 \dots c_n$.

Consider the following run of A over u:

$$q_0 \xrightarrow{w_0} q'_0 \xrightarrow{c_1, \gamma_1} q_1 \xrightarrow{w_1} q'_1 \xrightarrow{c_2, \gamma_2} q_2 \cdots q_n \xrightarrow{w_n} q'_n$$

Then for all $i \leq n$, we have $(q_i, q_i) \in S_i$ and $(q_i, q'_i) \in S'_i$.

After reading u, suppose that B is in state S and reads a call c, then it (i) pushes on the stack its current state S and the call letter c, and (ii) enter the state $S' = \{(q, q) \mid \exists q_0 \in I \wedge A \models (q_0, \perp) \xrightarrow{uc} q\}$. The set S' can easily be computed from S, c and the transition relation of A as follows:

$$S' = \{(q, q) \mid \exists (q', q'') \in S \wedge (q'', c, \gamma, q) \in \delta_c\}$$

Now suppose that B is in state S, reads an r and pops (c_n, S'_{n-1}). The new state of B is the subset of all summaries (q, q') over $w_{n-1} c_n w_n r$ (and such that q is reachable from an initial state). It can be computed easily from S'_{n-1}, c_n, r and S as follows:

$$\{(q, q') \mid \exists (q, q_1) \in S'_{n-1} \wedge (q_1, c_n, \gamma, q_2) \in \delta_c \wedge (q_2, q_3) \in S \wedge (q_3, r, \gamma, q') \in \delta_r\}$$

The internal transitions as well as the return transitions on the empty stack $(\perp)$ are treated similarly.

We now formally define the determinization. The formal proof of the correctness of the construction can be found in [AM09].

Theorem 4.2.2 (Determinization[Tan09, AM09]). *Let A be a* VPA *there exists a* dVPA *A' such that $L(A) = L(A')$ and $|A'| = Q^{2|A|}$.*

Let $A = (Q, I, F, \Gamma, \delta)$ and A' is the tuple $(Q', I', F', \Gamma', \delta')$ such that:

- $Q' = 2^{Q \times Q}$
- $I' = \{(q, q) \mid q \in I\}$
- $F' = \{S \in Q' \mid \exists (q, q') \in S : q' \in F\}$
- $\Gamma' = Q' \times \Sigma_c$
- $\delta' = \delta'_c \uplus \delta'_i \uplus \delta'_r$ *with:*

  Calls: *For all* $S \in Q'$, $c \in \Sigma_c$*:*

  $$B \models S \xrightarrow{c,+(S,c)} S' \quad \textit{iff} \quad S' = \{(q, q) \mid \exists (q', q'') \in S : A \models q'' \xrightarrow{c,+\gamma} q\}$$

  Returns: *For all* $S, S', S'' \in Q', r \in \Sigma_r, c \in \Sigma_c$*:*
  $B \models S \xrightarrow{r,-(S'',c)} S'$ *iff*
  $S' = \{(q, q') \mid \exists (q, q_1) \in S'' \wedge A \models q_1 \xrightarrow{c,+\gamma} q_2 \wedge (q_2, q_3) \in S \wedge A \models q_3 \xrightarrow{r,-\gamma} q'\}$

  Returns on $\perp$: *For all* $S, S' \in Q', r \in \Sigma_r$*:*
  $B \models S \xrightarrow{r,-\perp} S'$ *iff*
  $S' = \{(q, q') \mid \exists (q, q_1) \in S \wedge A \models q_1 \xrightarrow{r-\perp} q'\}$

  Internals: *For all* $S, S' \in Q'$, $a \in \Sigma_i$*:*
  $S \xrightarrow{a} S'$ *iff*
  $S' = \{(q, q') \mid \exists (q, q_1) \in S : q_1 \xrightarrow{a} q'\}$

A VPA that recognizes the complement of the language of a VPA A is obtained by determinization 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.

We just showed that the class of VPL is closed under intersection, union, complementation. The closure under Kleene star and concatenation are obtained with a slight modification of the classic construction for NFA. The modification is required to deal with return transitions on empty stack. Indeed, suppose that the automaton C recognizes the concatenation of the language of the VPA A and B. If the language of A contains a word, say u, with unmatched calls, the stack of A is not empty after reading u. So, C must mark that last pushed symbol as a bottom of the stack symbol before starting the simulation of the second automaton [AM09].

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.

Proposition 4.2.3 (Closure). *The class of* VPL *is closed under union, intersection, complement, Kleene star and concatenation. When the languages are given by* VPAs*, these closure are effective and can be computed in exponential time for the complement and polynomial for the other operators. If a* VPA *is deterministic, its complement can be computed in polynomial time.*

As a consequence of Proposition 4.2.3, Proposition 3.1.9, and the fact that, for any $a \in \Sigma$, $\{a\}$ is a VPL, we have that any regular language on Σ is a VPL (for any partition of Σ into calls, returns and internals).

Note that, this does not hold if the automaton is not allowed to make return transitions on the empty stack ($\perp$), neither if it should accepts 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 $c \in \Sigma_c$.

Given a NFA A one can easily construct an equivalent VPA whose runs are in a one-to-one correspondence with the runs of A.

Proposition 4.2.4. *We have:* REG $\subsetneq$ VPL*. Moreover, let $A \in$* NFA*, one can effectively construct in linear time an equivalent* VPA.

Proof. Let $A = (Q, I, F, \delta)$, then $B = (Q, I, F, \Gamma, \delta')$ with $\Gamma = \{\gamma\}$ and $\delta' = \delta'_c \uplus \delta'_i \uplus \delta'_r$ is defined as follows:

$$\text{For all } (p, a, q) \in \delta \quad \text{then} \begin{cases} (p, a, \gamma, q) \in \delta'_c & \text{if } a \in \Sigma_c \\ (p, a, \gamma, q) \in \delta'_r \quad \text{and} & \\ (p, a, \perp, q) \in \delta'_r & \text{if } a \in \Sigma_r \\ (p, a, q) \in \delta'_i & \text{if } a \in \Sigma_i \end{cases}$$

Obviously we have $L(A) = L(B)$. □

Note that, B is deterministic if and only if A is.

While the class of VPL is a strict subclass of CFL, 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 4.2.5 (Relation with CFL)**.** *We have:* $\mathsf{VPL} \subsetneq \mathsf{CFL}$*. Let* $A \in \mathsf{VPL}$*, one can construct in linear time an equivalent* NPA*. Moreover, let* $L \in \mathsf{CFL}(\Sigma)$ *there exists a* VPL L' *on* $\hat{\Sigma}$ *such that* $\pi(L') = L$*.*

Proof. We consider a pushdown automaton A that accepts L. Without loss of generality we can assume that A is a simple pushdown automaton (See Proposition 3.2.18), *i.e.* it pushes or pops at most one symbol with each transition and it accepts by final states. Intuitively the VPA A' will accept the words that A accepts, annotated with the stack behavior of A.

The VPA A' operates over the structured alphabet $\hat{\Sigma}$. A' behaves as A do but differs on the symbol it reads as follows. If A can read a and pushes a symbol γ, A' can read the corresponding call, $\overline{a}$, and pushes the same symbol γ. If A can read a and pops a symbol γ, A' can read the corresponding return, $\underline{a}$, and pops the same symbol γ. If A can read a without touching (read nor write) the stack, A' can read the internal a.

Clearly, $\pi(L(A')) = L(A)$. □

4.3 Decision Procedures

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 4.3.1 (Emptiness, membership)**.** *The emptiness and membership problem for* VPA *are decidable in* PTIME*.*

The universality, inclusion and equivalence problems are all EXPTIME-C. Moreover, equivalence and inclusion are still EXPTIME-C if we restrict the problem to VPA that accepts only well-nested words.

Theorem 4.3.2 (Universality, inclusion and equivalence)**.** *Universality, inclusion and equivalence of* VPA *are* EXPTIME-*c, and in* PTIME *when the* VPA

are deterministic. The inclusion and equivalence problem for VPA *A such that $L(A) \in L_{wn}$ are* ExpTime*-c.*

Proof. The proof for VPA is done in [AM09]. Let us prove that the result also holds when we restrict to VPA that accept only well-nested words. The upper bound is obtained directly form the upper bound for VPA.

For the lower bound, let A and B be two VPA over Σ, we construct in PTime two VPA A' and B' with $L(A') \subseteq L_{wn}$ and $L(B') \subseteq L_{wn}()$ and such that $L(A) = L(B)$ if and only if $L(A') = L(B')$. We add to the alphabet the call symbol c_0 and the return symbol r_0. The languages $L_{c_0} = \{c_0^n \mid n \geq 0\}$ and $L_{r_0} = \{r_0^n \mid n \geq 0\}$ are regular languages and therefore they are VPL. The language of well-nested word L_{wn} is also a VPL. Therefore the language

$$wn(A) = (L_{c_0} L(A) L_{r_0}) \cap L_{wn} = \{c_0^n u r_0^n \mid u \in L(A) \wedge n \geq 0 \wedge c_0^n u r_0^n \in L_{wn}\}$$

is a VPL, as VPL are closed under concatenation and intersection. One can easily check that A is equivalent to, resp. included into, B if and only if $wn(A)$ is equivalent, resp. included into, $wn(B)$. □

The inclusion of a regular language into a VPL and a VPL into a regular language is obviously decidable. However, inclusion of a regular language into a CFL is undecidable, therefore so is the inclusion of a VPL into a CFL. But, inclusion of a CFL into a regular language is decidable as one can compute the product of the complement of the regular language with the CFL, this gives a CFL that is empty if and only if the inclusion holds. In contrast, we prove in the next proposition that the inclusion of a CFL into a VPL is undecidable.

Proposition 4.3.3 (Inclusion with CFL)**.** *Let $C \in$ CFL and $V \in$ VPL, it is undecidable whether $V \subseteq C$, respectively $C \subseteq V$.*

Proof. The undecidability of $V \subseteq C$ is a direct consequence of the undecidability of the inclusion of regular languages into CFL.

For the other direction, CFL $\subseteq$ VPL, let $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ be an instance of PCP defined on the finite alphabet Δ. We construct a CFL and a VPL language defined on the structured alphabet $\Sigma = (\Sigma_c, \emptyset, \Sigma_r)$ 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 $C \subseteq V$. □

$\subseteq$	NFA	VPA	NPA
NFA	PSpace-c	PSpace-c	undec
VPA	PSpace-c	ExpTime-c	undec
NPA	PSpace-c	undec	undec

Table 4.1: Inclusion between NFA, VPA, and NPA.

The decidability status for the inclusion problems between NFA, VPA, and NPA are summarized in Table 4.1.

4.4 Reduced VPA

A configuration (q, σ) is accessible if there exist $q_0 \in I$ and $u \in \Sigma^*$ with $(q_0, \perp) \xrightarrow{u} (q, \Sigma)$. It is co-accessible if there exist $q_f \in F$, $u \in \Sigma^*$ with $(q, \sigma) \xrightarrow{u} (q_f)$.

Definition 4.4.1. *A* VPA *is* reduced *if all accessible configurations are also co-accessible. It is* trimmed *if it is reduced and all co-accessible configuration are also accessible.*

We now recall how a VPA can be reduced in PTime. The complete proof to show how to trim a VPA (which is a little more difficult than reducing it) can be found in [CRT11]. This is the VPA counterpart of the trimming procedure for NFA. Trimming an NFA amounts to removing the state that are either not accessible or not co-accessible. For VPA, it is a little more involved as the accessibility, resp. co-accessibility, depends not only on the state but on the stack, *i.e.* on the configuration (q, σ). In other words, a configuration (q, σ) might be accessible and co-accessible, but another configuration with the same state (q, σ') might be, for instance, accessible but not co-accessible. Trimming a VPA cannot be achieved by just removing states or transitions.

Let $A = (Q, I, F, \Gamma, \delta)$ be a VPA and suppose that $L(A) \in L_{wn}$. This assumptions implies that A has no return transitions on $\perp$ and that its stack is empty whenever it accepts a word.

We first compute the set

$$W = \{(p, q) \in Q \times Q \mid \exists u \in \Sigma^* : A \models (p, \perp) \xrightarrow{u} (q, \perp)\}$$

It can be computed by a fix point algorithm. Indeed, it is the smallest set W and such that:

(i) if $q \in Q$, then $(q,q) \in W$,

(ii) if $(p,r),(r,q) \in W$, then $(p,q) \in W$,

(iii) if $(p,r) \in W$ and $(r,a,q) \in \delta_i$, then $(p,q) \in W$,

(iv) if $(r,s) \in W$, $(p,c,\gamma,r) \in \delta_c$ and $(s,r',\gamma,q) \in \delta_r$, then $(p,q) \in W$.

There is a one-to-one correspondence between accepting runs of the reduced VPA B and accepting runs of the original VPA A. Let $u \in L_{wn}$, it can be uniquely decomposed as $u = cw_1rw_2$, where $c \in \Sigma_c$, $r \in \Sigma_r$, and $w_1, w_2 \in L_{wn}$. Consider the following run of A over u:

$$p \xrightarrow{c,+\gamma} p' \xrightarrow{\quad w_1 \quad} q' \xrightarrow{r,-\gamma} p'' \xrightarrow{\quad w_2 \quad} q$$

By definition of the set W, we have $(p',q') \in W$ and $(p'',q) \in W$. The corresponding run of B is:

$$(p,q) \xrightarrow{c,+\gamma} (p',q') \xrightarrow{\quad w_1 \quad} (q',q') \xrightarrow{r,-\gamma} (p'',q) \xrightarrow{\quad w_2 \quad} (q,q)$$

A state of B is a pair (p,q) of states of A. The first component, p, is the state of A (in the corresponding run). While the second state q, the target state, is the state that A will reach after reading u.

We made the assumption that the VPA stack is empty whenever it is in a final state, therefore the pair formed by the initial state and the final state of any accepting run is in W. Initially B guesses the initial and final state of the run $(p,q) \in I \times F$, which must also belong to W.

Suppose that the current state of B is $(p,q) \in W$ and its stack is $\sigma \in \Gamma'^*$. Moreover suppose that $((q,q),\sigma)$ is a co-accessible configuration.

When reading a call symbol c, B simulates the transition $p \xrightarrow{c,+\gamma} p'$ of A by pushing $\gamma' = (\gamma,q)$ on the stack (the target state is pushed also on the stack) and going in state (p',q') where q' is the state that B guesses A will reach after reading w_1. B chooses a target state q' such that there exists some transition $A \models q' \xrightarrow{r,-\gamma} p''$ with $(p'',q) \in W$. This ensures that the chosen run is co-accessible.

When reading a return symbol r, the state of B must be of the form (p,p) as this is the guess that B made at the corresponding call position. Suppose it pops the stack symbol $\gamma' = (\gamma,q')$, In that case B simulates the transition $A \models p \xrightarrow{r,-\gamma} p'$, it goes in state (p',q') provided that $(p',q') \in W$ (to ensure the co-accessibility of the reached configuration).

The internal transitions are treated similarly.

We now defined the reduced VPA $B = (Q', I', F', \Gamma', \delta')$ with $Q' = W$, $I' = (I \times F) \cap W$, $F' = F \times F$, $\Gamma' = \Gamma \times Q$, and $\delta' = \delta'_c \uplus \delta'_i \uplus \delta'_r$ with:

- $B \models (p,q) \xrightarrow{c,+(\gamma,q)} (p',q')$ iff $(p',q') \in W$, $A \models p \xrightarrow{c,+\gamma} p'$ and there exist $r \in \Sigma_r$ and $p'' \in Q$ such that $A \models q' \xrightarrow{r,-\gamma} p''$ with $(p'',q) \in W$,
- $B \models (p,p) \xrightarrow{r,-(\gamma,q')} (p',q')$ iff $(p',q') \in W$, and $A \models p \xrightarrow{r,-\gamma} p'$,
- $B \models (p,q) \xrightarrow{a} (p',q)$ iff $(p',q) \in W$, and $A \models p \xrightarrow{a} p'$.

It is not difficult to show, by induction on the length of the words, that A and B are equivalent and that B is reduced.

Finally we can get rid of the assumption that $L(A) \in L_{wn}(\Sigma)$ as follows [1]. We can construct a VPA A' over the structured alphabet

$$\Sigma' = (\Sigma_c \uplus \{c_0\}, \Sigma_i, \Sigma_r \uplus \{r_0\})$$

that accepts the language

$$L(A') = \{c_0^n w r_0^m \mid w \in L(A) \wedge c_0^n w r_0^m \in L_{wn}(\Sigma')\}$$

A' is obtained by adding an initial states from which A' reads any number of c_0, then it simulates A and finally it empties the stack by reading the exact number of r_0. A' takes care of always accepting by empty stack (by marking the stack symbols that are pushed on the empty stack). We construct B' the reduced equivalent of A', then we remove the transitions on c_0 and r_0. The result is a reduced VPA equivalent to the original VPA A.

Theorem 4.4.2 ([CRT11]). *For any* VPA *one can construct in* PTIME *an equivalent reduced, resp. trimmed,* VPA.

Note however that this construction does not preserve determinism.

4.5 Conclusion

In this chapter, we have recall the fundamental results for visibly pushdown automata. This indicates that, contrarily to pushdown automata, VPA form a

[1] This is however not required for the rest of this document, as we need the reduced procedure for well-nested VPL only.

	$\overline{L}$	$L_1 \cup L_2$	$L_1 \cap L_2$	$h(L)$	L^r	$L_1 L_2$	$L \cap \mathsf{REG}$
REG	yes	yes	yes	yes	yes	yes	yes
CFL	no	yes	no	yes	yes	yes	yes
dCFL	yes	no	no	no	no	no	yes
VPL	yes	yes	yes	no	no	yes	yes

Table 4.2: Closure under complement, union, intersection, morphism, mirror image, concatenation, and intersection with a regular language for the classes REG, CFL, dCFL, and VPL.

	emptiness / membership	equivalence / inclusion	universality
NFA	PTime	PSpace-c	PSpace-c
DFA	PTime	PTime	PTime
NPA	PTime	undec	undec
DPA	PTime	dec/undec	dec
VPA	PTime	ExpTime-c	ExpTime-c
dVPA	PTime	PTime	PTime

Table 4.3: Decision problems for ϵ-NFA, NFA, DFA, ϵ-NPA, NPA and DPA.

robust generalization of finite state automata. In table 4.2 we summarize the closure properties of VPA and compare them to finite state and pushdown automata. In table 4.3 we summarize the results regarding the decision procedures for VPA and compare them to finite state and pushdown automata.

Related Work

The introduction of visibly pushdown languages in [AM04] inspired numerous related works, which are in part listed in [Alu11].

The nested words and nested word automata were introduced in [AM06, AM09]. They are an equivalent formalism where the nesting relation of words are made explicit, instead of being induced by the structure of the alphabet.

In [Alu07], the strong connection between VPA and tree automata is investigated. A formalism equivalent to VPA is introduced in [GNR08]. These automata are VPA that runs directly on trees, *i.e.* they run on the linearization of the tree (See Section 6.4).

Finally, some generalization of VPA have been proposed [NS07, Cau06, CH08]. These classes are strictly more expressive than VPA but still preserve the same closure properties and the same complexity for the decision problems.

Chapter 5

Visibly Pushdown Transducers

Contents

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

Following the definitions of NFT and PTs in Chapter 3, 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.

We saw in Chapter 3 that the excellent closure and decision properties of NFA translate into good properties for NFT. On the other hand, non-closure and undecidability results for pushdown automata yield a class of transducers, the pushdown transducers, for which nearly all decision problems are undecidable and closure properties are scarce. In this section, we show that the good properties of VPA, with regards to closure and decision problems, yield an interesting class of transducers.

The main results of this chapter are the decidability of the functionality and k-valuedness. This is obtained, in part, thanks to the product construction for VPA. This construction yields a machine capable of simulating two (or more) runs in parallel on the same input. This is exactly what is needed for testing functionality or more generally k-valuedness: a transducer is not functional, resp. k-valued, if there are two, resp. $k+1$, accepting runs on the same input that produce two, resp. $k+1$, different outputs.

We start this chapter by defining formally visibly pushdown transducers and their semantics. Then we investigate their domains and ranges as well as closure and non-closure properties, in particular we show that, in contrast to the class of NFT, the class of VPTs is not closed under composition. Next, we turn to decision problems, notably we show that the type checking against VPL is undecidable, furthermore, we show that it is already the case for NFT (against VPL). We dedicate the next sections to proving the decidability of functionality and k-valuedness for VPTs. As a consequence, we show that the inclusion and the equivalence problems are decidable for functional VPTs. We end this chapter with a comparison between VPTs, NFT and NPT.

5.1 Definitions

Let $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$ be a structured alphabet and Δ an alphabet.

Definition 5.1.1 (Visibly Pushdown Transducers)**.** *A visibly pushdown transducer (*VPT*) from Σ to Δ is a pair $T = (A, \Omega)$ where $A = (Q, I, F, \Gamma, \delta) \in$* VPA *is the underlying automaton and Ω is a morphism from δ to Δ^* called the output.*

A *run* ρ of T over a word $u = a_1 \dots a_l$ is a run of its underlying automaton A (Definition 4.1.2), *i.e.* it is a sequence of transition $\rho = t_1 \dots t_l \in \delta^*$ (where δ is considered as an alphabet and δ^* is the set of words on δ). The output of ρ is the word $v = \Omega(\rho) = \Omega(t_1 \dots t_l) = \Omega(t_1) \dots \Omega(t_l)$. 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 $R(T) = \{(u, v) \mid \exists q \in I, q' \in F, \sigma \in$

Γ^*, $(q, \bot) \xrightarrow{u/v} (q', \sigma)\}$. We say that a transduction is a VPT transduction whenever there exists a VPT that defines it.

As usual, the *domain* of T (denoted by $dom(T)$), resp. the *range* of T (denoted by $range(T)$), is the domain of $R(T)$, resp. the range of $R(T)$. We say that T is functional whenever $R(T)$ is, and that T is deterministic 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 5.1.2. *As* VPA *do not allow ϵ-transitions, neither do* VPT *on the input. However some transitions might outputs the empty word ϵ. In that sense the class of* VPT *defined in Definition 5.1.1 is the class of real-time* VPT.

*We defined in Chapter 3 two classes of transducers, ϵ-*NFT *and ϵ-*NPT*, with ϵ-transitions. Contrarily to their ϵ-free counterparts, these classes enjoy closure under inverse.*

Adding ϵ-transition 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 drive the behavior of the stack, therefore, this is only compatible with internal ϵ-transitions. 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*:*

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

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

Remark 5.1.3. *Visibly pushdown transducers have been first introduced in [RS08] and independently in [TVY08]. In these papers,* VPTs *allow for ϵ-transitions that can produce outputs and only a single letter can be produced by each transition. Using ϵ-transitions the way of [RS08] and [TVY08] causes many interesting problems to be undecidable, such as functionality and equivalence (even of functional transducers). Therefore we prefer to stick to Definition 5.1.1 of* VPTs*, they are exactly the so called nested word to word transducers of [SLLN09] and corresponds to the definition of [FRR+10b]. XML-DPDTs [KS07], equivalent to left-to-right attribute grammars, correspond to the deterministic* VPTs.

5.2 Examples

We present various examples to illustrate the definition of VPT.

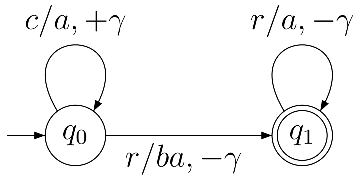


Figure 5.1: A VPT T_1 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Example 5.2.1. *This first example is a deterministic* VPT $T_1 = (A, \Omega)$ *represented in Figure 5.1. It operates on the input structured alphabet* $\Sigma = (\Sigma_c, \Sigma_i, \Sigma_r)$ *where* $\Sigma_c = \{c\}$*,* $\Sigma_r = \{r\}$*, and* $\Sigma_i = \{\}$*. It produces word 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* $L(A_1)$ *of* A_1*, it is 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 follow.* $\Omega_1((q_0, c, \gamma, q_0)) = a$*,* $\Omega_1((q_0, r, \gamma, q_1)) = ba$*, 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 a* γ *on 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 a* γ *(that is if at least one* c *was read before), it pops the* γ *symbol (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* γ *symbol on the stack, for each* r *it outputs an* a *and pops a* γ*. The run is accepting if and only if it ends in the final state* q_1*.*

Clearly T_1 *implements the following transductions:*

$$c^n r^m \to a^n b a^m \quad \textit{for all} \quad 1 \leq m \leq n$$

The range of T_1 is a context-free language but it is not a visibly pushdown language for any partition of the output alphabet:

$$range(T_1) = \{a^n b a^m \mid 1 \leq m \leq n\}$$

□

The second and third examples are a slight modification of the first example.

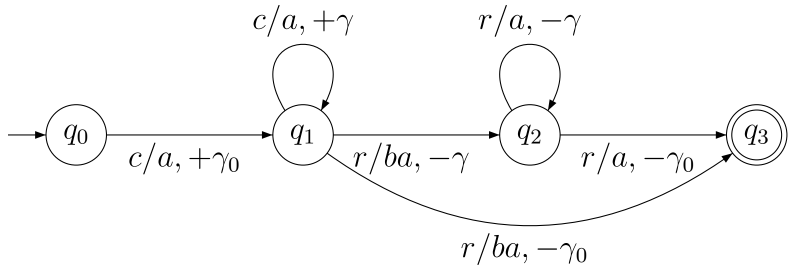


Figure 5.2: VPT T_2 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

Example 5.2.2. *The* VPT T_2 *of Figure 5.2 behaves similarly as the* VPT T_1 *of the previous example, but uses an additional stack symbol γ_0 to ensure that the number of c is equal to the number of r.*

Clearly it implements the transductions:

$$c^n r^n \rightarrow a^n b a^n \quad \textit{for all} \quad 1 \leq n$$

□

Example 5.2.3. *The* VPT T_3 *of Figure 5.3 behaves similarly as the* VPT T_2 *of the previous example, but in addition it can 'start again' after processing a word $c^n r^n$. The* VPT T_3 *implements the transduction:*

$$c^{n_1} r^{n_1} \dots c^{n_k} r^{n_k} \rightarrow a^{n_1} b a^{n_1} \dots a^{n_k} b a^{n_k} \quad \textit{for all } k \geq 1, \textit{ and } n_1, \dots n_k \geq 1$$

□

The next example contains 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.

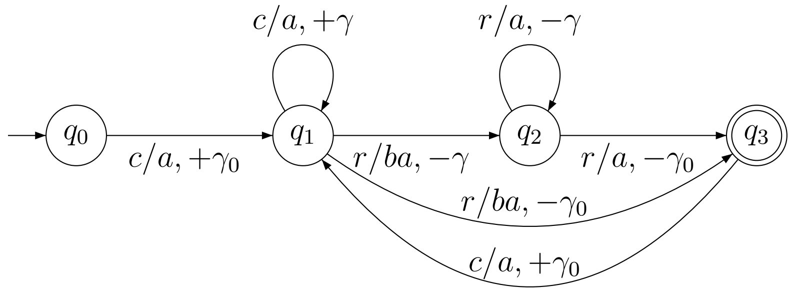


Figure 5.3: VPT T_3 on $\Sigma_c = \{c\}$ and $\Sigma_r = \{r\}$.

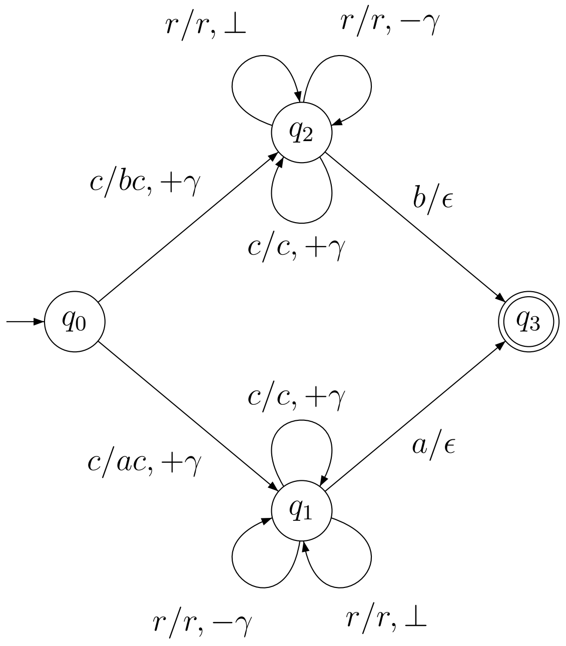


Figure 5.4: A VPT T_4 on $\Sigma_c = \{c\}$, $\Sigma_r = \{r\}$, and $\Sigma_i = \{a, b\}$.

Example 5.2.4. *The* VPT T_4 *of Figure 5.4 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 transduction:*

$$cw\alpha \to \alpha cw \quad \text{where } \alpha \in \{a, b\}, w \in \{c, r\}^*$$

Note that T_4 uses return transition on $\perp$, it permits to read the return symbol r even when the stack is empty. □

5.3 Properties

In this section, we present results about expressiveness, closure properties and decision problems for VPTs.

5.3.1 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 VPL (Porposition 4.1.3), and as the image of a VPL by a morphism is a context-free language (Proposition 3.2.12), so is the range of a VPT. Finally, by Proposition 4.2.5, it is easy to show that for any CFL L there exists a VPT such that its range is L.

Proposition 5.3.1 (Domain, range and image)**.** *Let T be a* VPT*. The domain dom(T) of T is a* VPL*, and its range range(T) 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 VPTs is a strict subclass of real-time pushdown transducers. Indeed, any VPT is clearly a real-time NPT. But some NPT transductions are not VPT transduction, for instance those NPT transduction whose domain is a context-free language but not a VPL.

Proposition 5.3.2 (Relation to NPT)**.** *The class of* VPT *transductions is a strict subclass of* NPT *transductions: R(VPT) $\subsetneq$ R(NPT).*

On the other hand, the class of VPTs subsumes the class of NFT.

Proposition 5.3.3 (Relation to NFT)**.** *The class of* NFT *transductions is a strict subclass of* VPT *transductions: R(NFT) $\subsetneq$ R(VPT).*

Proof. We show that for any NFT we can construct an equivalent VPT.

Let $T = (A, \Omega)$ a NFT, with $A = (Q, I, F, \delta)$. We let, as in Proposition 4.2.4, $A' = (Q, I, F, \Gamma, \delta')$ be a VPA with $\Gamma = \{\gamma\}$ and δ' is defined as follows. For each $(q, a, q') \in \delta$, if $a \in \Sigma_c$ then $(q, a, \gamma, q') \in \delta_c$, if $a \in \Sigma_r$ then $(q, a, \gamma, q') \in \delta_r$ and $(q, a, \perp, q') \in \delta_r$, and if $a \in \Sigma_i$ then $(q, a, q') \in \delta_i$.

Moreover we define the morphism $\pi : \delta'^* \to \delta^*$ as follows: for any $t \in \delta'$ with $t = (q, a, \gamma, q')$, $t = (q, a, \perp, q')$, or $t = (q, a, q')$, we define $\pi(t) = (q, a, q')$. It is easy to check that for any run ρ of A there is a unique run ρ' of A' with $\pi(\rho') = \rho$.

For any $t \in \delta'$ we define $\Omega'(t) = \Omega(\pi(t))$. Let $T' = (A', \Omega')$, we have $\Omega'(\rho') = \Omega(\pi(\rho')) = \Omega(\rho)$, therefore $R(T) = R(T')$. □

This last result obviously does not hold for ϵ-NFT, as our VPT do not have ϵ-transitions, they can, therefore, not implement all transductions defined by ϵ-NFT.

5.3.2 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.

Definition 5.3.4 (Product of T by A). *Let $T = (A, \Omega)$ be a* VPT *with $A = (Q, I, F, \Gamma, \delta)$, and $B = (Q', I', F', \Gamma', \delta')$ be a* VPA. *The* restriction *of T to B is a* VPT *denoted by $T_{|B} = (A \times B, \Omega')$ such that for all $(t, t') \in \delta \otimes \delta'$ we have $\Omega'((t, t')) = \Omega(t)$.*

Recall that the product of A with B (Definition 4.2.1) is defined as the VPA $A \times B = (Q \times Q', I \times I', F \times F', \Gamma \times \Gamma', \delta \otimes \delta')$, where elements of $\delta \otimes \delta'$ are identified with pairs of $(t, t') \in \delta \times \delta'$ with the same input letter. Any run $\rho \in (\delta \otimes \delta')^*$ is identified with a pair formed by a run of A, ρ_A, and a run of B, ρ_B, it is accepting if both runs are accepting. The output $\Omega'(\rho)$ is $\Omega(\rho_A)$. Clearly, the product of T with a VPA B defines the restriction of the transduction $R(T)$ to the language of B.

Proposition 5.3.5 (Domain restriction). *Let T be a* VPT*, B a* VPA *and $L = L(B)$. We have: $R(T)|_L = R(T_{|B})$, and $T_{|B}$ can be computed in* PTime.

The range of $T_{|B}$ is equal to the image of $L(B)$ by T. Therefore by Proposition 5.3.1, the image of a VPL by a VPT is a context-free language.

Corollary 5.3.6. *Let T be a* VPT*, and B be a* VPA*. The image of $L(B)$ by T is a context-free language and a* 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 VPTs are closed under *union.*

Proposition 5.3.7 (Closure under union)**.** *The class of* VPTs *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 5.3.8 (Non-Closure)**.** *The class of* VPT*, resp.* dVPT*, is not closed under intersection, complement, composition, nor inverse. The class of* dVPT *is not closed under union.*

Proof. To show that the class of VPTs and dVPT is not closed under intersection the proof of this fact (Proposition 3.1.17) for NFT carries over to VPT. Indeed, when the alphabet Σ contains only internals, the class of VPT, resp. dVPT, identifies with the class of NFT, resp. DFT.

Non-closure under inverse is a consequence of Proposition 5.3.1: 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.

The latter argument is also sufficient to show non-closure under composition for VPT and dVPT.

Non-closure under composition can be simply proved by using Proposition 5.3.6 and by producing two VPTs whose composition transforms a VPL into a non CFL language. More formally, let $\Sigma = (\Sigma_c, \Sigma_r, \Sigma_i)$ be the structured alphabet with $\Sigma_c = \{c_1, r_1, c_2\}$, $\Sigma_r = \{r_2, c_3, r_3\}$, $\Sigma = \{a\}$. First consider the following VPL language: $L_1 = \{c_1^n r_2^n a^m \mid n, m \geq 0\}$. We can easily construct a VPT that transforms L_1 into the language $L_2 = \{c_1^n a^n r_3^m \mid n, m \geq 0\}$. Applying the identity transducer on L_2 restricted to well-nested words, it produces the non CFL language $L_3 = \{c_1^n a^n r_3^n \mid n \geq 0\}$. 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 theses two transducers are deterministic. Therefore this completes the proof that neither VPT nor dVPT are closed under composition. ☐

5.3.3 Decision Problems

The emptiness and translation membership problems are decidable in PTime for pushdown transducers 3.2.15. Therefore, since VPTs are pushdown transducers, these results trivially hold for VPTs.

Proposition 5.3.9 (Decidable problems). *The emptiness and the translation membership problems are decidable in* PTime *for* VPT.

The type checking problem is undecidable for NPT and NFT against pushdown automata (as a consequence of the undecidability of the inclusion of pushdown automata), but it is decidable against regular languages. We now show that the type checking problem is also undecidable for these classes of transducers when the input and output languages are given by VPA. Therefore the decidability of the type checking problem does no depend on the class of transducers but on the class of languages that specify the input and output languages: it is decidable for regular languages and undecidable for VPL and CFL. This also holds for VPTs. The results are summarized in Table 5.1.

Table 5.1: Decidability of the type checking problem

	NFA	VPA	NPA
NFT	PSpace-c	undec	undec
VPT	ExpTime	undec	undec
NPT	ExpTime	undec	undec

Theorem 5.3.10 (Type Checking). *The type checking problem for* NFT, VPT *and* NPT *against* VPA *are 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 NFT from $\hat{\Sigma}$ into Σ such that for all $u \in \hat{\Sigma}^*$ we have $T(u) = \pi_\Sigma(u)$. By Proposition 4.2.5, 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 4.3.3), so is the type checking of NFT against VPA. This undecidability trivially carries over to VPT and PT. □

As a consequence of the fact that for any NFT one can effectively construct an equivalent VPT, and the fact that equivalence and inclusion are undecidable for NFT we have the same result for VPTs.

Theorem 5.3.11. *The equivalence and inclusion problems are undecidable for* VPT.

We will see in the next section that the equivalence problem for dVPT is decidable. It is, as in the case of NFT, a consequence of the decidability of functionality.

5.4 Functional VPTs

In this section we prove that deciding whether a VPT is functional can be done in PTime. As a consequence, we show that the equivalence and the inclusion problems for functional VPTs are both decidable.

Let us start with an example.

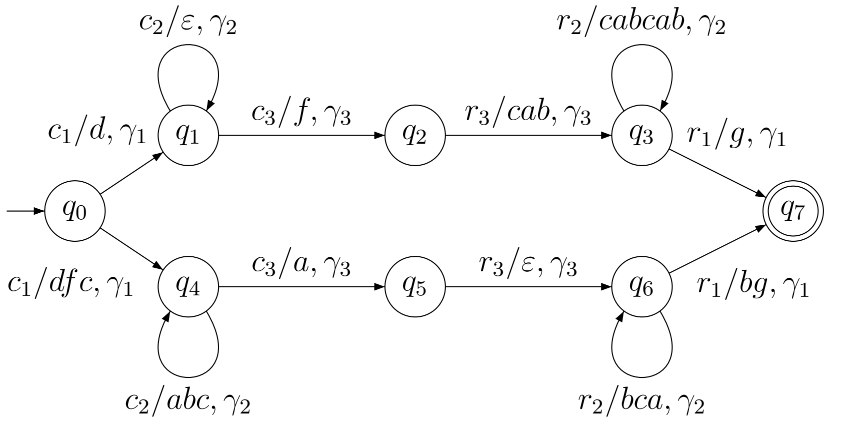


Figure 5.5: A functional VPT on $\Sigma_c = \{c_1, c_2, c_3\}$ and $\Sigma_r = \{r_1, r_2, r_3\}$.

Example 5.4.1. *Consider the* VPT *T of Figure 5.5. Call (resp. return) symbols are denoted by c (resp. r). 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 5.4.1 is functional. Indeed, the upper part performs the transformation:

$$c_1c_2^nc_3r_3r_2^nr_1 \to dfcab(cabcab)^ng = df(cab)^{2n+1}g$$

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 the size of the difference between the outputs of two runs over the same input might be unbounded (recall that the decision procedure for NFT relies on the boundedness of this delay for functional NFT). After reading $c_1c_2^n$ the upper part outputs just d, while the lower part outputs $dfc(abc)^n$ the difference or *delay* between both outputs is $fc(abc)^n$ which 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. An unbounded delay is therefore possible. This is not the case for functional NFT for which the delay between two runs on the same input is bounded (provided both run can be completed into an accepting run with a same input word).

The proof of the decidability of the functionality is done via a reduction to the *morphism equivalence problem* for context-free languages[1]. This problem asks, given two morphisms and a context-free language, whether the morphisms are equal on all words that belongs to the language. It was proved to be decidable in PTime by Plandowski [Pla94].

The structure of the reduction is as follows. Given a VPT T, we first construct a VPT T^2 (formally defined below), the square of T. The range of T^2, as the range of any VPT, is a context-free language. We then define two morphisms that maps outputs of T^2 to output of T. Finally, we show that T is functional if and only if these two morphisms are equivalent on the context-free language *range*(T^2). As stated above, this latter problem is decidable in PTime.

We first present the squaring construction. Then we show how to decide functionality based on the decidability of the morphism equivalence problem.

Square. 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

[1]This reduction first appear in [SLLN09] for proving the decidability of the equivalence of *deterministic* (and therefore functional) VPTs.

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.* $O_T \times O_T$. Furthermore, outputs of runs of T^2 are sequences of pairs of words in $\Omega(\delta)$.

Recall that the transitions of the product $A_1 \times A_2$ of two VPA, A_1 and A_2 (See Definition 4.2.1), 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 $A_1 \times A_2$.

Definition 5.4.2 (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 *$T^2 = (A \times A, \Omega')$ 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 5.3.1) over the alphabet $O_T \times O_T$.

Example 5.4.3. *The square T^2 of the* VPT *$T = (A, \Omega)$ of Example 5.4.1 is represented in Figure 5.6 (note that for the sake of readability we omitted the stack symbols). We have $\Omega(\delta) = \{\varepsilon, d, dfc, abc, a, f, cab, cabcab, g, bca, bg\}$, and the output alphabet of T^2 is the set $\Omega(\delta) \times \Omega(\delta)$, for example it contains the pairs $(\varepsilon, \varepsilon), (\varepsilon, d), (cabcab, bg) \ldots$*

The range of T^2 is a context-free language, it is the union of the following languages:

$$L_{11} = \{(d,d)(\varepsilon,\varepsilon)^n(f,f)(cab,cab)(cabcab,cabcab)^n(g,g) \mid n \geq 0\}$$

$$L_{14} = \{(d,dfc)(\varepsilon,abc)^n(f,a)(cab,\varepsilon)(cabcab,bca)^n(g,bg) \mid n \geq 0\}$$

$$L_{41} = \{(dfc,d)(abc,\varepsilon)^n(a,f)(\varepsilon,cab)(bca,cabcab)^n(bg,g) \mid n \geq 0\}$$

$$L_{44} = \{(dfc,dfc)(abc,abc)^n(a,a)(\varepsilon,\varepsilon)(bca,bca)^n(bg,bg) \mid n \geq 0\}$$

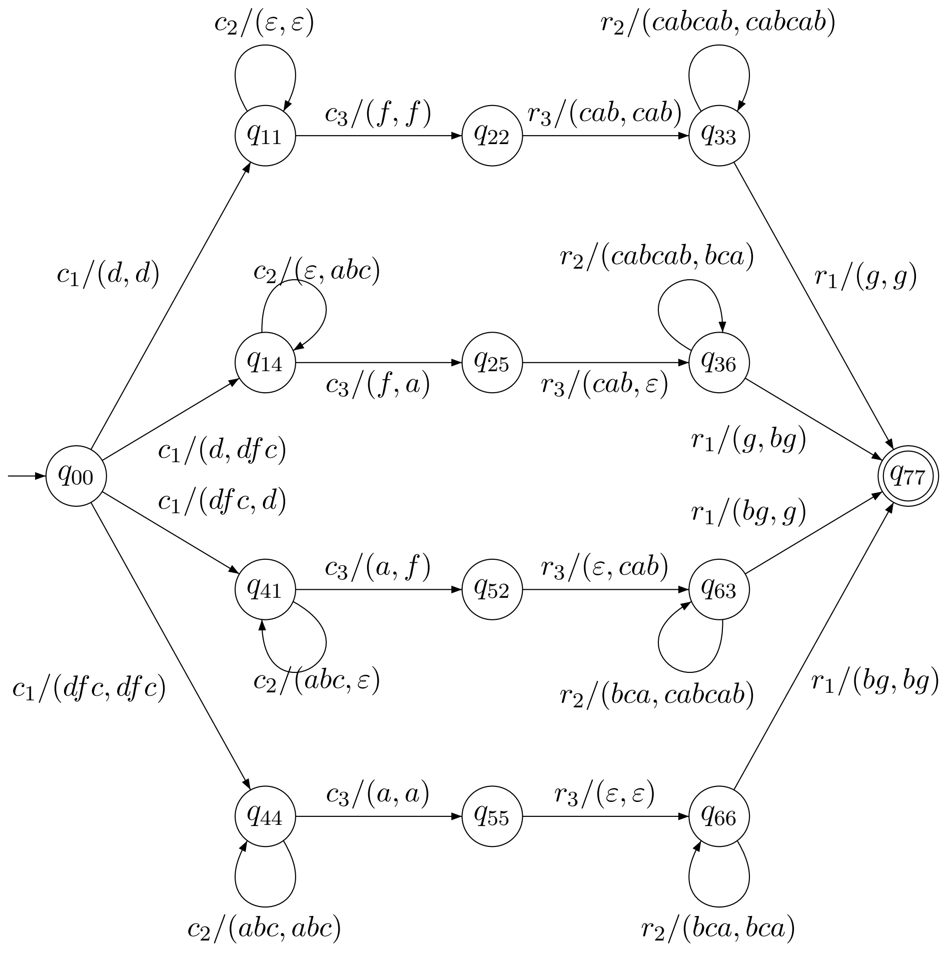


Figure 5.6: The square T^2 of T (stack symbols are omitted).

Functionality. 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 Φ_1 and Φ_2 as follows: $\Phi_1((u_1,u_2)) = u_1$ and $\Phi_2((u_1,u_2)) = u_2$. Clearly, T is functional if and only if these two morphisms are equal on any output of T^2, *i.e.* if for all $w = (u_1,u'_1)\dots(u_k,u'_k) \in \mathit{range}(T^2)$ we have $\Phi_1(w) = \Phi_2(w)$ 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.

Definition 5.4.4 (Morphism equivalence problem for CFL)**.** *Let Σ and Δ be two alphabets. The morphism equivalence problem asks, given* CFL *L over Σ and two morphisms Φ_1, Φ_2 from Σ to Δ^*, whether for all $u \in L$, $\Phi_1(u) = \Phi_2(u)$.*

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

Theorem 5.4.5 ((Plandowski [Pla94]).)**.** *Let Φ_1, Φ_2 be two morphisms from Σ to Δ and a CFG in Chomsky normal form G, testing whether Φ_1 and Φ_2 are equivalent on $L(G)$ can be done in* PTime.

As explained above, a VPT T is functional iff the morphisms Φ_1 and Φ_2 are equivalent on the range of T^2, which is a context-free language. By Proposition 5.3.1, one can construct in PTime a pushdown automaton that recognizes the range of T^2. This pushdown automaton can be transformed into a CFG in PTime (Proposition 3.2.9). In turn, this CFG can be transformed in PTime into a CFG in Chomsky normal form (Proposition 3.2.6). Finally, by Theorem 5.4.5, one can test in PTime the equivalence of two morphisms on the language of a CFG in Chomsky normal form. These observations yield a PTime procedure for testing the functionality of T.

Theorem 5.4.6 (Functionality)**.** *Functionality of* VPTs *is decidable in* PTime.

Equivalence and inclusion

Given two functional VPTs, 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 (Proposition 4.3.2). Therefore as a consequence of Theorem 5.4.6, we have:

Theorem 5.4.7 (Equivalence and inclusion of fVPTs)**.** *The inclusion and the equivalence problems for* fVPTs *are* ExpTime-c.

The EXPTIME bound is a consequence of the hardness of testing equivalence or inclusion of the domains. Testing the equivalence or the inclusion of the domains is easier when the VPTs are deterministic, it can be done in PTIME (Proposition 4.3.2). Therefore both procedures, that is equivalence or inclusion testing, can be done in PTIME when the VPTs are deterministic.

If we assume that both transducers are total, then obviously we do not have to test the equivalence or inclusion of their domains. Therefore, in that case equivalence and inclusion of their transductions can also be tested in PTIME.

Theorem 5.4.8 (Equivalence of deterministic/total VPTs). *The equivalence and the inclusion problems for* deterministic VPTs*, resp. are in* PTIME *[SLLN09]. The equivalence and the inclusion problems for* total *functional* VPTs *are in* PTIME*.*

5.5 k-valued VPTs

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

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 equivalence problem, which generalizes to k morphisms, instead of 2, the morphism equivalence problem.

The main tool of the decision procedure is the class of bounded reversal counter automata introduced by Ibarra[Iba78]. We proceed in three steps. First we show that the procedure in [HIKS02] 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 [HIKS02] can be executed in co-NPTIME. Finally, we show how to use this last result to decide the k-valuedness problem for VPTs 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 5.5.1. *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 [Min67].

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 [Iba78], 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 5.5.2. *Let $r, m \in \mathbb{N}$, a 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 a* NFA*, resp. when it is a* NPA*.*

For finite state automata with reversal-bounded counters emptiness is decidable in PTime (if r and m are fixed)[GI83]. 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.

Theorem 5.5.3 ([GI83])**.** *Let $r, m \in \mathbb{N}$ be fixed. The emptiness problem for r-reversal m-counter finite state automata is decidable in* PTime*.*

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 [Iba78]. In [FRR$^+$10b], we show that this procedure can be executed in co-NPTime. 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 [VSS05] (See Proposition 3.2.21).

Before presenting the proof we need a simple preliminary result. A machine with one r-reversal counter, can be simulated by a machine with $(r + 1)/2$ 1-reversal counters as follows. The transitions that occur during the first increasing mode of the counter and the first decreasing mode are simulated by the first 1-reversal counter. When switching from the first decreasing mode to the second increasing mode, the second 1-reversal counter takes over (it is initialized by increasing it as many time as one can decrease the previous counter till reaching zero). Therefore, the sequence of $\lceil (r + 1)/2\rceil$ 1-reversal counters simulates one r-reversal counter, in such a way that the value of the original counter in between its i-th and $i + 1$-th increasing mode is equal to the value of the i-th counter. More generally, a machine with m r-reversal counters, can be simulated by one machine with $m\lceil (r + 1)/2\rceil$ 1-reversal counters.

Lemma 5.5.4. *Let $r, m \in \mathbb{N}$ be fixed. For any $A \in$* $\mathsf{RBCA}(r, m)$*, resp.* RBCPA *(r,m), one can construct in* PTime *some $A' \in$* $\mathsf{RBCA}(1, \lceil m(r + 1)/2\rceil)$*, resp.* $\mathsf{RBCPA}(1, \lceil m(r + 1)/2\rceil)$*, such that $L(A) = L(A')$.*

We now turn to the proof of the NP-easiness of testing emptiness of RBCPA. Given a reversal-bounded counter automaton A, the idea is to construct a semi-linear set for the Parikh image of the language of A. The emptiness of A then reduces to the emptiness of its Parikh image.

To construct the Parikh image of A, we proceed in two steps. First we construct a pushdown automaton B that simulates A in the following sense. It recall the state of A, fully simulates the stack of A, but, as it can not encode the counters, it makes visible the operations on the counters as follows. The input alphabet is augmented with two symbols per counter, they represent an increase, resp. a decrease, on the corresponding counter. When A increases or decreases a counter, B must read the corresponding increase or decrease symbol. The rest of the behavior of A, *i.e.* current state and stack operation, is simulated by B. All words accepted by A are accepted by B (once you remove from these the symbols of the operation on the counters). On the other hand, some words accepted by B are not accepted by A because zero tests and negative counter values are not taken into account. One can construct an existential Presburger formula representing the Parikh image of B. The second step is to apply a simple modification to this formula, so that the new formula accepts words that the former accepts provided that, for each counter, the number of increase and decrease are equal. As we explain in the proof, we can suppose that any accepting run of A ends with all its counters with value 0. Therefore, the new Presburger formula represents the Parikh image of A. The language of A is empty iff its Parikh image is, and when given as an existential Presburger formula this can be tested in NPTime.

Lemma 5.5.5. *Let $m, r \in \mathbb{N}$ be fixed. The emptiness problem for r-reversal m-counters pushdown automata is decidable in co-*NPTime.

Proof. First, by Lemma 5.5.4, we can suppose that $r = 1$, that is we can suppose that the automaton counters are *one*-reversal.

Second, note that we can suppose, without loss of generality, that the machine does the zero tests at the end of the computation only. Indeed, when testing a counter for zero the machine considers three cases: (*i*) if it has not increased the counter yet, then the test is trivially true, (*ii*) otherwise if it is in increasing mode, then the test is trivially false, and finally (*iii*) if it is in decreasing mode, it flags the counter so that it can not be decreased anymore (because it guesses it is zero) and postpones the zero test until the end of the computation.

We now show that the emptiness of a 1-reversal pushdown machine A with k

counters on an alphabet Σ is in co-NPTime. For this, we recall the construction of [HIKS02] for testing emptiness of reversal-bounded machines with counters. The idea is to construct a semi-linear set for the Parikh image of A (Section 3.2.2). The emptiness of A then reduces to the emptiness of its Parikh image. Following [HIKS02], one extends the alphabet Σ with $3k$ letters $+_j, -_j, s_j \notin \Sigma$ intended to simulate the increasing, decreasing transitions of the j-th counter, and the transitions that do not change the j-th counter (skip). We denote by Σ_+ this alphabet. We construct a pushdown automaton B on Σ_+ that simulates A. When reading a letter $a \in \Sigma$, B tries to apply a transition of A, and passes into a mode in which it verifies that the next letters correspond to the increasing, decreasing or skip actions on the counters of the transition. Moreover, since A is 1-reversal bounded, B has to ensure that each counter does at most one reversal. The language of B is the set of words of the form $w = a_1t_1a_2t_2 \dots a_nt_n$ where $a_i \in \Sigma$ and each t_i is a word of the form $b^i_1 \dots b^i_k$ where $b^i_j \in \{+_j, -_j, s_j\}$, $j \in \{1, \dots, k\}$. Moreover, we require that (i) there exists a run of A on $a_1 \dots a_n$ ending up in a final state such that the counter actions of the transitions correspond to $t_1 \dots t_n$ (ii) for all $j \in \{1, \dots, k\}$, $b^1_j \dots b^n_j \in \{+_j, s_j\}^*\{-_j, s_j\}^*$ (one reversal). Let $\psi(w) = a_1 \dots a_n$ and $\psi_j(w) = b^1_j \dots b^n_j$ for all $j \in \{1, \dots, k\}$. Condition (i) is enforced by a simple simulation of A, and condition (ii) is enforced by adding vectors of $\{+, -\}^k$ to the states indicating whether the j-th counter is in increasing or decreasing mode. Note that $L(A) \subseteq \psi(L(B))$, but this inclusion may be strict, as we do not require that the counters end up in a zero value. More formally, we have $L(A) = \bigcap_{j=1}^k \{\psi(w) \mid w \in L(B) \text{ and } \psi_j(w) \in s_j^*(+_j.s_j^*)^\ell(-_j.s_j^*)^\ell, \ell \geq 0\}$.

As $L(B)$ is a context-free language, it is known by Parikh's theorem that the Parikh image of $L(B)$ is semi-linear. Therefore there exists an existential Presburger formula ϕ with $|\Sigma| + 3k$ free variables $(x_a)_{a\in\Sigma}$ and $(x_{+_j}, x_{-_j}, x_{s_j})_{j\in\{1,\dots,k\}}$ which defines the Parikh image of $L(B)$. Moreover, this formula can be constructed in time $O(|B|)$ (Proposition 3.2.21[VSS05]). Finally, the formula $\exists x_{+_1}\exists x_{-_1}\exists x_{s_1} \dots \exists x_{+_k}\exists x$
$\bigwedge_{j=1}^k x_{+_j} = x_{-_j}$ defines exactly the Parikh image of $L(A)$. Since B can be constructed in $O(|A| \cdot 2^k)$ (which is polynomial as k is fixed) and the satisfiability of existential Presburger formulas is in NP (Proposition 3.2.20), one gets an NP algorithm to test the emptiness of A.

We now provide the construction of B. Let $A = (\Sigma, Q, \Gamma, F, I, \delta)$ be a 1-reversal pushdown automaton with k counters where $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \{pop, push\} \times \{+, -, skip\}^k \times Q$ is the transition function. If the j-th component is $+$ then the j-th counter is increased by one. If the j-the component is $-$ and the j-th counter is not 0, then the counter is decreased by 1, if the j-th component is *skip* then nothing is changed on the j-th counter. We also assume that any computation is accepting if it is accepting by the underlying pushdown automaton and the counter values start and end with zero, and there is only one reversal per counter.

The pushdown automaton B on Σ_+ is constructed as follows. Its set of states is $Q' = ((\delta \times \{1, \ldots, k\}) \cup Q) \times \{+, -\}^k$. The components $+,-$ are used to ensure that there is at most one reversal. When reading a letter $a \in \Sigma$, B tries to apply a transition $t \in \delta$ and goes in a mode where it checks that the next letters correspond to the counter operations of t (in order). For instance, suppose that it is in state $(q, \overline{m})$ where $\overline{m} = (m_1, \ldots, m_k) \in \{+, -\}^k$ and reads an $a \in \Sigma$. Suppose that there is a transition $t = (q, a, \gamma, push, v_1, \ldots, v_k, q') \in \delta$. Then B pushes γ and goes in state $(t, 1, \overline{m})$, for which the only possible continuation is to read a letter $+_1$ if $v_1 = +$, $-_1$ if $v_1 = -$, or s_1 if $v_1 = skip$. Moreover, if the next letter is $+_1$ and m_1 was $-$ (indicating that there was already one reversal), then there is no transition. Otherwise the tuple $\overline{m}$ is update as follows: m_1 is replaced by $-$ if the next letter is $-_1$, by $+$ if the next letter is $+_1$, and let unchanged if it was a *skip* action. It results in a new tuple $\overline{m'}$, and the B goes to $(t, 2, \overline{m'})$ and applies the same rule, and so on until it reaches state $(t, k, \overline{m''})$. Then it applies again the same rule but goes to the state $(q', \overline{m''})$ where a new cycle group of letters can be read. Therefore B has $O(|A| \cdot k \cdot 2^k)$ states. □

Recently, it was proved that this co-NP upper bound is also a lower bound, that is, the emptiness problem is co-NPComplete [HL11]. The proof is a reduction of the knapsack problem and takes advantage of the fact that the unary representation of any natural number n can be defined by a pushdown automata of size $O(\log n)$ (See Example 3.2.13).

Theorem 5.5.6 ([Iba78, FRR$^+$10b, HL11]). *Let $m, r \in \mathbb{N}$. The emptiness problem for r-reversal m-counters pushdown automata is co-*NPComplete.

5.5.2 Multiple Morphism Equivalence Problem

The *multiple morphism equivalence problem* is a generalization of the morphism equivalence problem (Definition 5.4.4). 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 5.5.7 (Multiple Morphism Equivalence on Context-Free Languages). *Given ℓ pairs of morphisms $(\Phi_1, \Psi_1), \ldots, (\Phi_\ell, \Psi_\ell)$ from Σ_1^* to Σ_2^* and a context* *free language L on Σ_1, $(\Phi_1, \Psi_1), \dots, (\Phi_\ell, \Psi_\ell)$ 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 [HIKS02] on any class of languages whose Parikh images are effectively semilinear. 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 RBCA.

Theorem 5.5.8. *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 [HIKS02] 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 $L' = \{w \in L \mid \Phi_i(w) \neq \Psi_i(w)$ for all $i\}$. Clearly, $L' = \varnothing$ iff the morphisms are equivalent on L. We construct a pushdown automaton A' augmented with 2ℓ counters $c_{11}, c_{12}, \dots, c_{\ell 1}, c_{\ell 2}$ that simulates A on the input word and counts the lengths of the outputs by the 2ℓ morphisms. For all $i \in \{1, \dots, \ell\}$, 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 $a \in \Sigma_1$, 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 $i \in \{1, \dots, \ell\}$, 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 $O(n \cdot m \cdot |\Sigma_2|^{2\ell})$ 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 5.5.6). □

5.5.3 Deciding k-valuedness

We extends the squaring construction we used for deciding functionality. We define a notion of k-power for the class of VPTs, 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 VPTs (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 $O_T = \Omega(\delta)$ is 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 5.5.9 (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 cartesian product of δ) to $(O_T)^k$ defined by $\vec{\Omega}(t_1, \dots, t_k) = (\Omega(t_1), \dots, \Omega(t_k))$.*

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

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

if and only if

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

The outputs of T^k are sequences of k-uples 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 $\Phi_1, \dots, \Phi_k$ as follows:

$$\begin{array}{rcll} \Phi_i & : & (O_T)^k & \to & \Sigma^* \\ & & (u_1, \dots, u_k) & \mapsto & u_i \end{array}$$

Clearly, we obtain the following equivalence:

Proposition 5.5.10. *T is k-valued iff $(\Phi_i, \Phi_j)_{1 \leq i \neq j \leq k+1}$ 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 $\Phi_i(\beta) = \Phi_j(\beta)$. There exists $u \in dom(T^{k+1})$ such that $T^{k+1}(u) = \beta$. The word β can be decomposed as $\beta = (v_1^1, \dots, v_{k+1}^1) \dots (v_1^n, \dots, v_{k+1}^n)$ where $(v_1^j, \dots, v_k^j) \in (O_T)^{k+1}$ for all j. Since T is k-valued, there exists $i \neq j$ such that $v_i^1 \dots v_i^n = v_j^1 \dots v_j^n$. By definition of Φ_i and Φ_j, $\Phi_i(u) = v_i^1 \dots v_i^n = w_j^1 \dots w_j^n = \Phi_j(u)$.

	T^{-1}	$\overline{T}$	$T_1 \cup T_2$	$T_1 \cap T_2$	$T_1 \circ T_2$
ϵ-NFT	yes	no	yes	no	yes
NFT	no	no	yes	no	yes
DFT	no	no	no	no	yes
ϵ-NPT	yes	no	yes	no	no
NPT	no	no	yes	no	no
DPT	no	no	no	no	no
VPT	no	no	yes	no	no
dVPT	no	no	no	no	no

Table 5.2: Closure under inverse, complement, union, intersection, and composition for ϵ-NFT, NFT, DFT, ϵ-NPT, NPT, DPT, VPT and dVPT.

Conversely, suppose that T is not k-valued. Therefore there exists $u \in dom(T) = dom(T^{k+1})$ and $v_1, \dots, v_{k+1}$ 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 $v_i^1 \dots v_i^n$ such that $\beta = (v_1^1, \dots, v_{k+1}^1) \dots (v_1^n, \dots, v_{k+1}^n) \in T^{k+1}(u)$. By definition of $\Phi_1, \dots, \Phi_{k+1}$, we get $\Phi_i(\beta) = v_i \neq v_j = \Phi_j(\beta)$ for all $i \neq j$. Therefore, Φ_i and Φ_j are not equivalent on $range(T^{k+1})$. □

By Proposition 5.3.1 the language $range(T^k)$ is a context-free language. By Theorem 5.5.8, as $range(T^k)$ is represented by an automaton of polynomial size if k is fixed, we get:

Theorem 5.5.11 (k-valuedness). *Let $k \geq 0$ be fixed. The problem of deciding whether a* VPT *is k-valued is in co-*NPTIME.

5.6 Conclusion

Visibly pushdown transducers define a class of transductions that lies in between the class of finite state transductions and the pushdown transductions.

The closure properties of each class is summarized in Table 5.2, and the decidability of the main decision problems are summarized in Table 5.3.

In conclusion, while the closure properties of VPT are not better than of pushdown transducers, the enjoy decidability of k-valuedness in co-NPTIME, of functionality in PTIME, and the equivalence of functional VPT is decidable, all those problems are undecidable for pushdown transducers. Moreover we conjecture

	emptiness / membership	equivalence / inclusion	universality	k-valuedness (1-val.)/ determinizable
ϵ-NFT	PTime	undec	undec	PTime/ PTime
NFT	PTime	undec	-	PTime/ PTime
functional NFT	PTime	PSpace-c	-	- / PTime
DFT	PTime	PTime	-	- / -
ϵ-NPT	PTime	undec	undec	undec / undec
NPT	PTime	undec	-	undec / undec
functional NPT	PTime	undec	-	- / undec
DPT	PTime	dec/undec	-	- / -
VPT	PTime	undec	-	co-NPTime (PTime) / open
functional VPT	PTime	ExpTime-c	-	- / open
dVPT	PTime	PTime	-	- / -

Table 5.3: Decision problems for ϵ-NFT, NFT, DFT, ϵ-NPT, NPT, DPT, VPT, fVPT, and dVPT.

that determinizability, equivalence of k-valued transducers and the bounded-valuedness problem are all decidable for VPT, while they are undecidable for NPT.

On the other hand, the decision problems for VPT are decidable whenever the corresponding problem is decidable for NFT. But, the class of NFT is closed under composition and has a decidable type checking problem against NFA, while the class of VPT is not closed under composition, and the corresponding type checking problem, *i.e.* type checking problem against VPA, is undecidable.

In the next chapter, we introduce a subclass of VPT, the class of well-nested VPT, closed under composition and whose corresponding type checking problem is decidable. Furthermore, we highlight the strong connection between tree transducers and wnVPT. This comparison also applies to VPT.

Chapter 6

Well-Nested VPTs

Contents

The main results of the previous chapter showed that functionality and, more generally, k-valuedness are decidable for VPTs. As a consequence, the equivalence and inclusion problem for functional VPTs 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 VPTs. 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 VPTs 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 chapter we introduce the class of well-nested VPTs (wnVPTs). These transducers are VPTs that produce words over a *structured output* alphabet $\Delta = (\Delta_c, \Delta_i, \Delta_r)$. The "visibly" restriction for wnVPTs 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 VPTs that is closed under composition and has a decidable type checking problem against VPLs.

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

- *For all $t_c = (q, c, \gamma, q') \in \delta_c$ and $t_r = (p, r, \gamma, p') \in \delta_r$, we have that $\Omega(t_1)\Omega(t_2)$ 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 VPTs.

Example 6.1.2. *In the example of Figure 6.1 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: $u_1v_1 = c_1c_2r_2c_2ar_1ar_2 \in L_{wn}$. 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: $u_2v_2 = c_1r_1a \in L_{wn}$. Finally, the lone internal transition produces a well-nested word: $c_1r_1 \in L_{wn}$.*

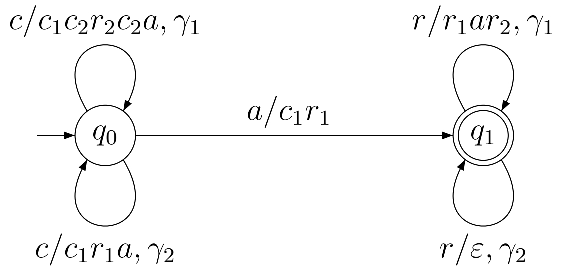


Figure 6.1: A wnVPT from $\Sigma_c = \{c\}, \Sigma_r = \{r\}$, and $\Sigma_i = \{a\}$ to $\Delta_c = \{c_1, c_2\}, \Delta_r = \{r_1, r_2\}$, and $\Delta_i = \{a\}$.

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

Moreover, there are context-free languages that are not the range of a wnVPT. For example the context-free language $\{r^nc^n \mid n \geq 0\}$, where $r \in \Sigma_r$ and $c \in \Sigma_c$, is not the range of any wnVPTs. But the context-free language $\{a^nb^n \mid n \geq 0\}$, where $a, b \in \Sigma_i$, is not a VPL, but it is the range of the VPT transduction $\{(c^nr^n, a^nb^n) \mid n \geq 0\}$, 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 wnVPTs 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.

Proposition 6.1.3. *Let $T \in \mathsf{wnVPT}$, $u \in L_{wn}$ and $v \in T(u)$, then $v \in L_{wn}$. Moreover, for all $u \in L_{wn}$, $v \in \Sigma^*$, and $q, q' \in Q$ if $T \models (q, \perp) \xrightarrow{u,v} (q', \perp)$ then $v \in L_{wn}$.*

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

6.2 Composition

The closure under composition of wnVPTs 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, produces two words whose concatenation forms a well-nested word. This well-nested property of a wnVPT carry over to sequences of transitions. This is formalized in the following Lemma and is proved easily by induction on the length of the length of the sequence.

Lemma 6.2.1. *Let T be a wnVPT, and let $T \models (q_1, \perp) \xrightarrow{v/w} (q_1', \sigma)$, and $T \models (q_2, \sigma) \xrightarrow{v'/w'} (q_2', \perp)$, then $ww' \in L_{wn}$.*

Given two well-nested VPTs 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 VPTs 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.

Proposition 6.2.2 (Closure properties)**.** *The class of* wnVPTs *is effectively closed under 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 $c_a r_a \in \Sigma_c \Sigma_r$).

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

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

Let $\Gamma' = \{\sigma \mid \exists q, q' \in Q_2, w \in \Omega_1(\delta_1) : (q, \perp) \xrightarrow{w/.} (q', \sigma)\}$. 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:

- $T_1 \models (q_1, \perp) \xrightarrow{c/v} (q_1', \gamma)$, and
- $T_2 \models (q_2, \perp) \xrightarrow{v/w} (q_2', \sigma)$.

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:

- $T_1 \models (q_1, \gamma) \xrightarrow{r/v} (q_1', \perp)$, and
- $T_2 \models (q_2, \sigma) \xrightarrow{v/w} (q_2', \perp)$.

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 $ww' \in L_{wn}$. By definition of δ, we have $T_2 \models (q_2, \perp) \xrightarrow{./w} (q_2', \sigma)$ and $T_2 \models (q_4, \sigma) \xrightarrow{./w'} (q_4', \perp)$. Therefore by Lemma 6.2.1 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 $\sigma_1 \dots \sigma_k = \gamma_1^2 \dots \gamma_{n2}^2$.

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 c 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 $(q_1, \gamma) \xrightarrow{c/v} (q_1', \bot)$ 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 correction of our construction. ☐

6.3 Type Checking against VPL

In this Section we show that the type checking problem is decidable for wnVPTs.

Theorem 6.3.1 (Type Checking). *Given a* wnVPT *T, two* VPAs *A_1, A_2, the problem of deciding if $T(L(A_1)) \subseteq L(A_2)$ is* ExpTime-c. *It is in* PTime *if A_2 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 Proposition 4.3.2).

To prove it is in ExpTime, we consider the wnVPT T_2 whose domain is $L(A_2)$ and whose relation is the identity relation. As wnVPTs are closed under composition, we can construct a wnVPT T' such that $T' = T \circ T_2$. Then note that $dom(T') = T^{-1}(L(A_2))$, and one can effectively construct in PTime a VPA that recognizes $dom(T')$ (Proposition 5.3.1). As $T(L(A_1)) \subseteq L(A_2)$ if and only if $L(A_1) \subseteq T^{-1}(L(A_2))$ and as all those transducers and automata can be constructed in polynomial time, we conclude that we can decide our problem in ExpTime by checking the former inclusion using the algorithm for language inclusion between VPA. □

6.4 Tree Transducers and wnVPTs

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 [Alu07]. 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 [ALH04]. A node n is encoded by a call (*i.e.* an opening tag) and its matching return (*i.e.* matching 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 [PS04]. We show that wnVPT are strictly less expressive than UTT. As a consequence, wnVPT are also less expressive than macro forest transducers and than macro tree transducers, two classes that include UTT.

A second model of unranked tree transformations is formed by the uniform tree transducers [MN03], a model inspired by XSLT [Cla99]. 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 transducers, such as the top-down tree transducers (TDTT) [CDG+07]. 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 wnVPTs.

6.4.1 Trees and Tree Transductions

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 $t_1 \cdot \dots \cdot t_k \cdot 0$ for $k \geq 0$ and $t_1, \dots, t_k \in \mathcal{U}_\Sigma$. We extend $\cdot$ to concatenation of hedges in the following manner:

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

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 6.4.1. *The following tree $f(f(a\ a\ a)b\ b\ b)$ is a 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. .*

Unranked Tree Transduction

An unranked tree transduction is a relation R 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 6.4.2 (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 6.4.3 (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 6.4.4 (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 6.4.5 (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* $R_{odd} = \{f(a^{2n}) \to f((ab)^n) \mid n \geq 0\}$.

6.4.2 wnVPT on 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:

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

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 6.4.6. *The word*

$$\overline{f}\ \overline{g}a\ \overline{g}bc\underline{g}\ \underline{g}\ def\ \underline{f}$$

is the linearization of the tree

$$f(g(ag(bc))def)$$

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

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

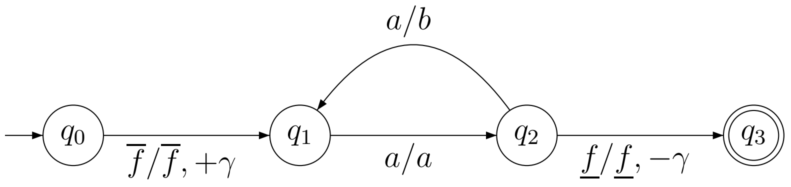

a/b
q_0
q_1
q_2
q_3
$\overline{f}/\overline{f}, +\gamma$
a/a
$\underline{f}/\underline{f}, -\gamma$

Figure 6.2: A wnVPT T_{odd} on $\Sigma_c = \{\overline{f}\}$, $\Sigma_r = \{\overline{r}\}$, $\Sigma_i = \{a, b\}$.

Example 6.4.7 (Yield)**.** *The yield transduction of Example 6.4.2 is easily defined by a* wnVPT *that deletes all calls and returns and copy any internal to the output.*

Example 6.4.8 (Odd)**.** *The odd transduction of Example 6.4.5 maps trees of the form $f(a^{2n})$ onto trees of the form $f((ab)^n)$, the linearization gives:*

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

It can be defined by the wnVPT T_{odd} *of Figure 6.2.*

Clearly, the duplicate and the swap transductions of Example 6.4.3 and 6.4.4 cannot be implemented by wnVPTs. Indeed, one can easily check that the linearization of the range of R_2 (the duplicate transuction) is not context-free. While regarding R_{swap}, with a classical pumping argument one can prove that it is not definable by a wnVPT.

6.4.3 Unranked Tree Transducers

In this section we present a model of unranked tree transducers (UTT) that run directly on unranked trees. They are defined as parameter-free *macro forest transducers (MFT)* [PS04].

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:

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

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

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

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:

$$\begin{array}{lcl} [\![0]\!]_\rho & = & \{0\} \\ [\![(q,x)]\!]_\rho & = & [\![q]\!](\rho(x)) \\ [\![ur]\!]_\rho & = & \{t\cdot h \mid t \in [\![u]\!]_\rho, h \in [\![r]\!]_\rho\} \\ [\![r_1r_2]\!]_\rho & = & \{h_1\cdot h_2 \mid h_1 \in [\![r_1]\!]_\rho, h_2 \in [\![r_2]\!]_\rho\} \\ [\![f(r)]\!]_\rho & = & \{f(h) \mid h \in [\![r]\!]_\rho\} \end{array}$$

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

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

Example 6.4.9 (Yield)**.** *The yield transduction of Example 6.4.2 is defined by a* UTT $T = (Q, \{q\}, \delta)$ *with* $Q = \{q\} \cup \{q_f \mid f \in \Sigma\}$ *and the following rules:*

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

where the procedure (or state) q guess (with non-determinism) whether the first tree, with its root labelled f, has children or not. If it has no child then it applies q_f, which output f. Otherwise, it recursively call q on the hedge of the children.

Example 6.4.10 (Swap)**.** *The swap transduction of Example 6.4.4 is defined by a* UTT $T = (Q, \{q_0\}, \delta)$ *with* $Q = \{q_0, q, q', q_\perp\}$ *and the following rules:*

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

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 6.4.11 (Duplicate and Odd). *The duplicate and odd transduction of Example 6.4.3 and Example 6.4.5 can be defined with a transducer similar to the* UTT *of the previous example.*

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

Proposition 6.4.12. UTT *are strictly more expressive than VPT.*

Proof. Let Σ be an alphabet and $T = (Q, I, F, \delta)$ 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 $(p_1, \perp)$ to $(q_1, \perp)$. 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 $p, p', p'', q, q' \in Q$:

$$(p,p)(0) \quad \rightarrow \quad 0$$

$$(p,q)(f(x_1)\cdot x_2) \quad \rightarrow \quad \mathsf{lin}^{-1}(uyv)\cdot(p'',q)(x_2) \text{ if } \begin{cases} t_c = (p, \overline{f}, \gamma, p') \in \delta_c \\ t_r = (q', \underline{f}, \gamma, p'') \in \delta_r \\ u = \Omega(t_c) \\ v = \Omega(t_r) \\ y = (p', q')(x_1) \end{cases}$$

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. □

Macro Tree and 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.

On the other hand, MFT are a slight generalization of MTT, they can be viewed as macro tree transducers (MTT) [EV85] with the ability to concatenate hedges. 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 [Man03]. Moreover, any MFT transduction can be obtained as the composition of two MTT [PS04]. 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 [EM03]. wnVPT transductions are linear size increase (as there is no duplication), therefore MSO transductions and macro tree transductions subsume wnVPT transductions.

6.4.4 Uniform Tree Transducers

Inspired by XSLT transformations, Neven and Martens [MN03] have introduced *uniform tree transducers* 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 [MN04, MN05, MN07].

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:

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

where $[\![.]\!]_h$ for an hedge $h = t_1 \cdot \dots \cdot t_k \in \mathcal{H}_\Sigma$ 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 wnVPTs. On the other hand, the transduction R_{odd} of Example 6.4.5 cannot be defined by a uniform tree transducer as different transformations are applied on odd and even children respectively.

Proposition 6.4.13. wnVPT *and uniform tree transducers are incomparable.*

6.4.5 Top-Down Tree Transducers

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). In this section we investigate the expressive power of top-down binary tree transducers that run on fcns encodings of unranked trees.

Ranked Trees 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 :

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

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 [CDG+07]. 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 fcns : $\mathcal{H}_\Sigma \to \mathcal{T}^2_\Sigma$ such that:

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

Example 6.4.14 (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. 1. 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:*

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

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

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

Example 6.4.15 (Hedge). *The hedge*

$$a_1 a_2 \dots a_n$$

is encoded as

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

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.*

Top-down binary tree transducers. Let Σ be an alphabet with constant and binary symbols only. A top-down binary tree transducer (TDTT) over Σ [CDG+07] 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 $\rho : \{x_1, x_2\} \to \mathcal{T}^r_\Sigma$ 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 6.4.16. *Let T be a* TDTT*. There exists $k \in \mathbb{N}$, such that for all $(t, t') \in R(T)$, we have $h(t') < kh(t)$.*

TDTT can define unranked tree transductions as follows. Let T be a TDTT on $\Sigma \cup \{\perp\}$, it defines the unranked tree transduction $T_\mathcal{U}$ if :

$$R(T) \circ \text{fcns} = \text{fcns} \circ T_\mathcal{U}$$

Example 6.4.17 (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 6.4.16 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 6.4.18. TDTT *and* wnVPT *are incomparable.*

Relation to UTT 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.

A UTT $T = (Q, I, \delta)$ is *non-duplicating*, resp. *order-preserving*, if for all rules $q(f(h_1)h_2)) \xrightarrow{r} \in \delta$, neither h_1 nor h_2 occur more than one time in r, resp. h_2 never occurs before h_1 in r in a depth-first left-to-right order.

In fact non-duplicating and order-preserving UTT are still more expressive than wnVPTs. One can show that wnVPT corresponds to 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. 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.

6.4.6 Macro Visibly Pushdown Transducers

The extra expressiveness of tree transducers comes from the ability to duplicate or move arbitrarily large part of the document. Let us show how one could extend the expressivity of wnVPT following the ideas used in macro tree and forest transducers. In this section we outline how one would extend wnVPT with parameters (*i.e.* macro). However, we let for future works the study of these objects.

Intuitively macro visibly pushdown transducers are wnVPT equipped with parameters. MVPT process a stream as wnVPT do, (i.e. token by token, pushing a stack symbol on the stack when reading a call and popping on return symbols), and in addition: (*i*) MVPT states can receive parameters, these parameters are output fragment that can be concatenated to the current output or pass by to the next state (as parameter), (*ii*) on call transitions MVPT can launch a copy of the transducer on the current subword, the result of this processing (i.e. an output fragment) is passed by as parameter to the next state. The main point is that all copies process the input (independently) in parallel (they all process the same input token at the same moment). When a copy is launched it starts on the next token, i.e. on the same token as the current stream (and as all other active copies).

Let Y be a finite set of parameters $\{y_1, \dots, y_k\}$. For all $i \leq k$, we denote by Y_i the set $\{y_1, \dots, y_i\}$. We define $W(Y)$ as the set of words on $\Sigma \cup Y$. For any ranked alphabet Ω, we denote by $T_\Omega(Y)$ the set of terms over Ω with variables in Y (we exclude the terms y for all $y \in Y$).

Definition 6.4.19 (MVPT). A *macro visibly pushdown transducer* on finite words over Σ is a tuple $T = (Q, Q_0, Q_f, \Gamma, \delta)$ where $Q = \biguplus_{i=0}^{k} Q^{(i)}$ is a finite set of ranked states, $Q_0 \subseteq Q^{(0)}$, respectively $Q_f \subseteq Q$, the set of initial states, respectively final states, Γ the stack alphabet, $\delta = \delta_c \uplus \delta_r$, with $\delta_c \subseteq Q \times \Sigma_c \times W(Y) \times \Gamma \times T_Q(Y)$, $\delta_r \subseteq Q \times \Sigma_r \times W(Y) \times \Gamma \times T_Q(Y)$. Moreover, for all $(q, \alpha, w, \gamma, \xi) \in \delta$, we require that $w \in W(Y_{ar(q)})$, and $\xi \in T_Q(Y_{ar(q)})$.

Any state $q \in Q^{(k)}$ is interpreted as a function $[\![q]\!] : \Sigma^* \times (2^{\Sigma^*})^k \times \Gamma^* \to 2^{\Sigma^*}$. Intuitively, it takes the rest of the stream, the values of the k parameters, and the stack content as arguments, and outputs a set of words. As an example, suppose $p, p' \in Q^{(1)}$ and $p(y_1) \xrightarrow{\langle a\rangle/\langle b\rangle y_1 y_1 \langle /b\rangle, \gamma} p'(p(y_1))$. In the rhs, the root state p', (intuitively the next state of the transition) evaluates the current stream (i.e at the next token), while states occurring as parameter of p' (i.e. p) evaluates the current "subhedge", i.e. the subword that starts just after $\langle a\rangle$ and ends up just before its matching return. Suppose that we are in state p with parameters w, and we now read a word of the form $\langle a\rangle w' \langle /a\rangle w''$. The transducer outputs $\langle b\rangle ww \langle /b\rangle$, and evaluates $w'\langle /a\rangle w''$ with p' which takes one parameter: the result of the evaluation of p on w' with parameter w.

Since we want to give a streaming-oriented semantics of MVPT, we have to take into account the fact that at each moment, we get a new letter. So the MVPT has to decide when the input bound to a parameter is finished or not. This is done by using the stack: when evaluating w', we start with an empty stack. When reading a return on an empty stack, we know that w' has entirely be read (since w' is well-nested). Formally, for all $q \in Q^{(k)}$, all $q' \in Q^{(n)}$, all $\sigma \in \Gamma^*$, and all $S_1, \dots S_k \subseteq \Sigma^*$:

$$[\![q]\!](c.w, S_1, \dots, S_k, \sigma) \quad = \quad \bigcup_{q \xrightarrow{c/o,\gamma} q'(p_1,\dots,p_n) \in \delta_c} \hat{\rho}(o).[\![q']\!](w, [\![p_1]\!]_{\rho,w}, \dots, [\![p_n]\!]_{\rho,w}, \gamma\sigma)$$

$$[\![q]\!](r.w, S_1, \dots, S_k, \sigma) \quad = \quad \left\{ \begin{array}{l} \{\epsilon\} \text{ if } \sigma = \perp \\ \bigcup_{q \xrightarrow{r/o,\gamma} q'(p_1,\dots,p_n) \in \delta_c} \hat{\rho}(o).[\![q']\!](w, [\![p_1]\!]_{\rho,w}, \dots, [\![p_n]\!]_{\rho,w}, \sigma') \text{ if } \sigma = \gamma\sigma' \end{array} \right.$$

$$[\![q]\!](\epsilon, S_1, \dots, S_k, \sigma) \quad = \quad \left\{ \begin{array}{ll} \{\epsilon\} & \text{if } \sigma = \perp \text{ and } q \in F \\ \varnothing & \text{otherwise} \end{array} \right.$$

where $\rho : y_i \mapsto S_i$ is a valuation of the variable $y_1, \dots y_k$, and $\hat{\rho}(o)$ is inductively defined by: $\hat{\rho}(\alpha w) = \{\alpha.w' \mid w' \in \hat{\rho}(w)\}$, for all $\alpha \in \Sigma$, $\hat{\rho}(\epsilon) = \{\epsilon\}$ and $\hat{\rho}(y) = \rho(y)$. The rhs are interpreted as follows:

$$\begin{array}{lcl} [\![q(p_1, \dots, p_n)]\!]_{\rho,w} & = & [\![q]\!](w, [\![p_1]\!]_{\rho,w}, \dots, [\![p_n]\!]_{\rho,w}, \bot) \\ [\![y]\!]_{\rho,w} & = & \rho(y) \end{array}$$

The relation defined by an MVPT T, denoted by $R(T)$, is defined by:

$$\tau_T \; = \; \{(w, w') \mid \exists q \in Q_0^{(k)},\; w' \in [\![q]\!](w, \varnothing, \dots, \varnothing, \bot)\}$$

Let us take an example based on an XML transformation.

The input documents are documents of the following form:

```
<person>
   <name> Toto </name>
   <street> av Louise </street>
   <city> Brussels </city>
   <email> toto@brussels.be </email>
</person>
```

The output documents have the following form:

```
<person>
   <name> Toto </name>
   <address>
      <street> av Louise </street>
      <city> Brussels </city>
   </address>
</person>
```

So the transformation copies the name, encloses the `street` and `city` tags in an `address` tag, and deletes the `email` tag. To implement this with an MVPT we take four states q_1, q_3, q_{addr} of arity 0 and one state q_2 of arity 1. Both q_1 and q_{addr} are final.

$$
\begin{array}{lll@{\qquad}lll}
q_1 & \xrightarrow{\langle person\rangle/\langle person\rangle,\gamma_1} & q_2(q_{addr}) & q_{addr} & \xrightarrow{\langle name\rangle/\epsilon,\gamma_3} & q_{addr} \\
q_2(y) & \xrightarrow{\langle name\rangle/\langle name\rangle,\gamma_2} & q_2(y) & q_{addr} & \xrightarrow{\langle /name\rangle/\epsilon,\gamma_3} & q_{addr} \\
q_2(y) & \xrightarrow{\langle /name\rangle/\langle /name\rangle\langle address\rangle y\langle /addess\rangle,\gamma_2} & q_3 & q_{addr} & \xrightarrow{\langle street\rangle/\langle street\rangle,\gamma_3} & q_{addr} \\
q_3 & \xrightarrow{\langle street\rangle/\epsilon,\gamma_2} & q_3 & q_{addr} & \xrightarrow{\langle /street\rangle/\langle /street\rangle,\gamma_3} & q_{addr} \\
q_3 & \xrightarrow{\langle /street\rangle/\epsilon,\gamma_2} & q_3 & q_{addr} & \xrightarrow{\langle city\rangle/\langle city\rangle,\gamma_3} & q_{addr} \\
q_3 & \xrightarrow{\langle city\rangle/\epsilon,\gamma_2} & q_3 & q_{addr} & \xrightarrow{\langle /city\rangle/\langle /city\rangle,\gamma_3} & q_{addr} \\
q_3 & \xrightarrow{\langle /city\rangle/\epsilon,\gamma_2} & q_3 & q_{addr} & \xrightarrow{\langle email\rangle/\epsilon,\gamma_3} & q_{addr} \\
q_3 & \xrightarrow{\langle email\rangle/\epsilon,\gamma_2} & q_3 & q_{addr} & \xrightarrow{\langle /email\rangle/\epsilon,\gamma_3} & q_{addr} \\
q_3 & \xrightarrow{\langle /email\rangle/\epsilon,\gamma_2} & q_3 & q_3 & \xrightarrow{\langle /person\rangle/\langle /person\rangle,\gamma_1} & q_1
\end{array}
$$

An execution would proceed as follows: start in q_1, read and copy to the output the `person` tag, move to state q_2 and concurrently launch a new execution starting in state q_{addr} (that we describe below) whose resulting output is passed as parameter to state q_2. In state q_2, read and copy to the output the `name` tag, then read and copy the closing `name` tag and concatenate the result of its parameter (result of the procedure q_{addr}) enclosed in an `address` tag, then continue similarly with state q_3. The parallel execution of q_{addr} starts just after the opening tag `person`, and goes until just before the matching closing tag `person`, it ignores all tags except the `street` tag that it copies to the output.

We conjecture that the expressiveness of such transducers is equivalent to that of macro forest transducers. The stream oriented nature of the model might be amenable to an efficient implementation. However these questions are out of the scope of this work, so that we let them for a future work.

6.5 Conclusion

In this chapter we have introduced the class of wnVPT and showed that it is closed under composition and that the type checking against VPA is ExpTime-c. We compare in Table 6.1 and Table 6.2 the properties of this class with the classes of NFT, VPT and NPT.

We then showed the strong connection between wnVPT and tree transducers. And showed that the macro forest and tree transducers are strictly more expressive than wnVPT. For these classes the functionality and k-valuedness problems have not been investigated. To the best of our knowledge, wnVPTs consist in

	T^{-1}	$\overline{T}$	$T_1 \cup T_2$	$T_1 \cap T_2$	$T_1 \circ T_2$
NFT	no	no	yes	no	yes
NPT	no	no	yes	no	no
VPT	no	no	yes	no	no
wnVPT	no	no	yes	no	yes

Table 6.1: Closure under inverse, complement, union, intersection, and composition for NFT, NPT, VPT and wnVPT.

	emptiness / membership	equiv. / inclusion	type checking against VPA	k-valued. (1-val.)/ determinizable
NFT	PTIME	undec	undec	PTIME/ PTIME
NPT	PTIME	undec	undec	undec / undec
NPT	PTIME	undec	undec	- / undec
VPT	PTIME	undec	undec	co-NPTIME (PTIME) / open
wnVPT	PTIME	undec	EXPTIME-C	co-NPTIME (PTIME) / open

Table 6.2: Decision problems for NFT, NPT, VPT, and wnVPT.

the first (non-deterministic) model of unranked tree transformations (that support concatenation of tree sequences) for which k-valuedness and equivalence of functional transformations are known to be decidable.

Functionality and k-valuedness are decidable for finite (ranked) tree transducers [Sei92, Sei94], but we showed that on binary encodings of unranked trees they do not support concatenation and thus that they are incomparable to wnVPTs. Considering finite tree transducers, their ability to duplicate subtrees is the main concern when dealing with k-valuedness. However for wnVPTs, it is more their ability to concatenate sequences of trees which makes this problem difficult.

Chapter 7

Streaming Evaluation

Contents

In this chapter, we present a memory efficient algorithm that performs the transformation defined by visibly pushdown transducers. We then investigate the memory required to perform the transformation and we provide procedures that decide, given a transducer, whether the memory needed can be bounded.

Visibly pushdown transducers operate on words, like for example XML documents, with an implicit nesting structure induced by the structure of the alphabet. According to the underlying execution model, a VPT processes the input words from left to right, which corresponds to a depth-first left-to-right traversal when the nested word is considered as a tree, and uses the stack for dealing with the successive nesting levels of the input. In the setting of XML processing, this corresponds to the streaming model. Streaming is, usually, a memory efficient model as the entire document need not necessarily be stored in memory. For example, validating an XML documents against a schema can be done in streaming and requires a memory proportional to the nesting level of the input document [KM10] (which is usually low [BMV05]). In contrast, one can consider nested words as trees and performs operations on the tree structure directly, for example in a top-down or a bottom-up manner. In XML this corresponds to the document object model (DOM [ALH04]), it often requires the entire document to be read and stored in memory before processing.

However, due to non-determinism, not all transformation defined by VPT can be evaluated efficiently in streaming. For instance, swapping the first and last letter of a word can be defined by a VPT as follows: guess the last letter and transform the first letter into the guessed last letter, keep the value of the first letter in the state, and transform any value in the middle into itself. This transformation requires to keep the entire word in memory until we can verify that the guess was correct.

Our aim is thus to identify classes of transductions that are suitable to space-efficient streaming evaluation. We first consider the requirement that a transducer can be implemented by a program using a *bounded memory* (BM), *i.e.* computing the output word using a memory independent of the size of the input word.

However when dealing with nested words in a streaming setting, the bounded memory requirement is quite restrictive. Indeed, even performing such a basic task as checking that a word is well-nested or checking that a nested word belongs to a regular language of nested words requires a memory dependent on the height (the level of nesting) of the input word [SV02]. This observation leads us to the second question: decide, given a transducer, whether the transduction can be evaluated with a memory that depends only on the size of the transducer and the height of the word (but not on its length). In that case, we say that the transduction is *height bounded memory* (HBM). This is particularly relevant to XML transformations as XML documents can be very long but have usually a small depth [BMV05]. HBM does not specify *how* memory depends on the height. A stronger requirement is thus to consider HBM transductions whose evaluation can be done with a memory that depends *polynomially* on the height of the input word.

Contributions. First, we give a general space-efficient evaluation algorithm, called EVAL, for functional VPTs. After reading a prefix of an input word, the number of configurations of the (non-deterministic) transducer as well as the number of output candidates to be kept in memory may be exponential in the size of the transducer and the height of the input word (but not in its length). Our algorithm produces as output the longest common prefix of all output candidates, and relies on a compact representation of sets of configurations and remaining output candidates (the original output word without the longest common prefix). We prove that it uses a memory polynomial in the size of the transducer, linear in the height of the input word, and linear in the maximal length of a remaining output candidate.

We prove that BM is equivalent to subsequentiability for finite state transducers (NFT), which is known to be decidable in PTIME. BM is however undecidable for arbitrary pushdown transducers but we show that it is decidable in co-NPTIME for VPTs with well-nested domains.

Like BM, HBM is undecidable for arbitrary pushdown transductions. We show that it is decidable in co-NPTIME for transductions defined by VPTs. In particular, we show that the algorithm EVAL runs in HBM iff the VPT satisfies some property, which is an extension of the so called *twinning property* for NFT [Cho77] to nested words. We call it the *horizontal twinning property*, as it only cares about configurations of the transducers with stack contents of identical height. This property only depends on the transduction, *i.e.* is preserved by equivalent transducers.

When a VPT-transduction is height bounded memory, the memory needed may be exponential in the height of the word. We thus refine VPT-transductions to *twinned transductions* for which performing the transformation with our algorithm uses a memory *quadratic* in the height of the input word. This class is characterized by a twinning property that takes the height of the configurations into account. A VPT satisfying this twinning property is called *twinned*. We show, via a non-trivial reduction to the emptiness of pushdown automata with bounded reversal counters, that it is decidable in co-NPTime whether a VPT is twinned.

Finally, the most challenging result of this chapter is to show that being twinned depends only on the transduction and not on the VPT that defines it. Thus, this property indeed defines a class of transductions. As a consequence of this result, all subsequentializable VPTs are twinned, because subsequential VPTs trivially satisfy the twinning property. The class of twinned transductions captures a strictly larger class than subsequentializable VPTs while staying in the same complexity class for evaluation, i.e. polynomial space in the height of the input word when the transducer is fixed.

7.1 Evaluation Algorithm: Eval

In this section, we present the algorithm Eval that performs the transductions defined by fVPTs. For clarity sake and with no loss of generality, we present this algorithm under some assumptions.

Assumptions. First, input words of our algorithms are words $u \in \Sigma^*$ concatenated with a *special end of the word symbol* $\$ \notin \Sigma$.

Second, we only consider input words with *no internal symbols*, as they can easily be encoded by successive call and return symbols.

Third, we assume all VPTs to be *reduced*, *i.e.* every accessible configuration is co-accessible (See Definition 4.4.1). Recall that given any VPT, computing an equivalent reduced VPT can be performed in polynomial time (Theorem 4.4.2).

Finally, input words are assumed to be *valid input*, *i.e.* the algorithm supposes that the input belong to the domain. In practice, Eval outputs the longest common prefix of all possible runs over the input, if the entire input is read but none of these runs is accepting, then the algorithm should raise an error and the produced output should be discarded.

In the next section, we first give an overview of the algorithm Eval. We investigates in depth the algorithm in the following sections.

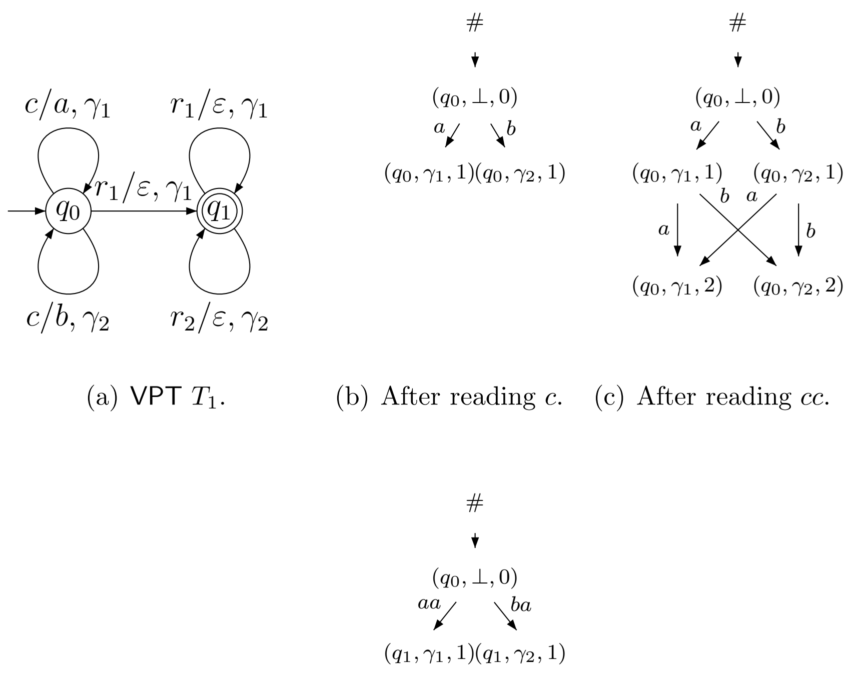
7.1.1 Overview of Eval

Let T be a given fVPT. The algorithm Eval performs the transduction defined by T, that is, on input word $u\$$ it produces as output the word $T(u)$. As T is not necessarily deterministic, the number of possible runs per input u might be unbounded. Computing all these runs and, when the input stream is finished, writing the output of any of the accepting runs (they are all equal as T is functional), is not efficient. The algorithm Eval is an optimization of this naive algorithm, based on three observations.

Equivalent runs. At any time a run of a VPT can be describe by its state, its stack content and the output produced until now. Recall also that on the same input word all runs have the same stack height. We show that, at any time, the number of runs for which the algorithm has to maintain the information (state, stack and output) is bounded by a function of the height of the stack. Indeed, if, at a given time, two runs have the same state and same stack content they must have the same output. In other words, for a given input word u and for every accessible configuration (q, σ) of T, there is at most one v such that $(q_i, \bot) \xrightarrow{u/v} (q, \sigma)$ with $q_i \in I$. For the sake of contradiction, suppose that there exist two runs $(q_i, \bot) \xrightarrow{u/v} (q, \sigma)$ and $(q_i', \bot) \xrightarrow{u/v'} (q, \sigma)$ over the same word u with different outputs $v \neq v'$, as (q, σ) is co-accessible (because T is reduced) there exist u', w with $(q, \sigma) \xrightarrow{u'/w} (q', \sigma')$ for some q' a final state of T. But then $(q_i, \bot) \xrightarrow{uu'/vw} (q', \sigma')$ and $(q_i', \bot) \xrightarrow{uu'/v'w} (q, \sigma)$ are two accepting runs of T over uu' with different outputs. This is a contradiction with the hypothesis that T is functional. Hence, the algorithm needs only to maintain sets of triple (q, σ, w) where q is the current state of the run, σ its corresponding stack content, and w the part of the output that has not been output yet. We call such triple *d-configuration* and write $\mathsf{Dconfs}(T) = Q \times \Gamma^* \times \Sigma^*$ for the set of d-configurations of T.

Compact representation. The number of possible d-configurations the algorithm has to maintain can be exponential in the height of the stack. We show now how to avoid this exponential blow-up.

The set of current d-configurations is stored in a compact structure that shares common stack contents. Consider for instance the VPT T_1 in Fig. 7.1(a). After reading cc, current d-configurations are $\{(q_0, \gamma_1\gamma_1, aa), (q_0, \gamma_1\gamma_2, ab), (q_0, \gamma_2\gamma_1, ba), (q_0, \gamma_2\gamma_2, bb)\}$. Hence after reading c^n,



(a) VPT T_1. (b) After reading c. (c) After reading cc.

(d) After reading ccr_1.

Figure 7.1: Data structure used by Eval.

the number of current d-configurations is 2^n. However, the transition used to update a d-configuration relates the stack symbol and the output word. For instance, the previous set is the set of tuples $(q_0, \eta_1\eta_2, \alpha_1\alpha_2)$ where (η_i, α_i) is either (γ_1, a) or (γ_2, b). Based on this observation, we propose a data structure avoiding this blowup. As illustrated in Fig. 7.1(b) to 7.1(d), this structure is a directed acyclic graph (DAG). Nodes of this DAG are tuples (q, γ, i) where $q \in Q$, $\gamma \in \Gamma$ and $i \in \mathbb{N}$ is the depth of the node in the DAG. Each edge of the DAG is labelled with a word, so that a branch of this DAG, read from the root # to the leaf, represents a d-configuration (q, σ, v): q is the state in the leaf, σ is the concatenation of stack symbols in traversed nodes, and v is the concatenation of words on edges. For instance, in the DAG of Fig. 7.1(c), the branch $\# \rightarrow (q_0, \bot, 0) \xrightarrow{b} (q_0, \gamma_2, 1) \xrightarrow{a} (q_0, \gamma_1, 2)$ encodes the d-configuration $(q_0, \gamma_2\gamma_1, ba)$ of the VPT of Fig. 7.1.(a). We show in Section 7.1.3 that this compact structure yields an exponentially more succinct representation of the set of runs.

Flushing outputs. After reading a prefix u' of a word u, Eval has output the common prefix of all corresponding runs, i.e. $\mathsf{lcp_{in}}(u', T) = \mathsf{lcp}(\mathsf{reach}(u'))$ where $\mathsf{reach}(u') = \{v \mid \exists(q_0, q, \sigma) \in I \times Q \times \Gamma^*,\ (q_0, \bot) \xrightarrow{u'/v} (q, \sigma)\}$. When a new input symbol is read, the DAG is first updated. Then, a bottom-up pass on this DAG computes $\mathsf{lcp_{in}}(u', T)$ in the following way. For each node, let ℓ be the largest common prefix of labels of outgoing edges. Then ℓ is removed from these outgoing edges, and concatenated at the end of labels of incoming edges. At the end, the largest common prefix of all output words on branches is the largest common prefix of words on edges outgoing from the root node $\#$.

Let $\mathsf{out}_{\neq}(u')$ be the maximal size of outputs of T on u' where their common prefix is removed: $\mathsf{out}_{\neq}(u') = \max_{v \in \mathsf{reach}(u')} |v| - |\mathsf{lcp_{in}}(u', T)|$ and $\mathsf{out}^{\max}_{\neq}(u)$ its maximal value over prefixes of u: $\mathsf{out}^{\max}_{\neq}(u) = \max_{u' \text{ prefix of } u} \mathsf{out}_{\neq}(u')$. In the next sections, we prove the following complexity results for Eval:

Proposition 7.1.1. *Let T be an* fVPT*, and $u \in \Sigma^*$. The space used by* Eval *for computing $T(u)$ is in $O(|Q|^2 \cdot |\Gamma|^2 \cdot (h(u)+1) \cdot \mathsf{out}^{max}_{\neq}(u))$, and processing each symbol of u is in time polynomial in $|Q|$, $|\Gamma|$, $|\delta|$, $h(u)$, $\mathsf{out}^{max}_{\neq}(u)$ and $|\Sigma|$.*

All over this section we assume an implementation of VPTs such that the set S of transitions with a given left-hand side can be retrieved in time $O(|S|)$. We define the current height of a prefix of a nested word in the following way: $hc(u) = 0$ if u is well-nested, and $hc(ucv) = hc(u) + 1$ if $c \in \Sigma_c$ and v is well-nested.

7.1.2 Algorithm Naive

We start with the algorithm Naive, that we subsequently improve to obtain Eval. The algorithm Naive simply computes all the runs (with their respective outputs) of the fVPT T on the input word u, stores them in a data structure and, at the end of u, outputs the only output word: it will be the same in all accepting runs, as T is functional.

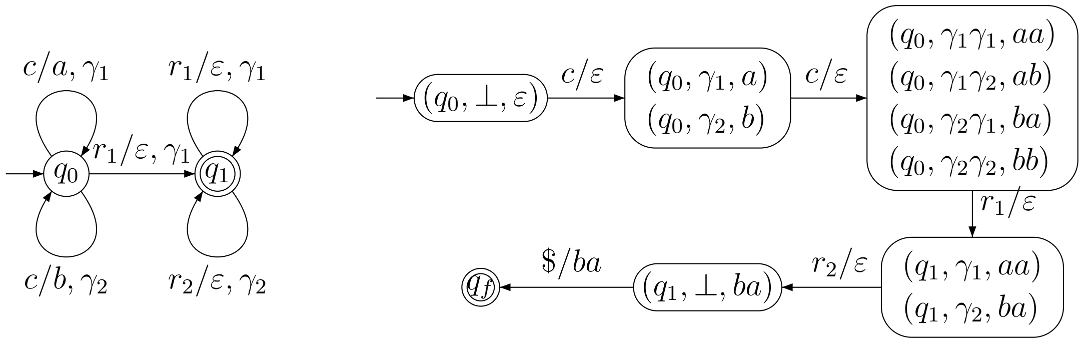
Naive consists in maintaining the set of d-configurations corresponding to the runs of T on the input word u. Hence, it is based on the operation $\mathsf{update}(C, a)$ that returns the set of d-configurations obtained after applying rules of T using input symbol a to each d-configuration of C. The function $\mathsf{update} : \mathsf{Dconfs}(T) \times \Sigma \to \mathsf{Dconfs}(T)$ maps a set of d-configurations and an input symbol to another set of d-configurations. For call symbols $c \in \Sigma_c$,

$$\mathsf{update}(C, c) = \bigcup_{(q,\sigma,v)\in C} \{(q', \sigma\gamma, vv') \mid q \xrightarrow{c/v',\gamma} q'\}$$

and for return symbols $r \in \Sigma_r$,

$$\mathsf{update}(C, r) = \bigcup_{(q,\sigma\gamma,v)\in C} \{(q', \sigma, vv') \mid q \xrightarrow{r/v',-\gamma} q'\}$$

The function update can be considered as the transition function of a transition system with states $\mathsf{Dconfs}(T)$ (i.e. an infinite number of states). We can easily turn it into an infinite state transducer, i.e. an NFT with infinitely many states: this transducer returns ϵ at every input symbol, except for the last one \$, where it returns the output word. This is illustrated in Fig. 7.2. Formally, an


$c/a, \gamma_1$
$r_1/\varepsilon, \gamma_1$
$r_1/\varepsilon, \gamma_1$
q_0
q_1
$c/b, \gamma_2$
$r_2/\varepsilon, \gamma_2$
$(q_0, \bot, \varepsilon)$
c/ε
(q_0, γ_1, a)
(q_0, γ_2, b)
c/ε
$(q_0, \gamma_1\gamma_1, aa)$
$(q_0, \gamma_1\gamma_2, ab)$
$(q_0, \gamma_2\gamma_1, ba)$
$(q_0, \gamma_2\gamma_2, bb)$
r_1/ε
(q_1, γ_1, aa)
(q_1, γ_2, ba)
r_2/ε
$(q_1, \bot, ba)$
$\$/ba$
q_f

(a) A VPT T_1.

(b) Part of the IST t corresponding to T_1, computed on ccr_1r_2.

Figure 7.2: Illustration of the computations of Naive on an input word.

infinite state transducer t (IST for short) is defined exactly like an NFT, except that its number of states may be infinite (but countable). In particular, the acceptance condition remains the same, so that the transduction $R(t)$ is still a set of pairs of finite words (u, v).

Given the functional VPT $T = (Q, I, F, \Gamma, \delta)$, consider the IST $t = (\mathsf{Dconfs}(T) \uplus \{q_f\}, I_t, \{q_f\}, \delta_t)$ where $I_t = \{\{(q_0, \bot, \epsilon) \mid q_0 \in I\}\}$. To deal with the last symbol \$, we have to characterize the sets of d-configurations reached after reading words in $dom(T)$. T being functional, each of these sets of d-configurations C comes with a single output word v:

$$\mathcal{C} = \left\{ (C, v) \in 2^{\mathsf{Dconfs}}(T) \times \Sigma^* \;\middle|\; \begin{array}{l} \exists q \in F.\ (q, \bot, v) \in C \text{ and} \\ \forall (q', \bot, v') \in C,\ q' \in F \implies v' = v \end{array} \right\}$$

Rules in δ_t are:

$$\begin{array}{ll} C \xrightarrow{a/\epsilon} \mathsf{update}(C,a) & \text{for } C \subseteq \mathsf{Dconfs}(T) \text{ and } a \in \Sigma \\ C \xrightarrow{\$/v} q_f & \text{for } (C,v) \in \mathcal{C} \end{array}$$

Lemma 7.1.2. *$(u,v) \in R(T)$ iff $(u\$,v) \in R(t)$.*

Proof. It can be checked easily by induction on $|u|$ that for every $u \in \Sigma^*$, the current state of t after reading u is $\bigcup_{q_0 \in I}\{(q,\sigma,v) \mid (q_0,\bot) \xrightarrow{u/v} (q,\sigma)\}$. Let us check whether reading the last symbol \$ leads to a correct state. Let $u \in \Sigma^*$. If $u \notin dom(T)$, then there is no run of T on u of the form $(q,\bot) \xrightarrow{u/v} (q',\bot)$ with $q \in I$ and $q' \in F$. Hence, the state C reached by t after reading u, if it exists, is such that $(C,v) \notin \mathcal{C}$ for all $v \in \Sigma^*$, so $u \notin dom(t)$. If $u \in dom(T)$, then the state of t reached after reading u is $C = \bigcup_{q_0 \in I}\{(q,\bot,v) \mid (q_0,\bot) \xrightarrow{u/v} (q,\bot)\}$. As T is functional, there is a unique v for all $(q,\bot,v) \in C$ such that $q \in F$, and such elements of C exist, so that $(C,v) \in \mathcal{C}$, and $(u\$,v) \in R(t)$. □

As the IST t is deterministic, algorithm Naive simply computes the unique run of t on the input word u. Let $\mathsf{out}(u) = \max_{v \in \mathsf{reach}(u)} |v|$, and let $\mathsf{out}^{\max}(u)$ be its maximal value over prefixes: $\mathsf{out}^{\max}(u) = \max_{u' \text{ prefix of } u} \mathsf{out}(u')$.

Proposition 7.1.3. *The maximal amount of memory used by* Naive *for processing $u \in \Sigma^*$ is in $O(|Q| \cdot |\Gamma|^{h(u)} \cdot \mathsf{out}^{max}(u))$. The preprocessing time, and the time used by* Naive *to process each symbol of u are both polynomial in $|Q|$, $|\Gamma|$, $|\delta|$, $h(u)$, $\mathsf{out}^{max}(u)$ and $|\Sigma|$.*

Proof. As T is functional, there cannot be two distinct co-accessible d-configurations (q,σ,v) and (q,σ,v') in C. This remark proves the space complexity. For time complexity, updating a d-configuration is just a research of rules to apply. Each of them will generate a new d-configuration, and we assumed it is retrieved in constant time. □

7.1.3 Algorithm Naive$_{\textbf{compact}}$.

We present the data structure S_u^T representing all d-configurations stored by Naive on the VPT T after reading u, i.e., the state of t reached after reading u. This structure is illustrated in Fig. 7.1. This first improvement avoids the exponential blowup in $h(u)$. The structure S_u^T is a labeled DAG (directed acyclic graph) whose nodes are configurations of T (with an additional root node #, and each node has a depth) and edges are labeled by delays: $\mathit{nodes}(S_u^T) = \{\#\} \uplus Q \times (\Gamma \cup \{\bot\}) \times \mathbb{N}$ and $\mathit{edges}(S_u^T) \subseteq \mathit{nodes}(S_u^T) \times \Sigma^* \times \mathit{nodes}(S_u^T)$. A branch of the DAG, read from the root $\#$ to the leaf, represents a d-configuration (q, σ, v): q is the state in the leaf, σ is the concatenation of stack symbols in traversed nodes, and v is the concatenation of words on edges.

The structure S_u^T is defined inductively on u according to the following algorithms.

Initialization. The edges relation, denoted by $\hookrightarrow$, is initialized as followed:

$$\mathit{edges}(S_\epsilon^T) = \{\# \overset{\epsilon}{\hookrightarrow} (q, \bot, 0) \mid q \in I\}$$

Call transition. When a call letter $c \in \Sigma_c$ is read, the structure S_u^T is updated such that, for every leaf of S_u^T, a child is added for every way of updating the corresponding configuration according to a rule of T. If a leaf cannot be updated, it is removed, and also the possible new generated leaves (procedure REMOVE_EDGES). Algorithm 1 describes how S_{uc}^T is computed from S_u^T.

Algorithm 1 Updating structure S with a call symbol.

```
procedure UPDATE_CALL($S, c$)
    $\mathit{newEdges} \leftarrow \emptyset$
    $\mathit{orphans} \leftarrow \emptyset$
    for $(q, \gamma, i) \in \mathit{leaves}(S)$ do
        if $\exists v, \gamma', q' \mid q \xrightarrow{c/v,+\gamma'} q'$ then
            for $(v, \gamma', q') \mid q \xrightarrow{c/v,+\gamma'} q'$ do
                $\mathit{newEdges.add}((q, \gamma, i) \overset{v}{\hookrightarrow} (q', \gamma', i+1))$
        else
            $\mathit{orphans.add}((q, \gamma, i))$
    $\mathit{edges}(S) \leftarrow \mathit{edges}(S) \cup \mathit{newEdges}$

procedure REMOVE_EDGES($S, \mathit{orphans}$)
    while $\mathit{orphans} \neq \emptyset$ do
        $n \leftarrow \mathit{orphans.pop}()$
        for $m \mid \exists v,\ m \overset{v}{\hookrightarrow} n$ do
            $\mathit{remove}(S, m \overset{v}{\hookrightarrow} n)$
            if $\nexists n', v',\ m \overset{v'}{\hookrightarrow} n'$ then $\mathit{orphans.add}(m)$
```

Return transition. For a return letter $r \in \Sigma_r$, we try to pop every leaf: if it is possible, the leaf is removed and the new leaves updated, otherwise we remove the leaf and propagate the removal upwards (procedure REMOVE_EDGES). This is described in Algorithm 2. Only edges and reachable nodes need to be stored,

Algorithm 2 Updating structure S with a return symbol.

```
procedure UPDATE_RETURN($S, r$)
    $newEdges \leftarrow \emptyset$
    $orphans \leftarrow \emptyset$
    for $(q_\ell, \gamma_\ell, i) \in leaves(S)$ do
        if $\exists v, q \mid q_\ell \xrightarrow{r/v,-\gamma_\ell} q$ then
            for $(v, q) \mid q_\ell \xrightarrow{r/v,-\gamma_\ell} q$ do
                for $(q_0, \gamma_0, v_0) \mid (q_0, \gamma_0, i-1) \overset{v_0}{\hookrightarrow} (q_\ell, \gamma_\ell, i) \in edges(S)$ do
                    for $(n, v_1) \mid n \overset{v_1}{\hookrightarrow} (q_0, \gamma_0, i-1) \in edges(S)$ do
                        $newEdges.add(n \overset{v_1 v_0 v}{\hookrightarrow} (q, \gamma_0, i-1))$
        else
            $orphans.add((q_\ell, \gamma_\ell, i))$
    $remove\_edges(S, orphans)$
    $remove\_leaves(S)$
    $edges(S) \leftarrow edges(S) \cup newEdges$

procedure REMOVE_LEAVES($S$)
    for $n \in leaves(S)$ do
        for $(m, v) \mid m \overset{v}{\hookrightarrow} n \in edges(S)$ do
            $remove(S, m \overset{v}{\hookrightarrow} n)$
```

so that $|S_u^T| \leq (hc(u)+1) \cdot |Q|^2 \cdot |\Gamma|^2 \cdot \mathsf{out}(u)$.

We prove the correctness of this construction using the transition function $\Rightarrow_u$ based on edges of S_u^T, that gathers the stack content and delay. The relation $\Rightarrow_u$ is the smallest relation in $(Q \times \Gamma^* \times \mathbb{N}) \times \Sigma^* \times (Q \times \Gamma^* \times \mathbb{N})$ containing $\hookrightarrow$ such that: if $(q_0, \sigma_0, i) \overset{v}{\Rightarrow}_u (q_1, \sigma_1\gamma, j)$ and $(q_1, \gamma, j) \overset{v'}{\hookrightarrow} (q_2, \gamma', j+1)$ then $(q_0, \sigma_0, i) \overset{vv'}{\Longrightarrow}_u (q_2, \sigma_1\gamma\gamma', j+1)$ (we may have $\sigma = \epsilon$ and $\gamma = \perp$). The set of d-configurations stored in S_u^T is defined by: $C(S_u^T) = \{(q, \sigma, v) \mid \exists i,\ (q, \sigma, i) \in leaves(S_u^T)$ and $\# \overset{v}{\Rightarrow}_u (q, \sigma, i)\}$. The following lemma shows that S_u^T exactly encodes the d-configurations computed by the IST t.

Lemma 7.1.4. *$C(S_u^T)$ is the state of the IST t after reading u.*

Proof. As mentioned in the proof of Lemma 7.1.2, the current state of t after reading u is $\bigcup_{q_i \in I}\{(q,\sigma,v) \mid (q_i,\perp) \xrightarrow{u/v} (q,\sigma)\}$. Hence proving the following invariant is sufficient to prove this lemma:

> For every $0 \leq i \leq hc(u)$, $\# \overset{v}{\Rightarrow}_u (q,\sigma,i)$ iff there exists $q_0 \in I$ such that $(q_0,\perp) \xrightarrow{u_1\cdots u_k/v} (q,\sigma)$ where $k = \max\{j \mid hc(u_1\cdots u_j) = i\}$.

Intuitively, this invariant states that every triple (q,σ,i) in the DAG structure S_u^T corresponds to a run on the longest prefix u' of u with a current height $hc(u')$ of i, and therefore, shorter prefix u'' of u of current height i have no associated triple.

We prove it by induction on $|u|$. If $|u| = 0$, then $i = 0$ and the equivalence holds, as we can assume ϵ-loops (without output) on initial states.

Call transition. Assume that the property holds for a given u, we prove that it also holds for the well-nested prefix uc. Let *orphans* be the set of leaves collected by the outermost for-loop of UPDATE_CALL. These are the leaves $(q,\gamma,hc(u))$ of S_u^T such that no rule (q,c,γ',q') exists in δ. Hence, corresponding configurations are blocked, and can be removed. The procedure REMOVE_EDGES propagates these deletions, so that after the call to this procedure, the structure exactly contains configurations that can be updated by c. Hence, by induction hypothesis, the equivalence holds for $0 \leq i \leq hc(u)$. For $i = hc(uc) = hc(u)+1$, let $k = \max\{j \mid hc(u_1\cdots u_j) = i\}$. We have:

$$\begin{array}{llr}
\multicolumn{2}{l}{\# \overset{vv'}{\Longrightarrow}_{uc} (q,\sigma\gamma\gamma',i)} & \\
\text{iff} & \exists q_1,\ \# \overset{v}{\Rightarrow}_{uc} (q_1,\sigma\gamma,i-1) \quad \text{and} \quad (q_1,\gamma,i-1) \overset{v'}{\hookrightarrow} (q,\gamma',i) & (1)\\
\text{iff} & \exists q_1,\ \exists q_i \in I,\ (q_i,\perp) \xrightarrow{u/v} (q_1,\sigma\gamma) \quad \text{and} \quad (q_1,\gamma,i-1) \overset{v'}{\hookrightarrow} (q,\gamma',i) & (2)\\
\text{iff} & \exists q_1,\ \exists q_i \in I,\ (q_i,\perp) \xrightarrow{u/v} (q_1,\sigma\gamma) \quad \text{and} \quad q_1 \xrightarrow{c/v',+\gamma'} q & (3)\\
\text{iff} & \exists q_i \in I,\ (q_i,\perp) \xrightarrow{uc/vv'} (q,\sigma\gamma\gamma') &
\end{array}$$

(1) is by definition of $\Rightarrow_{uc}$. (2) holds because, as mentioned above, REMOVE_EDGES removes non-accessible configurations, and by induction hypothesis. Here, we also have $k-1 = \max\{j \mid hc(u_1\cdots u_j) = i-1\}$, as uc ends with a call symbol: so $u_1\cdots u_{k-1} = u$ and $u_1\cdots u_k = uc$. (3) is due to the way UPDATE_CALL operates: it adds children to leaves according to rules of δ.

Return transition. Now we show that the property holds for the well-nested prefix ur, if it holds for u. Let $h = hc(u)$. Procedure UPDATE_RETURN checks, for each leaf, whether a rule can be applied. If not, the leaf is removed, and orphaned edges too, as explained for call symbols. Then, the hth level is removed and the $(h-1)$th updated, according to rules of T. Hence the property remains true for $0 \leq i \leq h-2$. We have:

$$\begin{aligned}
&\# \overset{v}{\Rightarrow}_{ur} (q, \sigma\gamma\gamma_0, h-1) \\
&\text{iff} \quad \exists q_0, q_1, q_\ell, \gamma_\ell, v', v_0, v_1,\ \# \overset{v'}{\Rightarrow}_u (q_1, \sigma\gamma, h-2) \text{ and} \\
&\qquad (q_1, \gamma, h-2) \overset{v_1}{\hookrightarrow} (q_0, \gamma_0, h-1) \in edges(S_u^T) \text{ and} \\
&\qquad (q_0, \gamma_0, h-1) \overset{v_0}{\hookrightarrow} (q_\ell, \gamma_\ell, h) \in edges(S_u^T) \text{ and} \\
&\qquad q_\ell \xrightarrow{r/v'', -\gamma_\ell} q \text{ and } v = v'v_1v_0v'' && (1) \\
&\text{iff} \quad \exists q_0, q_\ell, \gamma_\ell, v', v_0,\ \# \overset{v'}{\Rightarrow}_u (q_0, \sigma\gamma\gamma_0, h-1) \text{ and} \\
&\qquad (q_0, \gamma_0, h-1) \overset{v_0}{\hookrightarrow} (q_\ell, \gamma_\ell, h) \in edges(S_u^T) \text{ and} \\
&\qquad q_\ell \xrightarrow{r/v'', -\gamma_\ell} q \text{ and } v = v'v_0v'' && (2) \\
&\text{iff} \quad \exists q_\ell, \gamma_\ell, v',\ \# \overset{v'}{\Rightarrow}_u (q_\ell, \sigma\gamma\gamma_0\gamma_\ell, h) \in edges(S_u^T) \text{ and} \\
&\qquad q_\ell \xrightarrow{r/v'', -\gamma_\ell} q \text{ and } v = v'v'' && (3) \\
&\text{iff} \quad \exists q_\ell, \gamma_\ell, v', q_i \in I\ (q_i, \bot) \xrightarrow{u/v'} (q_\ell, \sigma\gamma\gamma_0\gamma_\ell) \text{ and} \\
&\qquad q_\ell \xrightarrow{r/v'', -\gamma_\ell} q \text{ and } v = v'v'' && (4) \\
&\text{iff} \quad \exists q_i \in I,\ (q_i, \bot) \xrightarrow{ur/v} (q, \sigma\gamma\gamma_0)
\end{aligned}$$

Equivalence (1) reflects how UPDATE_RETURN generates the new leaves. (2) and (3) come from the definition of $\Rightarrow_u$. (4) is obtained by induction hypothesis and the fact that, if $k = \max\{j \mid hc(u_1 \cdots u_j) = h\}$, then $u_1 \cdots u_k = u$. □

The depth of the DAG obtained after reading u is the current height of u plus 1, each level has at most $|Q| \cdot |\Gamma|$ nodes, and each edge is labelled with a word of length less than $\mathsf{out}(u)$.

Proposition 7.1.5. *The maximal amount of memory used by* NAIVE$_{\text{COMPACT}}$ *on* $u \in \Sigma^*$ *is in* $O(|Q|^2 \cdot |\Gamma|^2 \cdot (h(u)+1) \cdot \mathsf{out}^{max}(u))$. *The preprocessing time, and the time used by* NAIVE *to process each symbol of* u *are both polynomial in* $|Q|$, $|\Gamma|$, $|\delta|$, $h(u)$, $\mathsf{out}^{max}(u)$ *and* $|\Sigma|$.

7.1.4 Algorithm Eval

The NAIVE and NAIVE$_{\text{COMPACT}}$ algorithms do not produce any output until the end of the stream is reached. Therefore, the longer the output word is, the larger the memory needed is. But, if all runs have a common prefix, this common output can just be flushed, freeing some memory. If, by chance, no runs is accepting, the algorithm should raise an error and the output produced up to now must be discarded. The Eval algorithm improves Naive$_{\text{COMPACT}}$ by outputting at each transition the longest common prefix of all runs. A second improvement is an optimization of the DAG structure with a factorization of the output along the edges.

We extend the definition of the largest common prefix to sets of d-configurations: if $C \subseteq \mathsf{Dconfs}(T)$, then $\mathsf{lcp}(C) = \mathsf{lcp}(\{v \mid (q, \sigma, v) \in C\})$. Let $\mathsf{rem\_lcp}$ be the function that removes the largest common prefix to a set of d-configurations: $\mathsf{rem\_lcp}(C) = \{(q, \sigma, v') \in \mathsf{Dconfs}(T) \mid (q, \sigma, \mathsf{lcp}(C) \cdot v') \in C\}$. From an fVPT T, we define the IST $\tau = (\mathsf{Dconfs}(T) \uplus \{q_f\}, I_\tau, \{q_f\}, \delta_\tau)$ where $I_\tau = \{\{(q_0, \bot, \epsilon) \mid q_0 \in I\}\}$. We keep the same definition of $\mathcal{C}$ as in Naive, and rules of δ_τ are:

$$\begin{array}{ll} C \xrightarrow{a/\mathsf{lcp}(\mathsf{update}(C,a))} \mathsf{rem\_lcp}(\mathsf{update}(C, a)) & \text{for } C \in \mathsf{Dconfs}(T) \text{ and } a \in \Sigma \\ C \xrightarrow{\$/v} q_f & \text{for } (C, v) \in \mathcal{C} \end{array}$$

We start by proving the correctness of the definition of the IST τ. This definition is illustrated in Fig. 7.4.

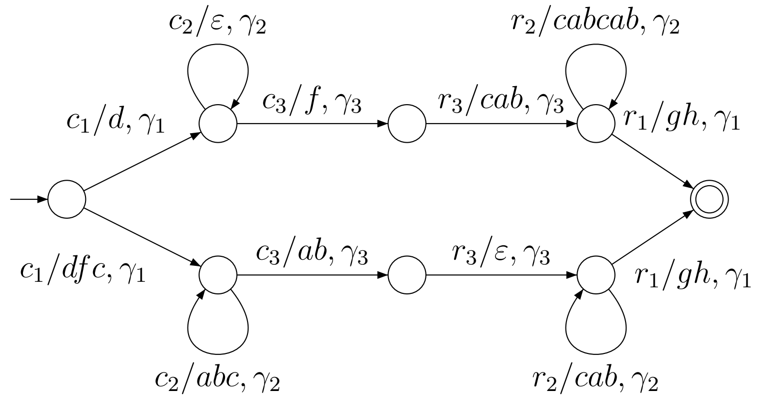


Figure 7.3: A functional VPT on $\Sigma_c = \{c_1, c_2, c_3\}$ and $\Sigma_r = \{r_1, r_2, r_3\}$.

Lemma 7.1.6. *$(u, v) \in R(T)$ iff $(u\$, v) \in R(\tau)$.*

Proof. By an induction on $|u|$, it can be checked that $C_0 \xrightarrow{u/\mathsf{lcp_{in}}(u,T)}_\tau C$ where C_0 is the only element in I_τ and C is obtained from the run of t on u: $C = \mathsf{rem\_lcp}(C')$ with $C_0 \xrightarrow{u/\epsilon}_t C'$. The remainder of the proof is similar to the proof of Lemma 7.1.2. □

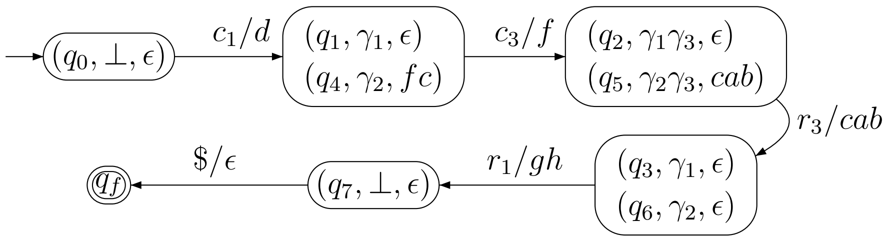
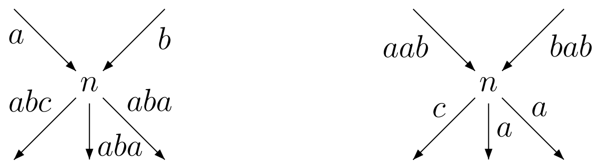

$(q_0, \perp, \epsilon)$
c_1/d
$(q_1, \gamma_1, \epsilon)$
(q_4, γ_2, fc)
c_3/f
$(q_2, \gamma_1\gamma_3, \epsilon)$
$(q_5, \gamma_2\gamma_3, cab)$
r_3/cab
$(q_3, \gamma_1, \epsilon)$
$(q_6, \gamma_2, \epsilon)$
r_1/gh
$(q_7, \perp, \epsilon)$
$\$/\epsilon$
q_f

Figure 7.4: Part of the IST τ corresponding to the VPT in Fig. 7.3, computed by Eval on input $c_1c_3r_3r_1$.



(a) Internal node n of the DAG. (b) Node n after update by *factorize*.

Figure 7.5: Changes performed by *factorize* on a node.

We now provide algorithms for the second step of the computation performed by Eval on an input symbol a. Recall that the first step is the same as in Naive$_{\text{compact}}$, i.e. Algorithm 1 if a is a call symbol, and Algorithm 2 if it is a return symbol, which transforms S_u^T to a new structure S'. The second step is the computation of $\mathsf{lcp}(C(S'))$, which is output, and removed from every branch of S', using Algorithm 3.

Algorithm 3 starts with the procedure *factorize*, that processes every nodes in a bottom-up manner (from leaves to the root #). For every node, the lcp of all outgoing edges is moved to all incoming edges. This is illustrated in Fig. 7.5.

For every node $n \in \mathit{nodes}(S_u^T)$, let S_n be the structure obtained from S_u^T just after returning from $\mathit{factorize}(S, n, \mathit{done})$, and let $\Rightarrow_n$ be the relation defined like $\Rightarrow_u$, but on S_n. Note that the structure has the same set of nodes and edges after being processed by Algorithm 3, only the labels of edges are updated. Let B_n be the set of branches from node n to a leaf: $B_{n_0} = \{(n_0, n_1, \dots, n_k) \mid \forall 0 \leq i < k,\ \exists v_i,\ n_i \overset{v_i}{\hookrightarrow} n_{i+1} \text{ and } n_k \in \mathit{leaves}(S_u^T)\}$. For a branch b, we write $V_n(b)$ for its output in the structure S_n, and $V(b)$ for its output in S_u^T: $V_n((n_0, \dots, n_k)) = v_0 \cdots v_{k-1}$ where $n_i \overset{v_i}{\hookrightarrow} n_{i+1}$ for all $0 \leq i < k$ in S_n. We extend this definition to sets of branches: $V_n(B) = \{V_n(b) \mid b \in B\}$. Note also that each node is

Algorithm 3 Compute $\mathsf{lcp}(C(S))$, and update S to the structure encoding $\mathsf{rem\_lcp}(C(S))$.

```
procedure OUTPUT_LCP($S$)
    $factorize(S, \#, \emptyset)$
    $\ell \leftarrow \mathsf{lcp}(\{v \mid \exists n,\ \# \overset{v}{\hookrightarrow} n\})$
    output $\ell$
    for $m, v \mid \# \overset{v}{\hookrightarrow} m$ do
        let $p$ be such that $v = \ell \cdot p$
        replace $\# \overset{v}{\hookrightarrow} m$ by $\# \overset{p}{\hookrightarrow} m$ in $S$

function FACTORIZE($S$, $n$, $done$)
    if $n \notin leaves(S)$ then
        for $m, v \mid n \overset{v}{\hookrightarrow} m$ and $m \notin done$ do
            $done \leftarrow factorize(S, m, done)$
        if $n = \#$ then return $done \cup \{n\}$
        $factor \leftarrow \mathsf{lcp}(\{v \mid \exists m,\ n \overset{v}{\hookrightarrow} m\})$
        for $m, v \mid n \overset{v}{\hookrightarrow} m$ do
            let $p$ be such that $v = factor \cdot p$
            replace $n \overset{v}{\hookrightarrow} m$ by $n \overset{p}{\hookrightarrow} m$ in $S$
        for $p, v \mid p \overset{v}{\hookrightarrow} n$ do
            replace $p \overset{v}{\hookrightarrow} n$ by $p \xhookrightarrow{v \cdot factor} n$ in $S$
    return $done \cup \{n\}$
```

processed once using *factorize*, and in a bottom-up way: when processing node n, all its descendants have been updated before (cf lines 11 and 12). The following property is the main invariant proving the correctness of Algorithm 3.

Lemma 7.1.7. *For every node $n \neq \#$, let $\ell = \mathsf{lcp}(V(B_n))$. Then,*

1. *for every branch $b \in B_n$, $V(b) = \ell \cdot V_n(b)$*
2. *for every p, v such that $p \overset{v}{\hookrightarrow} n$ in S_n, $v = v'\ell$ with $p \overset{v'}{\hookrightarrow} n$ in S_u^T*

Proof. We prove the following property by bottom-up induction on the structure. This property is true on leaves, as *factorize* does not modify their incoming edges. Assume that the property holds for all descendants of a node n. Let $\ell = \mathsf{lcp}(V(B_n))$, and consider a branch $b \in B_n$. The function *factorize* applied at n computes the lcp of edges outgoing from n and removes it on every branch. Hence, if n' be the node processed by *factorize* before n, then:

$$V_{n'}(b) = \mathsf{lcp}(\{v \mid \exists p,\ n \overset{v}{\hookrightarrow} p \text{ in } S_{n'}\}) \cdot V_n(b) \tag{7.1}$$

Let p be the second node in branch $b = (n, p, \ldots)$. As p and all its descendants are processed before n' and not modified until the call of *factorize* on p, we have $V_{n'}(b) = V_p(b)$. Let us decompose $V_p(b)$ according to p: $V_p(b) = v_0 \cdot V_p(b_p)$ where $n \overset{v_0}{\hookrightarrow} p$ in S_p and b_p is the branch obtained from b by removing n. Using the induction hypothesis applied at p, we get $v_0 = v_1\ell'$ with $\ell' = \mathsf{lcp}(V(B_p))$, $n \overset{v_1}{\hookrightarrow} p$ in S_u^T, and $V(b_p) = \ell' \cdot V_p(b_p)$. Thus $V_p(b) = v_1 \cdot \ell' \cdot V_p(b_p) = v_1 \cdot V(b_p) = V(b)$. Equation (7.1) becomes:

$$V(b) = \mathsf{lcp}(\{v \mid \exists p,\ n \overset{v}{\hookrightarrow} p \text{ in } S_{n'}\}) \cdot V_n(b) \tag{7.2}$$

Let us write $\ell'' = \mathsf{lcp}(\{v \mid \exists p,\ n \overset{v}{\hookrightarrow} p \text{ in } S_{n'}\})$. It remains to prove that $\ell'' = \ell$. Notice that this will prove both parts of the lemma. We have $\ell'' = \mathsf{lcp}(\{v \mid \exists p,\ n \overset{v}{\hookrightarrow} p \text{ in } S_p\})$ because p and its descendants are unchanged between calls of *factorize* on p and n'. Using the induction hypothesis on each p, we obtain:

$$\ell'' = \mathsf{lcp}(\{v \cdot \mathsf{lcp}(V(B_p)) \mid n \overset{v}{\hookrightarrow} p \text{ in } S_u^T\}) = \mathsf{lcp}(V(B_n)) = \ell$$

This concludes the proof. □

The next lemma ensures that *factorize* preserves the semantic of the structure, i.e. the set of encoded configurations.

Lemma 7.1.8. $C(F(S_u^T)) = C(S_u^T)$.

Proof.

$$\begin{array}{ll}
(q,\sigma,v) \in C(S_u^T) & \\
\quad \text{iff } \exists i,\ (q,\sigma,i) \in \mathit{leaves}(S_u^T) \text{ and } \# \overset{v}{\Rightarrow}_u (q,\sigma,i) & \\
\quad \text{iff } \exists i,\ (q,\sigma,i) \in \mathit{leaves}(S_u^T) \text{ and } \exists p,\ \# \overset{v_0}{\hookrightarrow} p \text{ and } p \overset{v_1}{\Rightarrow}_u (q,\sigma,i) \text{ with } v = v_0 v_1 & \\
\quad \text{iff } \exists i,\ (q,\sigma,i) \in \mathit{leaves}(S_u^T) \text{ and } \exists p,\ \# \overset{v_0}{\hookrightarrow} p \text{ and } v = v_0 V(b) & \\
\qquad \text{where } b \text{ is a branch } p \Rightarrow_u (q,\sigma,i) & \\
\quad \text{iff } \exists i,\ (q,\sigma,i) \in \mathit{leaves}(S_u^T) \text{ and } \exists p,\ \# \overset{v_0}{\hookrightarrow} p \text{ and } v = v_0 \cdot \mathsf{lcp}(V(B_p)) \cdot V_n(b) & \\
\qquad \text{where } b \text{ is a branch } p \Rightarrow_u (q,\sigma,i) & (1) \\
\quad \text{iff } (q,\sigma,v) \in C(F(S_u^T)) &
\end{array}$$

Equivalence (1) is by Lemma 7.1.7.

□

It is now clear that Proposition 7.1.1 holds. Correctness of Algorithm 3 is ensured by Lemma 7.1.7, and the fact that at the root node it uses the same technique to factorize and output the lcp. In terms of memory requirement, the number of nodes of S_u^T remains bounded as before, while words on edges have length at most $\mathsf{out}_{\neq}^{\max}(u)$, as each of them participate in a d-configuration in $C(S_u^T)$, as proved by Lemma 7.1.8 and Lemma 7.1.4. All procedures are in polynomial time.

7.2 Bounded Memory Transductions

In this section, we consider the class of bounded memory transducers. They are the transductions whose evaluation is in constant memory if we fix the machine that defines the transduction. We show that the bounded memory problem is decidable for NFT and for VPT whose domain is well-nested (*i.e.* all words in the domain are well-nested), while it is undecidable for pushdown transducers.

Turing Transducers In order to define the complexity classes we target, we introduce a *deterministic* computational model for word transductions that we call *Turing Transducers (TT)*. Turing transducers have three tapes: one read-only left-to-right input tape, one write-only left-to-right output tape, and one standard working tape. Such a machine naturally defines a transduction: the input word is initially on the input tape, and the result of the transduction is the word written on the output tape after the machine terminates in an accepting state. We denote by $R(M)$ the transduction defined by M. The space complexity is measured on the working tape only.

We can now define transductions that can be evaluated with a constant amount of memory.

Definition 7.2.1. *A (functional) transduction $R \subseteq \Sigma^* \times \Sigma^*$ is* bounded memory (BM) *if there exists a Turing transducer M and $K \in \mathbb{N}$ such that $R(M) = R$ and on any input word $u \in \Sigma^*$, M runs in space complexity at most K.*

7.2.1 Finite State Transductions

We show that for finite state transductions, bounded memory is characterized by sub-sequential transductions (See Definition 3.1.23). This implies the decidability of BM for NFT as it is decidable in PTime whether a NFT defines a sub-sequential transductions (See Theorem 3.1.28).

Proposition 7.2.2. *Let T be a functional* NFT*.*

1. *$R(T)$ is BM iff T is subsequentializable;*
2. *It is decidable in* PTime *if $R(T)$ is BM [WK95].*

Proof. Statement 2 is proved in [WK95] (it is proved that subsequentializability is decidable in PTime). We prove statement 1. Clearly, if $R(T)$ is definable by a subsequential transducer T_d, then evaluating T_d on any input word u can be done with a space complexity that depends on the size of T_d only.

Conversely, if $R(T)$ is BM, there exists $K \in \mathbb{N}$ and a TT M that transforms any input word u into $R(T)(u)$ in space complexity K. Any word on the working tape of M is of length at most K. As M is deterministic, we can therefore see M as a subsequential NFT, whose states are pairs (q, w) where q is a state of T and w a word on the working tape (modulo some elimination of ϵ-transitions). □

7.2.2 Pushdown Transductions

BM is undecidable for pushdown transducers, since it is as difficult as deciding whether a pushdown automaton defines a regular language. It is also undecidable when we restrict the problem to transductions whose domains are well-nested.

Proposition 7.2.3. *It is undecidable whether a pushdown transducer is BM.*

Proof. We reduce the problem of deciding whether the language of a pushdown automaton P over an alphabet A is regular to BM. Any letter of A is seen as an internal symbol. We associate with P a pushdown transducer I_P which defines the identity on $L(P)$. Clearly, if $L(P)$ is regular, it is defined by a finite automaton which can easily be turned into a Turing transducer defining $R(I_P)$ and which uses a memory that depends on the size of the automaton only. Conversely, if $R(I_P)$ is BM, there exists a function f and a TT M equivalent to I_P and which uses at most $f(1, |M|)$ bits of memory, i.e. an amount of memory which depends on the size of M only. The machine M can easily be turned into a finite automaton which defines $L(P)$, whose states are the configurations of the working tape of M. □

Note that for deterministic pushdown transducers this reduction does not apply. Indeed, Stearns showed that it is decidable whether a deterministic context-free language is regular [Ste67].

7.2.3 Visibly Pushdown Transductions

For visibly pushdown transducers whose domain are well-nested words, BM is quite restrictive as, for instance, it imposes to verify whether a word is well-nested by using a bounded amount of memory. This can be done only if the height of the words of the domain is bounded by some constant which depends on the transducer only.

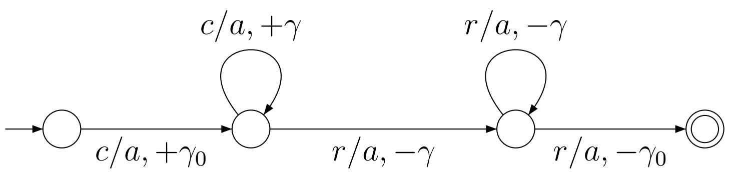


Figure 7.6: A deterministic VPT that is not BM.

Example 7.2.4. *The* VPT *in Figure 7.6 is deterministic and implements the transduction*

$$c^n r^n \to a^{2n} \qquad \textit{for } n \geq 2$$

This transduction is not BM as checking that the input word is well-nested requires a memory dependent on the height of the input word.

Given an integer $k \in \mathbb{N}$ and a VPT T, one can define an NFT, denoted by NFT(T, k), which is the restriction of T to input words of height less than k. The transducer is naturally constructed by taking as states the configurations (q, σ) of T such that $|\sigma| \leq k$.

Proposition 7.2.5. *Let T be a functional* VPT *with n states such that dom$(T) \subseteq L_{wn}(\Sigma)$.*

1. *$R(T)$ is BM iff (i) for all $u \in$ dom(T), $h(u) \leq n^2$, and (ii)* NFT(T, n^2) *is BM;*

2. *It is decidable in co-*NPTIME *whether $R(T)$ is BM.*

Proof. If $R(T)$ is BM, there exist K and a TT M such that M evaluates any input word in space at most K. We can easily extract from M a finite automaton that defines $dom(T)$, whose number of states m only depends on M and K. By a simple pumping argument, it is easy to show that the words in $dom(T)$ have a height bounded by m. If the height of the words in $dom(T)$ is bounded, then their height is bounded by n^2. Indeed, assume that there exists a word $u \in dom(T)$ whose height is strictly larger than n^2. Then there exists a run of T on u of the following form:

$$(i, \perp) \xrightarrow{u_1/v_1} (q, \sigma) \xrightarrow{u_2/v_2} (q, \sigma\sigma') \xrightarrow{u_3/v_3} (p, \sigma\sigma') \xrightarrow{u_4/v_4} (p, \sigma) \xrightarrow{u_5/v_5} (f, \perp)$$

such that $u = u_1u_2u_3u_4u_5$, σ' is not empty, and i (resp. f) is an initial (resp. final) state of T. The existence of this decomposition follows from the consideration of the set of pairs of positions in u corresponding to matching calls and returns of well-nested subwords of u. Clearly, for $n \in \mathbb{N}$, the words $u = u_1u_2^nu_3u_4^nu_5$ are all in $dom(T)$. Thus words with arbitrarily large heights are in $dom(T)$, yielding a contradiction. Therefore $\mathsf{NFT}(T, n^2)$ is equivalent to T. As in the proof of Proposition 7.2.2, we can regard M as a subsequential NFT T_M whose set of states are configurations of the machine. The NFT T_M is equivalent to T, and therefore to $\mathsf{NFT}(T, n^2)$. Since T_M is subsequential, $\mathsf{NFT}(T, n^2)$ is subsequentializable and therefore by Proposition 7.2.2, $\mathsf{NFT}(T, n^2)$ is BM. The converse is obvious.

Therefore to check whether $R(T)$ is BM, we first decide if the height of all input words in the domain of T is less or equal than n^2. This can be done in PTime $O(|T| \cdot n^2)$ by checking emptiness of the underlying automaton of T extended with counters up to n^2+1 that counts the height of the word. Then we check in co-NPTime whether evaluating T can be done in constant memory if we fix both the transducer and the height of the word (Theorem 7.3.12). Since here the height is bounded by n^2, it is equivalent to checking bounded memory. □

In this document, we do not address the more general and difficult question to decide whether a VPT with arbitrary domain is equivalent to some NFT. A necessary condition for a VPT to be NFT definable is that its domain should be regular. This latter question is decidable: one can determinized the (underlying) VPA and then apply the procedure for deciding whether a DPA defines a regular language, this procedure construct an equivalent NFA [Ste67]. However, it is not clear how to use this result to obtain decidability for VPT. Therefore, the question of deciding, given a VPT, whether there exists an equivalent NFT is open.

7.3 Height Bounded Memory Transductions

As we have seen in the previous section, bounded memory is too restrictive to still benefit from the extra expressiveness of VPT compared to NFT. Indeed, asking a VPT to be BM implies that the VPT cannot use its stack (unboundedly). It can therefore not recognize well-nested words of unbounded height.

In this section, we define a notion of bounded memory which is well-suited to VPTs. It asks that the memory needed for performing the transduction is bounded by the height of the input word. As the height of words is defined for well-nested words, we assume that the domain of all transductions in this section are well-nested words.

Definition 7.3.1. *A (functional) transduction $R \subseteq \Sigma^* \times \Sigma^*$ is* height bounded memory (HBM) *if there exists a Turing transducer M and a function $f : \mathbb{N} \to \mathbb{N}$ such that $R(M) = R$ and on any input word $u \in \Sigma^*$, M runs in space at most $f(h(u))$.*

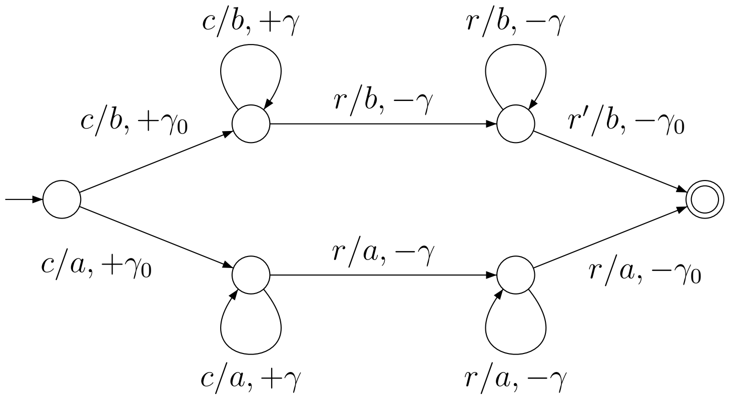


Figure 7.7: A functional VPT HBM but not twinned.

Example 7.3.2. *The VPT of Figure 7.7 implements the functional transduction:*

$$\{c^n r^{n-1} r' \to b^{2n} \mid n \geq 2\} \cup$$

$$\{c^n r^n \to a^{2n} \mid n \geq 2\}$$

While evaluating this transduction, one can not output anything until reaching the last letter r or r'. Therefore the memory required is proportional to the length of the word. However, in this case, the transduction is HBM as the height of the input word is proportional to its length.

Note that the deterministic VPT *of Example 7.6 is also HBM. Indeed, this is true for all deterministic* VPTs*, as the memory used is proportional to the height of the stack, i.e. proportional to the height of the input word.*

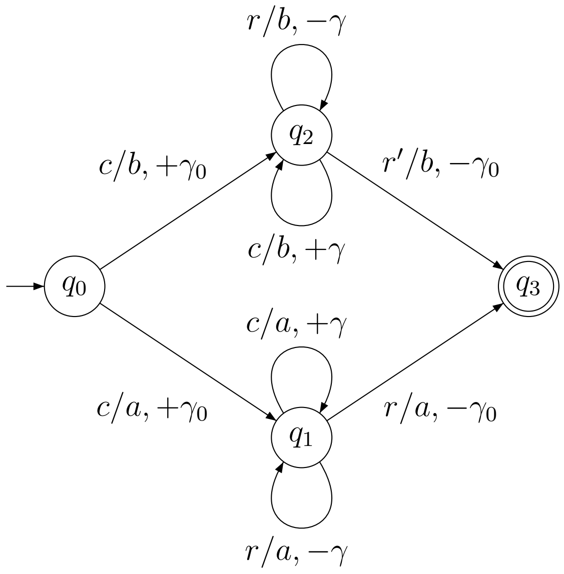


Figure 7.8: A functional VPT not HBM.

Example 7.3.3. *The* VPT *of Figure 7.8 implements the functional transductions:*

$$\{cwr' \rightarrow b^{|w|+2} \mid w \in L_{wn}(\Sigma)\} \cup$$

$$\{cwr \rightarrow a^{|w|+2} \mid w \in L_{wn}(\Sigma)\}$$

While evaluating this transduction, one can not output anything until the last letter r or r'. Therefore the memory needed is proportional to the length of the input word. For $n \geq 0$, the words $c(cr)^n r$ and $c(cr)^n r'$ are in the domain of the transduction and have height 2 but arbitrary length. This transduction is, therefore, not HBM.

Note that this definition ensures that the machine cannot store all the input words on the working tape in general. When the structured alphabet contains only internal letters, HBM and BM coincides, thus it is undecidable whether a pushdown transducer is HBM.

Theorem 7.3.4. *Height bounded memory is undecidable for functional pushdown transducers.*

The remainder of this section is devoted to the proof that HBM is decidable for fVPTs.

7.3.1 Horizontal Twinning Properties

BM functional NFT-transductions (or equivalently subsequentializable NFT) are characterized by the so called *twinning property* (See Theorem 3.1.28), which is decidable in PTime [WK95]. We introduce a similar characterization of HBM fVPTs-transductions, called the *horizontal twinning property* (HTP). The restriction of the horizontal twinning property to NFT is equivalent to the usual twinning property for NFT.

Intuitively, the HTP requires that two runs on the same input cannot have arbitrary large difference between their outputs. More precisely, recall that the delay, $\Delta(u, v)$, between two output words u, v is the pair of words $((u \wedge v)^{-1}u, (u \wedge v)^{-1}v)$, *i.e.* it is the pair (u, v) from which the longest common prefix is removed. The HTP requires that the delay between the outputs of any pair of runs over the same input word cannot be arbitrarily large.

Definition 7.3.5. *Let T be an* fVPT*. T satisfies the* horizontal twinning property *(HTP) if for all $u_1, u_2, v_1, v_2, w_1, w_2 \in \Sigma^*$, for all $q_0, q_0' \in I$, for all $q, q' \in Q$, and for all $\sigma, \sigma' \in \Gamma^*$,*

$$\textit{if} \begin{cases} (q_0, \perp) \xrightarrow{u_1/v_1} (q, \sigma) \xrightarrow{u_2/v_2} (q, \sigma) \\ (q_0', \perp) \xrightarrow{u_1/w_1} (q', \sigma') \xrightarrow{u_2/w_2} (q', \sigma') \end{cases} \quad \textit{then } \Delta(v_1, w_1) = \Delta(v_1 v_2, w_1 w_2).$$

Note that this definition corresponds to the twinning property for NFT in Definition 3.1.25.

Example 7.3.6. *Consider the VPT of Fig. 7.8. It does not satisfy the HTP, as the delay between the upper and the lower branches increases on words of the form $c(cr)^n$. One can check that the VPT of Fig. 7.7 satisfies the HTP. Finally, all deterministic* VPTs *trivially satisfy the HTP, as there cannot be two different runs on the same input.*

7.3.2 HTP is Decidable

We show that it is decidable, given a functional VPT, whether it satisfies the HTP or not. We construct a reversal bounded counter automaton (See Definition 5.5.2), that accepts some, and only, witnesses (but not necessarily all) of non-satisfiability of the HTP. Furthermore, if the given VPT does not satisfy the HTP the automaton accepts at least one witness. Therefore, the VPT satisfies the HTP if and only if the language of the reversal bounded counter automaton is empty. This latter problem is decidable in co-NPTime (Theorem 5.5.6).

We start with a lemma that allow us to simplify the construction of the counter automaton. It states that, to find a witness of non-satisfiability of the HTP, you can, equivalently, look for two outputs of different lengths or a mismatch between outputs, on some specific runs. This condition, which is equivalent to the HTP, is easier two verify with a counter automaton than the HTP.

Lemma 7.3.7. *Let T be an fVPT, T does not satisfy the HTP if and only if there exist $q_0, q_0' \in I$, $q, q' \in Q$, $\sigma, \sigma' \in \Gamma^*$ and $u_1, v_1, w_1, u_2, v_2, w_2 \in \Sigma^*$, with either $v_2 \neq \epsilon$ or $w_2 \neq \epsilon$, and:*

$$\left\{\begin{array}{lllll} (q_0,\bot) & \xrightarrow{u_1/v_1} & (q,\sigma) & \xrightarrow{u_2/v_2} & (q,\sigma) \\ (q_0',\bot) & \xrightarrow{u_1/w_1} & (q',\sigma') & \xrightarrow{u_2/w_2} & (q',\sigma') \end{array}\right. \quad (**)$$

and either (i) $|v_2| \neq |w_2|$ or (ii) $|v_2| = |w_2|$, $|v_1| \leq |w_1|$ and $v_1v_2 \not\preceq w_1w_2$.

Proof. Let suppose that there exist $q_0, q_0' \in I$, $q, q' \in Q$, $\sigma, \sigma' \in \Gamma^*$ and $u_1, v_1, w_1, u_2, v_2, w_2 \in \Sigma^*$, with either $v_2 \neq \epsilon$ or $w_2 \neq \epsilon$ and :

$$(q_0,\bot) \xrightarrow{u_1/v_1} (q,\sigma) \xrightarrow{u_2/v_2} (q,\sigma) \qquad (q_0',\bot) \xrightarrow{u_1/w_1} (q',\sigma') \xrightarrow{u_2/w_2} (q',\sigma')$$

And let show that if (i) or (ii) hold so does $\Delta(v_1, w_1) \neq \Delta(v_1v_2, w_1w_2)$.

First suppose that (i) holds. Let $x = v_1 \wedge w_1$ and $z = v_1v_2 \wedge w_1w_2$, clearly there exists $y \in \Sigma^*$ such that $z = xy$. By definition, we have $\Delta(v_1, w_1) = (x^{-1}v_1, x^{-1}w_1)$ and $\Delta(v_1v_2, w_1w_2) = ((xy)^{-1}v_1v_2, (xy)^{-1}w_1w_2)$. Therefore if (i) holds then either $|x^{-1}v_1| \neq |(xy)^{-1}v_1v_2|$ or $|x^{-1}w_1| \neq |(xy)^{-1}w_1w_2)|$. Indeed, suppose the first inequality does not hold, we have $|x^{-1}v_1| = |v_1| - |x| = |(xy)^{-1}v_1v_2| = |v_1| + |v_2| - |x| - |y|$, that is $0 = |y| - |v_2|$. Therefore one can check that the second inequality must hold. We have shown that (i) implies $\Delta(v_1, w_1) \neq \Delta(v_1v_2, w_1w_2)$.

Now, suppose (ii) holds and let us show that we also have $\Delta(v_1, w_1) \neq \Delta(v_1v_2, w_1w_2)$. We have (ii) $|v_2| = |w_2|$, $|v_1| \leq |w_1|$ and $v_1v_2 \not\preceq w_1w_2$. Note that for any $u, v, x, y \in \Sigma^*$ with $|u| \leq |v|$ and $\Delta(u, v) = (x, y)$ we have $u \not\preceq v$ if and only if $x \neq \epsilon$ and $y \neq \epsilon$. Therefore, if we pose $(y_1, y_2) = \Delta(v_1v_2, w_1w_2)$, we have, by hypothesis, $v_1v_2 \not\preceq w_1w_2$ and so $y_1 \neq \epsilon$ and $y_2 \neq \epsilon$. Let pose $(x_1, x_2) = \Delta(v_1, w_1)$, if $(x_1, x_2) = (y_1, y_2)$ then $x_1 \neq \epsilon$ and $x_2 \neq \epsilon$, that is

$v_1 \not\preceq w_1$, but then $\Delta(v_1v_2, w_1w_2) = \Delta(v_1, w_1) \cdot (v_2, w_2)$ which cannot be equal to $\Delta(v_1, w_1)$ (because, by hypothesis, one of v_2 and w_2 is not the empty word).

For the other direction, let suppose the HTP does not hold and let show that (i) or (ii) is satisfied for some words (as above). So there exist $q_0, q_0' \in I$, $q, q' \in Q$, $\sigma, \sigma' \in \Gamma^*$ and $u_1, v_1, w_1, u_2, v_2, w_2 \in \Sigma^*$ with :

$$(q_0, \bot) \xrightarrow{u_1/v_1} (q, \sigma) \xrightarrow{u_2/v_2} (q, \sigma) \qquad (q_0', \bot) \xrightarrow{u_1/w_1} (q', \sigma') \xrightarrow{u_2/w_2} (q', \sigma')$$

such that $\Delta(v_1, w_1) \neq \Delta(v_1v_2, w_1w_2)$. Moreover, let us suppose, by contradiction, that for all $k \in \mathbb{N}$, if we replace u_2 by ${u_2}^k$ (and thus v_2 and w_2 are replaced by ${v_2}^k$ and ${w_2}^k$ respectively), we get a system such that neither (i) nor (ii) do hold, that is for all $k \in \mathbb{N}$ we have $|{v_2}^k| = |{w_2}^k|$, $|v_1| \leq |w_1|$ and $v_1{v_2}^k \preceq w_1{w_2}^k$. We now prove that this implies $\Delta(v_1, w_1) = \Delta(v_1v_2, w_1w_2)$, which is a contradiction with the hypothesis. On the one hand, if for all k we have $v_1{v_2}^k \preceq w_1{w_2}^k$ then we have $w_1 = v_1{v_2}^a v_2'$ for some $a \in \mathbb{N}$ and some $v_2' \preceq v_2$. So, $\Delta(v_1, w_1) = \Delta(v_1, v_1{v_2}^a v_2') = (\epsilon, {v_2}^a v_2')$. On the other hand, $v_1{v_2}^k \preceq w_1{w_2}^k$ and $|v_2| = |w_2|$ implies that we have $w_1w_2 = v_1{v_2}^{a+1}v_2'$. So we have $\Delta(v_1v_2, w_1w_2) = \Delta(v_1v_2, v_1{v_2}^{a+1}v_2') = (\epsilon, {v_2}^a v_2')$. Therefore $\Delta(v_1, w_1) = \Delta(v_1v_2, w_1w_2)$. This concludes the proof. ☐

Lemma 7.3.8. *The HTP is decidable in co-*NPTIME *for* fVPTs.

Proof. The previous lemma showed that an fVPT T does not satisfy the HTP if and only if there exist $u_1, u_2, v_1, v_2, w_1, w_2 \in \Sigma^*$, $q_0, q_0' \in I$, $q, q' \in Q$, and $\sigma, \sigma' \in \Gamma^*$ that satisfy (**), and such that either we have (i) $|v_2| \neq |w_2|$, or (ii) $|v_2| = |w_2|$, $|v_1| \leq |w_1|$ and not $v_1v_2 \preceq w_1w_2$.

We define a pushdown automaton with bounded reversal counters (See Definition 5.5.2), A, that accepts the words $u = u_1u_2 \in dom(T)$ such that there exist $v_1, v_2, w_1, w_2 \in \Sigma^*$, $q_0, q_0' \in I$, $q, q' \in Q$, and $\sigma, \sigma' \in \Gamma^*$ that satisfy (**) and either (i) or (ii).

The automaton A simulates in parallel any two runs of T on the input word (with a squaring construction, see Definition 5.4.2). It guesses the end of u_1 and stores the states q and q' of the first and second run (in order to be able to check that the simulated runs of T are in state q, resp. q' after reading u_2). Non-deterministically, it checks whether (i) or (ii) holds.

To check (i), it uses two counters, one for each run. It does so by, after reading u_1, increasing the counters by the length of the output word of each transition of the corresponding run. Then, when reaching the end of u_2 it checks that both counters are different (by decreasing in parallel both counters and checking they do not reach 0).

Similarly, using two other counters, A checks that *(ii)* holds as follows. Note that condition *(ii)* implies that there is a position p such that the p-th letter a_1 of v_1v_2 and the p-th letter a_2 of w_1w_2 are different. The automaton A guesses the position $p \in \mathbb{N}$ of the mismatch, and initializes both counters to the value p. Then, while reading u_1u_2, it decreases each counter by the length of the output words of the corresponding run. When a counter reaches 0, A stores the output letter of the corresponding run. Finally, A checks that $a_1 \neq a_2$. T satisfies the HTP iff the language of A is empty. One can check that the construction of A is in PTime (similar to the squaring construction). The latter is decidable in co-NPTime (Theorem 5.5.6). □

7.3.3 HBM is Decidable for fVPTs

We now show that HTP characterizes HBM visibly pushdown transductions and therefore by Lemma 7.3.8 that HBM is decidable for VPT.

We proceed in two steps. We first show, in Lemma 7.3.9, that if $R(T)$ is HBM, then the HTP holds for T. Conversely, in Lemma 7.3.11 we show that if T satisfies the HTP, we can bound (exponentially with regards to the height of the input word) the maximal difference between outputs of T, $\mathsf{out}^{\max}_{\neq}$, on the same input word. Recall that we showed in Proposition 7.1.1 that the space complexity of Eval is polynomial in the height of the words and the length of the maximal difference between outputs ($\mathsf{out}^{\max}_{\neq}$). Therefore, Lemma 7.3.11 and Proposition 7.1.1 imply that the space complexity of Eval is bounded by the height of the input word when the HTP holds.

Lemma 7.3.9. *Let T be an* fVPT. *If $R(T)$ is HBM, then the HTP holds for T.*

Proof. First recall that we made the hypothesis that T is reduced, *i.e.* every accessible configuration is also co-accessible (if it is not, one must first reduce it Theorem 4.4.2). Suppose that the HTP does not hold for T. Therefore there are words u_1, u_2, u_3, u'_3, v_1, v_2, v_3, w_1, w_2, $w_3, w_3 \in \Sigma^*$, stacks σ, σ' and states $q, q' \in Q$, $q_0, q'_0 \in I$ and $q_f, q'_f \in F$ such that:

$$\left\{\begin{array}{lllllll} (q_0,\bot) & \xrightarrow{u_1/v_1} & (q,\sigma) & \xrightarrow{u_2/v_2} & (q,\sigma) & \xrightarrow{u_3/v_3} & (q_f,\bot) \\ (q'_0,\bot) & \xrightarrow{u_1/w_1} & (q',\sigma') & \xrightarrow{u_2/w_2} & (q',\sigma') & \xrightarrow{u'_3/w_3} & (q'_f,\bot) \end{array}\right.$$

and $\Delta(v_1, w_1) \neq \Delta(v_1v_2, w_1w_2)$. Let $K = \max(h(u_1u_2u_3), h(u'_1u'_2u'_3))$. By definition of $\mathsf{NFT}(T, K)$ (where states are configurations of T) and of the twinning property for NFT, the twinning property for NFT does not hold for $\mathsf{NFT}(T, K)$. Therefore $\mathsf{NFT}(T, K)$ is not subsequentializable [Cho77] and by Proposition 7.2.2 $R(\mathsf{NFT}(T, K))$ is not BM. Therefore $R(T)$ is not HBM, otherwise $R(T)$ could be evaluated in space complexity $f(h(u))$ on any input word u, for some function f. That corresponds to bounded memory if we fix the height of the words to K at most.

□

For the converse, we can apply the evaluation algorithm of Section 7.1, whose complexity depends on the maximal delay between all the candidate outputs of the input word. We first prove that this maximal delay is exponentially bounded by the height of the word.

We first need a simple lemma that shows some form of decomposition of the delay by concatenation.

Lemma 7.3.10. *For all $u, u', v, v' \in \Sigma^*$, $\Delta(uu', vv') = \Delta(\Delta(u, v) \cdot (u', v'))$.*

Proof. Let $X = uu' \wedge vv'$ and $Y = u \wedge v$. There exists A, B, C, D such that $A \wedge B = \epsilon$, $C \wedge D = \epsilon$, and:

$$\begin{array}{ll} uu' = XA & u = YC \\ vv' = XB & v = YD \\ \Delta(uu', vv') = (A, B) & \Delta(u, v) = (C, D) \end{array}$$

We have necessarily $|X| \geq |Y|$ since X is the longest common prefix of uu' and vv' and Y is the longest common prefix of u and v. Now we have $YCu' = XA$ and $YDv' = XB$, i.e. $Cu' = Y^{-1}XA$ and $Dv' = Y^{-1}XB$. Since $A \wedge B = \epsilon$, we have $\Delta(Cu', Dv') = (A, B)$, i.e. $\Delta(\Delta(u, v) \cdot (u', v')) = (A, B) = \Delta(uu', vv')$. □

Lemma 7.3.11. *Let T be an fVPT. If the HTP holds for T, then for all $s \in \Sigma^*$ we have $\mathsf{out}_{\neq}^{max}(s) \leq (|Q| \cdot |\Gamma|^{h(s)})^2 M$, where $M = \max\{|t| \mid q \xrightarrow{a/t,\gamma} q'\}$.*

Proof. Let $s \in \Sigma^*$. We consider two cases. We first assume that $s \in dom(T)$. Let $u \in \Sigma^*$ be a prefix of s, we will prove that $\mathsf{out}_{\neq}(u) \leq (|Q| \cdot |\Gamma|^{h(u)})^2 M$. Note that there exist $N = |Q| \cdot |\Gamma|^{h(u)}$ configurations reachable by words of height less than $h(u)$. The proof is similar to that of [BC02] for NFT. It proceeds by induction on the length of u. If $|u| \leq N^2$, then the result is trivial. Otherwise, assume that $|u| > N^2$ and let $(q, \sigma, w), (q', \sigma', w') \in Q \times \Gamma^* \times \Sigma^*$ such that there exist runs $\rho : (i, \perp) \xrightarrow{u|v} (q, \sigma)$ and $\rho' : (i', \perp) \xrightarrow{u|v'} (q', \sigma')$, with $i, i' \in I$, $v = \mathsf{lcp_{in}}(u, T) \cdot w$,

$v' = \mathsf{lcp}_{\mathsf{in}}(u,T)\cdot w'$, and such that $\mathsf{out}_{\neq}(u) = |w|$. As $|u| > N^2$, we can decompose these two runs as follows:

$$\begin{cases} \rho : (i,\bot) & \xrightarrow{u_1/v_1} & (q_1,\sigma_1) & \xrightarrow{u_2/v_2} & (q_1,\sigma_1) & \xrightarrow{u_3/v_3} & (q,\sigma) \\ \rho' : (i',\bot) & \xrightarrow{u_1/v'_1} & (q'_1,\sigma'_1) & \xrightarrow{u_2/v'_2} & (q'_1,\sigma'_1) & \xrightarrow{u_3/v'_3} & (q',\sigma') \end{cases}$$

In addition, we have $u = u_1u_2u_3$, $u_2 \neq \varepsilon$, $v = \mathsf{lcp}_{\mathsf{in}}(u,T)\cdot w = v_1v_2v_3$, and $v' = \mathsf{lcp}_{\mathsf{in}}(u,T)\cdot w' = v'_1v'_2v'_3$. Indeed, by the choice of N, there must exist a pair of configurations that occurs twice. By the HTP property, we obtain $\Delta(v_1v_2, v'_1v'_2) = \Delta(v_1,v'_1)$. By Lemma 7.3.10, this entails the equalities $\Delta(v_1v_2v_3, v'_1v'_2v'_3) = \Delta(\Delta(v_1v_2,v'_1v'_2)\cdot(v_3,v'_3)) = \Delta(\Delta(v_1,v'_1)\cdot(v_3,v'_3)) = \Delta(v_1v_3,v'_1v'_3)$. Thus, we obtain $\Delta(w,w') = \Delta(v,v') = \Delta(v_1v_2v_3,v'_1v'_2v'_3) = \Delta(v_1v_3,v'_1v'_3)$. As v_1v_3 and $v'_1v'_3$ are possible output words for the input word u_1u_3, whose length is strictly smaller than $|u|$, we obtain $|w| \leq \mathsf{out}_{\neq}(u_1u_3)$ and the result holds by induction.

We now consider the second case: $s \notin \mathit{dom}(T)$. Let s' be the longest prefix of s such that there exists s'' such that $s's'' \in \mathit{dom}(T)$. Since T is reduced (w.l.o.g., as explained in preliminaries), s' correspond to the longest prefix of s on which there exist a run of T. Therefore we have $\mathsf{out}^{\max}_{\neq}(s) \leq \mathsf{out}^{\max}_{\neq}(s')$ and we can apply the proof of the first case (as s' is a prefix of a word that belongs to $\mathit{dom}(T)$) and we get $\mathsf{out}^{\max}_{\neq}(s') \leq (|Q|\cdot|\Gamma|^{h(s')})^2M$. Moreover, $h(s') \leq h(s)$, therefore $\mathsf{out}^{\max}_{\neq}(s) \leq \mathsf{out}^{\max}_{\neq}(s') \leq (|Q|\cdot|\Gamma|^{h(s)})^2M$ and we are done. □

We have proved that the HTP characterizes HBM fVPT transductions. As HTP is decidable, so is HBM. Moreover, in the case the transduction is HBM, by Proposition 7.1.1 and Lemma 7.3.11, the memory required is at most exponential in the height of the input word.

Theorem 7.3.12. *Let T be an* fVPT*. It is decidable in co-*NPTIME *whether $R(T)$ is HBM. Moreover in this case, Algorithm* EVAL *runs in space complexity $O(|Q|^4\cdot|\Gamma|^{2h(u)+2}\cdot(h(u)+1)\cdot M)$ when evaluating u on T, where $M = \max\{|v| \mid q \xrightarrow{a/v,\gamma} q'\}$.*

Remark 7.3.13 (HBM vs Subsequentializable fVPTs)**.** *We have seen that a functional transduction defined by an* NFT *T is BM iff T is subsequentializable. The* VPT *of Figure 7.7 is an example of a transduction that is clearly not definable by a deterministic* VPT *but is HBM.*

Remark 7.3.14 (HBM is tight)**.** *We have mentioned that the space complexity of a* VPT *in HBM is at most exponential. We give here an example illustrating* *the tightness of this bound. The idea is to encode the tree transduction $f(t,a) \mapsto f(t,a) \cup f(t,b) \mapsto f(\overline{t},b)$ by a VPT, where t is a binary tree over $\{0,1\}$ and $\overline{t}$ is the mirror of t, obtained by replacing the 0 by 1 and the 1 by 0 in t. Thus taking the identity or the mirror depends on the second child of the root f. To evaluate this transformation in a streaming manner, one has to store the whole subtree t in memory before deciding to transform it into t or $\overline{t}$. The evaluation of this transduction cannot be done in polynomial space as there are a doubly exponential number of trees of height n, for all $n \geq 0$.*

7.4 Twinned VPTs

In the previous section, we have shown that a fVPT-transduction is in HBM iff the horizontal twinning property holds, and if it is in HBM, the algorithm of Section 7.1 uses a memory at most exponential in the height of the word. We exhibit in Section 7.6 an fVPT proving that this exponential bound is tight. To avoid this exponential cost, we identify in this section a subclass of HBM containing transductions for which the evaluation algorithm of Section 7.1 uses a memory *quadratic in the height of the word*. In order to achieve this, we strengthen the horizontal twinning property by adding some properties for well-nested loops. Some of our main and challenging results are to show the decidability of this property and that it depends only on the transduction, i.e. is preserved by equivalent transducers (in Section 7.5). We show that subsequential VPTs satisfy this condition, therefore our class not only contains all subsequential VPTs but also all *subsequentializable* VPTs.

7.4.1 Twinning Property

The property we introduce is a strengthening of the horizontal twinning property that we call the *twinning property (TP)*. Intuitively, the TP requires that two runs on the same input cannot accumulate increasing output delay on well-matched loops. They can accumulate delay on loops with increasing stack but this delay has to be caught up on the matching loops with descending stack.

Definition 7.4.1. *Let $T = (Q, I, F, \Gamma, \delta)$ be an* fVPT*. T satisfies the* twinning property *(TP) if for all $u_i, v_i, w_i \in \Sigma^*$ ($i \in \{1, \dots, 4\}$) such that u_3 is well-nested, and u_2u_4 is well-nested, for all $i, i' \in I$, for all $p, q, p', q' \in Q$, and for all $\sigma_1, \sigma_2 \in \bot.\Gamma^*$, for all $\sigma_1', \sigma_2' \in \Gamma^*$,*

$$\text{if} \quad \begin{cases} (i,\bot) \xrightarrow{u_1/v_1} (p,\sigma_1) \xrightarrow{u_2/v_2} (p,\sigma_1\sigma_1') \xrightarrow{u_3/v_3} (q,\sigma_1\sigma_1') \xrightarrow{u_4/v_4} (q,\sigma_1) \\ (i',\bot) \xrightarrow{u_1/w_1} (p',\sigma_2) \xrightarrow{u_2/w_2} (p',\sigma_2\sigma_2') \xrightarrow{u_3/w_3} (q',\sigma_2\sigma_2') \xrightarrow{u_4/w_4} (q',\sigma_2) \end{cases}$$

then $\Delta(v_1v_3, w_1w_3) = \Delta(v_1v_2v_3v_4, w_1w_2w_3w_4)$. *We say that a* fVPT T *is* twinned *whenever it satisfies the TP.*

Note that any twinned fVPT also satisfies the HTP (take $u_3 = u_4 = \epsilon$).

Example 7.4.2. *The VPT of Figure 7.7 does not satisfy the TP, as the delay between the two branches increases when iterating the loops. For example the delay after reading* ccr *is* (bbb, aaa) *but it is* $(bbbb, aaaa)$ *after reading* $cccrr$. *Recall however that this* VPT *does satisfy the HTP and is therefore HBM.*

On the contrary, consider the VPT *of Figure 7.9, it is obviously twinned as no word satisfies the premises of the TP (no word induces a nested loop on the upper and lower branches). However this transducer is not determinizable, as the output on the call symbols cannot be delayed to the matching return symbols.*

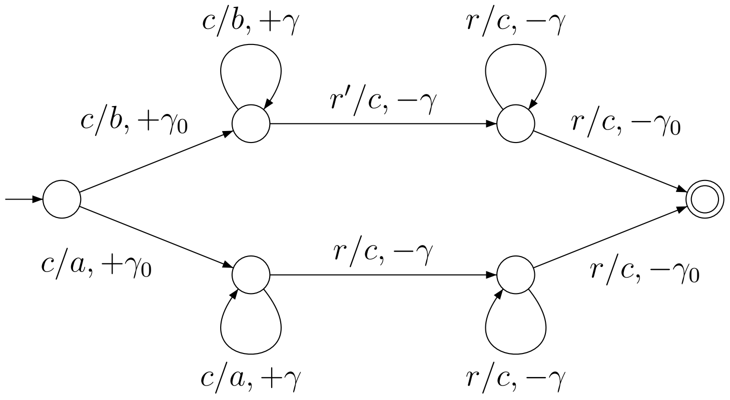


Figure 7.9: A functional VPT twinned but not determinizable.

7.4.2 Quadratic Height Bounded Memory Evaluation

We show in this section that VPT that satisfy the twinning property are evaluated by Eval in space quadratically (instead of exponentially for HBM transductions) bounded by the height of the input word.

The idea is the following. Consider an input word u, we have to show that the delay between two runs on u is quadratic in the height of u. Suppose there is a long well-nested subword in the input word, that is, the input word is of the form $u = u'wu''$ where w is well-nested. If w is sufficiently long, it can be shortened without changing the delay. There are two ways to shorten it: either it has a large height, in that case it is possible to apply the TP, or it is long and we can apply the HTP. As a result we show that the delay of u is equal to the delay of a word with the same delay and whose length is proportional to the current height of u. This yields the result.

Theorem 7.4.3. *Let T be an* fVPT *and $u \in \Sigma^*$. If T is twinned then the evaluation of T on u can be done in space complexity quadratic in $h(u)$ and exponential in $|T|$.*

Sketch. We have proven that the algorithm Eval runs in space complexity $O\left(p(T) \cdot (h(u)+1)^2 \cdot M\right)$ on T and $u \in \Sigma^*$, with $M = \max\{|v| : q \xrightarrow{a/v,\gamma} q'\}$ and $p(T) = |Q|^4 \cdot |\Gamma|^{2|Q|^4+2}$.

Let show that if the TP holds for T, then for any word $s \in \Sigma^*$, we have $\mathsf{out}^{\max}_{\neq}(s) \leq (h(s)+1) \cdot \Big((|Q| \cdot |\Gamma|^{|Q|^4})^2 + 1\Big) \cdot M$, where $M = \max\{|t| \mid q \xrightarrow{a/t,\gamma} q'\}$.

We assume that $s \in \mathit{dom}(T)$ (we can handle the case $s \notin \mathit{dom}(T)$ as in the proof of Lemma 7.3.11 for the HTP). We use the notion of *current height* $hc(u)$ of a prefix u of s. Consider a word u prefix of s. There exists a unique decomposition of u as follows: $u = u_0c_1u_1c_2 \dots u_{n-1}c_nu_n$, where $n = hc(u)$, and for any i, we have $c_i \in \Sigma_c$ and u_i is well-nested. Indeed, as $n = hc(u)$, the word u contains exactly n pending calls, that correspond to c_i's, and other parts of u can be gathered into well-nested words.

If each of the u_i's is such that $|u_i| \leq (|Q| \cdot |\Gamma|^{|Q|^4})^2$, then the property holds as length of word u can be bounded by $(hc(u)+1) \cdot \Big((|Q| \cdot |\Gamma|^{|Q|^4})^2 + 1\Big)$.

Otherwise, we prove that there exists a strictly shorter input word that produces the same delays as u when evaluating the transduction on it. Therefore, suppose that there exist two runs ϱ, ϱ' of T over u, and consider the smallest index i such that $|u_i| > (|Q| \cdot |\Gamma|^{|Q|^4})^2$. We distinguish two cases:

1. if $h(u_i) \leq |Q|^4$, then we can reduce the length of u_i using the HTP by exhibiting two configurations occurring twice in runs ϱ and ϱ'. This yields an input word u', strictly shorter than u, that produces the same delays as u (see the proof of Lemma 7.3.11).

2. if $h(u_i) > |Q|^4$, then we prove that we can "pump vertically" u_i (as depicted in Figure 7.10), and then reduce its length too. Indeed, let k be the first position in word u_i at which height $h(u_i)$ is obtained. As u_i is well-nested, we can define for each $0 \leq j < h(u_i)$ the unique position $\mathit{left}(j)$ (resp.

$right(j)$) as the largest index, less than k (resp. the smallest index, larger than k), whose height is j (see Figure 7.10). As $h(u_i) > |Q|^4$, there exist two heights j and j' such that configurations reached at positions $left(j)$, $left(j')$, $right(j)$ and $right(j')$ in runs ϱ and ϱ' satisfy the premises of the twinning property, considering the prefix $u_0c_1 \dots c_iu_i$ of u. Thus, one can replace in this prefix u_i by a shorter word u'_i and hence reduce its length, while preserving the delays reached after it. Let u' be the word obtained from u by substituting u'_i to u_i, hence $|u'| < |u|$. By Lemma 7.3.10, this entails that the delays reached after u and u' are the same, proving the result.

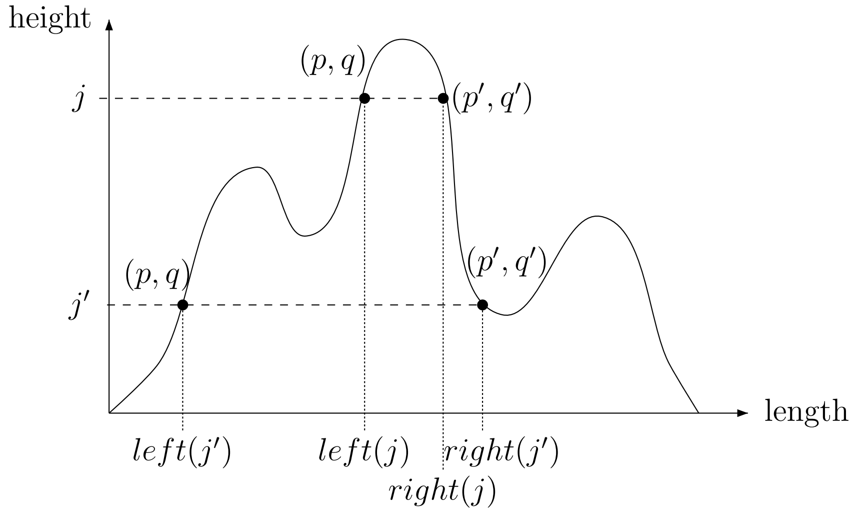


Figure 7.10: Vertical pumping in a well-nested word

□

7.4.3 Twinning Property is Decidable

As for the HTP, we can decide the TP using a reduction to the emptiness of a pushdown automaton with bounded reversal counters. The proof is technical but the techniques are similar to those used for proving the decidability of the HTP. We first prove a lemma that simplifies the construction of the counter automaton. Essentially, this lemma partitions the possible configurations in which a witness of non satisfiability of the TP can appear. A particular condition on the form of the witness is associated with each partition. These conditions, based on mismatch between the two output words, are easily checked with a counter automaton.

Lemma 7.4.4. *Let $A, B, C, D \in \Sigma^*$, such that $A = a_1 \dots a_l, B = b_1 \dots b_m, C = c_1 \dots c_n, D = d_1 \dots d_p$, and $l - m = n - p \geq 0$. We have that $\Delta(A, B) \neq \Delta(C, D)$ if and only if one of the following conditions is satisfied:*

1. *there exists k such that $a_{l-k} \neq b_{l-k}$ and either (i) $k \geq |C|$, or (ii) $c_{n-k} = d_{n-k}$;*
2. *there exists k such that $c_{n-k} \neq d_{n-k}$ and either (i) $k \geq |A|$, or (ii) $a_{l-k} = b_{l-k}$;*
3. *there exists k such that $a_{l-k} \neq c_{n-k}$ and either (i) $k < l - m$, or (ii) there exists k' with $a_{k'} \neq b_{k'}$ and $k + k' \leq l$;*
4. *there exist k, k' such that $b_{m-k} \neq d_{p-k}$ and $a_{k'} \neq b_{k'}$ and $k + k' \leq m$.*

Proof. Let us define $E = A \wedge B$ and $F = C \wedge D$, and also $A', B', C', D' \in \Sigma^*$ such that $A = EA', B = EB', C = FC'$ and $D = FD'$, i.e. $(A', B') = \Delta(A, B)$ and $(C', D') = \Delta(C, D)$.

We first prove that each condition implies $\Delta(A, B) \neq \Delta(C, D)$, i.e. that $(A', B') \neq (C', D')$. If $|A'| \neq |C'|$ or $|B'| \neq |D'|$ then the result is immediate. Now, assume that $|A'| = |C'|$ and $|B'| = |D'|$.

1. By hypothesis we have $a_{l-k} \neq b_{l-k}$ therefore $|A'| \geq k+1$ (as $a_{l-k} \in A'$) and $|B'| \geq k+1-(l-m)$ (as $b_{l-k} \in B'$). Thus $|C'| \geq k+1$ and $|D'| \geq k+m-l+1$. In particular, this implies that we are in case (ii) as $|C| \geq |C'| \geq k+1$. We consider two cases: (a) if $a_{l-k} \neq c_{n-k}$ we have $A' \neq C'$ (because $a_{l-k} \in A'$ and $c_{n-k} \in C'$ and are at the same position in A' and C' as $|A'| = |C'|$), or (b) if $a_{l-k} = c_{n-k}$ this means that $b_{l-k} \neq d_{n-k}$ (because $a_{l-k} \neq b_{l-k}$ and $c_{n-k} = d_{n-k}$), and so $B' \neq D'$ (because $b_{l-k} \in B'$ and $d_{n-k} \in D'$ because $|B'| = |D'|$).
2. Similar to proof of 1.
3. We prove that condition (i) implies $\Delta(A, B) \neq \Delta(C, D)$. Condition (ii) is proved similarly. We know that $|A'| \geq |A| - |B| = l - m > k$ therefore $a_{l-k} \in A'$. Similarly, $|C'| = |A'| > k$, so $c_{n-k} \in C'$. By hypothesis $a_{l-k} \neq c_{n-k}$, therefore, as $|A'| = |C'|$, A' and C' differ on their kth letter from the right.
4. Similar to proof of 3.

Now suppose that $\Delta(A, B) \neq \Delta(C, D)$ and let us show that one of the conditions is satisfied. First note that if $B' = \varepsilon$, then B is a prefix of A, and thus $|A'| = l - m$. In particular, we obtain $|A'| \leq |C'| \leq |C|$. We call this property $(\dagger_B)$. Similarly, $D' = \varepsilon$ entails $|C'| \leq |A'| \leq |A|$, what we denote by $(\dagger_D)$.

We prove the property by considering several cases:

- $|A'| > |C|$: take $k = |A'| - 1$, we have $a_{l-k} \neq b_{l-k}$. Note that b_{l-k} does not exist iff $l - k > m$, *i.e.* $B' = \varepsilon$. By property $(\dagger_B)$, this can not occur. In addition, by definition of k, we have $k \geq |C|$ and thus condition 1.(i) is satisfied.

- $|C'| > |A|$: take $k = |C'| - 1$, we have $c_{n-k} \neq d_{n-k}$ (as above, $(\dagger_D)$ ensures that d_{l-k} is well defined) and $k \geq |A|$: condition 2.(i) is satisfied.

- $|A'| \leq |C|$ and $|C'| \leq |A|$:

  - $|A'| > |C'|$ (this implies $|B'| > |D'|$): take $k = |A'| - 1$, we have $a_{l-k} \neq b_{l-k}$ and $c_{n-k} = d_{n-k}$ (existence of b_{l-k} and d_{n-k} is ensured by $(\dagger_B)$ and $(\dagger_D)$): condition 1.(ii) is satisfied.
  - $|C'| > |A'|$ (this implies $|D'| > |B'|$): take $k = |C'| - 1$, then condition 2.(ii) is satisfied.
  - $|A'| = |C'|$ (this implies $|B'| = |D'|$), we suppose $A' \neq C'$ and prove that condition 3 holds (the case $B' \neq D'$ is similar with condition 4 holding). Because $A' \neq C'$ and they have the same size, there must be a $k < |A'|$ with $a_{l-k} \neq c_{n-k}$, we consider two cases:
    - $|A'| \leq |A| - |B|$, this implies that $k < |A| - |B| = l - m$ therefore condition 3.(i) is satisfied.
    - $|A'| > |A| - |B|$, take $k' = l - |A'| + 1$ (the first position of A' in A). Then we have $a_{k'} \neq b_{k'}$ and thus condition 3.(ii) is satisfied.

□

Theorem 7.4.5. *The twinning property is decidable in co-*NPTime *for* fVPTs.

Proof. Let T be a fVPT. We construct in polynomial time a pushdown automaton with 6 counters (one reversal) that accepts any word $u = u_1u_2u_3u_4$ that satisfies the premise of the TP but such that $\Delta(v_1v_3, w_1w_3) \neq \Delta(v_1v_2v_3v_4, w_1w_2w_3w_4)$ (i.e. the TP is not verified). Therefore the TP holds if and only if no word is accepted by the automaton. This can be checked in co-NPTime (Theorem 5.5.6).

The automaton simulates any two runs, and guesses the decomposition $u_1u_2u_3u_4$ (it checks that the decomposition is correct by verifying that each run is in the same state after reading u_1 and u_2, and in the same state after reading u_3 and u_4). With two counters it can check that $|v_2v_4| = |w_2w_4|$, if it is not the case then it accepts u (indeed the TP is therefore not verified). The automaton checks that $\Delta(v_1v_3, w_1w_3) \neq \Delta(v_1v_2v_3v_4, w_1w_2w_3w_4)$ with the hypothesis (checked in parallel) that $|v_2v_4| = |w_2w_4|$. Let A, B, C, D be four words with $A = a_1 \dots a_l, B = b_1 \dots b_m, C = c_1 \dots c_n, D = d_1 \dots d_p$, $l \geq m$. We show in Lemma 7.4.4 below, that $\Delta(A, B) \neq \Delta(C, D)$ holds if, and only if, at least one out of four simple conditions is true. For example, the first condition states that there exists k such that $a_{l-k} \neq b_{l-k}$ and either (i) $k \geq |C|$, or (ii) $c_{n-k} = d_{n-k}$. The automaton guesses which condition holds and verifies it with the help of counters.

We detail how to check the first condition, the others can be checked with the same technique. The automaton guesses the value k and with two counters verifies that $a_{l-k} \neq b_{l-k}$ as follows. First it initializes both counters to $l - k$ (e.g. with an epsilon loop that increments both counters). Then it counts the letters of both words A and B up to $l - k$ and records a_{l-k} and b_{l-k} and verifies that they are not equal. Finally it must verify that either (i) or (ii) is satisfied. The verification of (i) is easy. To verify (ii), i.e. $c_{n-k} = d_{n-k}$, it uses two additional counters and proceeds similarly as for checking $a_{l-k} \neq b_{l-k}$, but checks equality instead of inequality. □

7.5 Twinned Transductions

The most challenging result of this thesis is to show that the TP only depends on the transduction and not on the transducer that defines it. The proof relies on fundamental properties of word combinatorics that allow us to give a general form of the output words $v_1, v_2, v_3, v_4, w_1, w_2, w_3, w_4$ involved in the TP. As many results of word combinatorics, the proof is a long case study, so that we give the full proof in Appendix A. First, we briefly give a taste of the combinatoric techniques used in our proof, and then we present the main ideas of the proof.

Some Combinatorics on Words

A word $x \in \Sigma^*$ is *primitive* if there is no word y such that $|y| < |x|$ and $x \in y^*$. The *primitive root* of a word $x \in \Sigma^*$ is the (unique) primitive word y such that

$x \in y^*$. In particular, if x is primitive, then its primitive root is x. Two words x and y are *conjugate* if there exists $z \in \Sigma^*$ such that $xz = zy$. It is well-known that two words are conjugate iff there exist $t_1, t_2 \in \Sigma^*$ such that $x = t_1t_2$ and $y = t_2t_1$. Two words $x, y \in \Sigma^*$ *commute* iff $xy = yx$.

The following lemma is a fundamental result of combinatorics on words. It states that if two words w, z of the form $w = x^n$ and $z = y^m$ have a sufficiently long common subword, then their primitive root are conjugate. In other words, in that case, you can write $w = (t_1t_2)^{n'}$ and $z = (t_2t_1)^{m'}$ for some words t_1t_2.

Lemma 7.5.1 (Folklore). *Let $x, y \in \Sigma^*$ and $n, m \in \mathbb{N}$. if x^n and y^m have a common subword of length at least $|x| + |y| - d$ (d being the greatest common divisor of $|x|$ and $|y|$), then their primitive roots are conjugate.*

This result is particularly useful when using pumping techniques that yield equation between words of the form $u_0u_1^nu_3u_4^nX$ and $v_0v_1^mv_3v_4^mY$, where we can take n and m as large as necessary. With these equations, we can infer an overlap between u_1^n and v_1^n as large as necessary and apply Lemma 7.5.1 to get that $u_1 = t_1t_2$ and $v_1 = t_2t_1$ for some words t_1 and t_2. With these techniques, we can infer the form of each word u_i and v_i, and deduce new equalities or inequalities involving these variables.

To get these equations for infinitely many n and m we make use of the following result by Hakala and Kortelainen.

Lemma 7.5.2 ([HK99]). *Let $u_0, u_1, u_2, u_3, u_4, v_0, v_1, v_2, v_3, v_4 \in \Sigma^*$. If*

$$u_0(u_1)^iu_2(u_3)^iu_4 = v_0(v_1)^iv_2(v_3)^iv_4$$

holds for $i \in \{0, 1, 2, 3\}$, then it holds for all $i \in \mathbb{N}$.

The rest of the combinatorics part of the proof is a long case study, where the cases consider whether some of these words are empty or not, or whether one word (involved in some equations) is longer or shorter than another.

Preservation of the TP between Equivalent Transducers

Theorem 7.5.3. *Let T_1, T_2 be two* fVPTs *such that $R(T_1) = R(T_2)$. T_1 is twinned iff T_2 is twinned.*

Sketch. We assume that T_1 is not twinned and show that T_2 is not twinned either. By definition of the TP there are two runs of the form

$$\begin{cases} (i_1, \perp) \xrightarrow{u_1/v_1} (p_1, \sigma_1) \xrightarrow{u_2/v_2} (p_1, \sigma_1\beta_1) \xrightarrow{u_3/v_3} (q_1, \sigma_1\beta_1) \xrightarrow{u_4/v_4} (q_1, \sigma_1) \\ (i_1', \perp) \xrightarrow{u_1/v_1'} (p_1', \sigma_1') \xrightarrow{u_2/v_2'} (p_1', \sigma_1'\beta_1') \xrightarrow{u_3/v_3'} (q_1', \sigma_1'\beta_1') \xrightarrow{u_4/v_4'} (q_1', \sigma_1') \end{cases}$$

such that $\Delta(v_1v_3, v_1'v_3') \neq \Delta(v_1v_2v_3v_4, v_1'v_2'v_3'v_4')$. We will prove that by repeating the loops on u_2 and u_4 sufficiently many times we will get a similar situation in T_2, proving that T_2 is not twinned. It is easy to show, by a pigeon hole argument, that there exist $k_2 > 0$, $k_1, k_3 \geq 0$, $w_i, w_i' \in \Sigma^*$, $i \in \{1, \dots, 4\}$, some states $i_2, p_2, q_2, i_2', p_2', q_2'$ of T_2 and some stack contents $\sigma_2, \beta_2, \sigma_2', \gamma_2'$ of T_2 such that we have the following runs in T_2:

$$\begin{cases} (i_2,\bot) \xrightarrow{u_1u_2^{k_1}/w_1} (p_2,\sigma_2) \xrightarrow{u_2^{k_2}/w_2} (p_2,\sigma_2\beta_2) \xrightarrow{u_2^{k_3}u_3u_4^{k_3}/w_3} (q_2,\sigma_2\beta_2) \xrightarrow{u_4^{k_2}/w_4} (q_2,\sigma_2) \\ (i_2',\bot) \xrightarrow{u_1u_2^{k_1}/w_1'} (p_2',\sigma_2') \xrightarrow{u_2^{k_2}/w_2'} (p_2',\sigma_2'\beta_2') \xrightarrow{u_2^{k_3}u_3u_4^{k_3}/w_3'} (q_2',\sigma_2'\beta_2') \xrightarrow{u_4^{k_2}/w_4'} (q_2',\sigma_2') \end{cases}$$

such that (q_1, σ_1) and (q_2, σ_2) are co-accessible with the same input word u_5, and (q_1', σ_1') and (q_2', σ_2') are co-accessible with the same input word u_5'. Now for all $i \geq 0$, we let

$$\begin{array}{ll} V^{(i)} = v_1(v_2)^{k_1+ik_2+k_3}v_3(v_4)^{k_1+ik_2+k_3} & W^{(i)} = w_1(w_2)^iw_3(w_4)^i \\ V'^{(i)} = v_1'(v_2')^{k_1+ik_2+k_3}v_3'(v_4')^{k_1+ik_2+k_3} & W'^{(i)} = w_1'(w_2')^iw_3'(w_4')^i \\ D_1(i) = \Delta(V^{(i)}, V'^{(i)}) & D_2(i) = \Delta(W^{(i)}, W'^{(i)}) \end{array}$$

In other words, $D_1(i)$ (resp. $D_2(i)$) is the delay in T_1 (resp. T_2) accumulated on the input word $u_1(u_2)^{k_1+ik_2+k_3}u_3(u_4)^{k_1+ik_2+k_3}$ by the two runs of T_1 (resp. T_2). There is a relation between the words $V^{(i)}$ and $W^{(i)}$. Indeed, since T_1 and T_2 are equivalent and (q_1, σ_1) and (q_2, σ_2) are both co-accessible by the same input word, for all $i \geq 1$, either $V^{(i)}$ is a prefix of $W^{(i)}$ or $W^{(i)}$ is a prefix of $V^{(i)}$. We have a similar relation between $V'^{(i)}$ and $W'^{(i)}$.

We prove in Appendix the following intermediate results: (i) there exists $i_0 \geq 0$ such that for all $i, j \geq i_0$ such that $i \neq j$, $D_1(i) \neq D_1(j)$; (ii) for all $i, j \geq 1$, if $D_1(i) \neq D_1(j)$, then $D_2(i) \neq D_2(j)$. The proofs of those results rely on fundamental properties of word combinatorics and a non-trivial case study that depends on how the words $v_1(v_2)^{k_1+ik_2+k_3}v_3(v_4)^{k_1+ik_2+k_3}$ and $w_1(w_2)^iw_3(w_4)^i$ are overlapping. Thanks to (i) and (ii), we clearly get that $D_2(i_0) \neq D_2(i_0+1)$, which provides a counter-example for the twinning property. $\square$

Determinizable VPTs are Twinned

Deterministic transducers have at most one run per input word, they, therefore, trivially satisfy the TP. As a consequence, by Theorem7.5.3 all determinizable VPTs do also.

Theorem 7.5.4. *Determinizable* VPTs *are twinned.*

The TP is not a sufficient condition to be determinizable, as shown for instance in Example 7.4.2. Therefore the class of transductions defined by transducers which satisfy the TP is strictly larger than the class of transductions defined by determinizable transducers. However, these transductions are in the same space complexity class for evaluation, i.e. polynomial space in the height of the input word for a fixed transducer.

7.6 Conclusion and Remarks

In this chapter, we investigated the streaming evaluation of nested word transductions, and in particular identifies an interesting class of VPT-transductions which subsumes subsequentializable transductions and can still be efficiently evaluated. The following inclusions summarize the relations between the different *classes* of transductions we have studied:

BM fVPTs $\subsetneq$ Subsequentializable VPTs $\subsetneq$ twinned fVPTs $\subsetneq$ HBM fVPTs $\subsetneq$ fVPTs

Moreover, we have shown that BM, twinned and HBM fVPTs are decidable in co-NPTime.

Further Directions An important property of the class of twinned fVPTs w.r.t. the class of subsequentializable VPTs is that it is decidable. It would thus be interesting to determine whether or not the class of subsequentializable VPTs is decidable. In addition, we also plan to extend our techniques to more expressive transducers, such as those recently introduced in [AD11], which extend VPTs with global variables and are as expressive as MSO-transductions, and can therefore swap or reverse sub-trees. Another line of possible future work concerns the extension of our evaluation procedure, which holds for functional transductions, to finite valued transductions.

Related Work In the context of XML, visibly pushdown automata based streaming processing has been extensively studied for validating XML streams [Kum07, SV02, BLS06, SS07]. The validation problem with bounded memory is studied in [BLS06] when the input is assumed to be a well-nested word and in [SV02, SS07] when it is assumed to be a well-formed XML document (this problem is still open). Querying XML streams has been considered in [GHS09]. It consists in selecting a set of tuples of nodes in the tree representation of the XML document. For monadic queries (selecting nodes instead of tuples), this can be achieved by a functional VPT returning the input stream of tags, annotated with Booleans indicating selection by the query. However, functional VPTs cannot encode queries of arbitrary arities. The setting for functional VPTs is in fact different to query evaluation, because the output has to be produced on-the-fly in the right order, while query evaluation algorithms can output nodes in any order: an incoming input symbol can be immediately output, while another candidate is still to be confirmed. This makes a difference with the notion of concurrency of queries, measuring the minimal amount of candidates to be stored, and for which algorithms [BJ07, GNT09] and lower bounds [BYFJ05, Ram10, Gau09] have been proposed. VPTs also relate to tree transducers (see Section 6.4), for which no comparable work on memory requirements is known. When allowing two-way access on the input stream, more space-efficient algorithms for XML validation [KM10] and querying [MV09] have been proposed.

Chapter 8

Look-ahead

Contents

In this chapter, we investigate the extension of visibly pushdown transducers with visibly pushdown look-ahead ($\mathsf{VPT}_{\mathsf{la}}$). They are visibly pushdown transducers whose transitions are guarded by visibly pushdown automata that can check whether the well-nested subword starting at the current position belongs to the language they define. First, we show that $\mathsf{VPT}_{\mathsf{la}}$ are not more expressive than VPTs, but are exponentially more succinct. Second, we show that the class of deterministic $\mathsf{VPT}_{\mathsf{la}}$ corresponds exactly to the class of functional VPTs, yielding a simple characterization of functional VPTs. As a consequence, we show that any functional VPTs is equivalent to an unambiguous one. Finally, we show that while $\mathsf{VPT}_{\mathsf{la}}$ are exponentially more succinct than VPTs, checking equivalence or inclusion of functional $\mathsf{VPT}_{\mathsf{la}}$ is, as for VPTs, ExpTime-c. Additionally, we derive similar results for visibly pushdown automata with look-ahead.

One of our motivations is to give a simple characterization of functional VPTs that can be checked easily. Deterministic VPTs are not expressive enough to capture all functional VPTs, as for instance swapping the first and last letters of a word cannot be done deterministically. Instead of non-determinism, we show that some limited inspection of the longest well-nested subword starting at the current position (called the *current well-nested prefix*) is required to capture (non-deterministic) functional VPTs. More precisely, we show that functional VPTs-transductions are captured by deterministic VPTs extended with visibly pushdown look-aheads that inspect the current well-nested prefix. Moreover, inspecting the current well-nested prefix is somehow the minimal necessary information to capture all functional VPTs.

A VPA with visibly pushdown look-ahead ($\mathsf{VPA}_{\mathsf{la}}$) is a VPA such that call transitions are guarded by visibly pushdown automata (VPA). When reading a call at position i, a $\mathsf{VPA}_{\mathsf{la}}$ can apply a call transition provided the longest well-nested word starting at position i is included in the language of the VPA of the transition. Visibly pushdown transducers with look-ahead ($\mathsf{VPT}_{\mathsf{la}}$) are, as usual, $\mathsf{VPA}_{\mathsf{la}}$ equipped with an output morphism.

We first show that $\mathsf{VPT}_{\mathsf{la}}$, resp. $\mathsf{VPA}_{\mathsf{la}}$, are as expressive as VPTs, resp. VPA, but exponentially more succinct. For this we present an exponential construction that shows how a VPA can simulate look-aheads. Moreover we show this exponential blow-up is unavoidable.

Then, we prove that deterministic $\mathsf{VPT}_{\mathsf{la}}$ and functional VPTs are equally expressive. This equivalence is obtained by a construction (which is also exponential) that replaces the non-determinism of the functional VPT with deterministic look-ahead. This also yields a simple characterization of functional VPTs.

Next we show that the equivalence and inclusion problems for functional $\mathsf{VPT}_{\mathsf{la}}$, resp. $\mathsf{VPA}_{\mathsf{la}}$, are, as for VPTs, resp. VPA, ExpTime-c. Therefore even though $\mathsf{VPT}_{\mathsf{la}}$ are exponentially more succinct than VPTs, testing equivalence of functional $\mathsf{VPT}_{\mathsf{la}}$ is not harder than for functional VPTs. This is done in two steps. First one checks equivalence of the domains. Then one checks that the union of the two transducers is still functional. We show that testing functionality is ExpTime-c for $\mathsf{VPT}_{\mathsf{la}}$: get rid of the look-aheads with an exponential blow-up and test in PTime the functionality of the constructed VPT. To verify that the domains are equivalent, the naive technique (removing the look-aheads and then verifying the mutual inclusion of the domains) yields a doubly exponential algorithm. Instead, we show that the domains of $\mathsf{VPT}_{\mathsf{la}}$ are linearly reducible to alternating top-down tree automata. Testing the equivalence of such automata can be done in ExpTime [CDG+07].

As an application of look-aheads, we show that a nice consequence of the constructions involved in the previous contributions is that functional VPTs are effectively characterized by unambiguous VPTs. This result was already known for finite-state transducers [Eil74, Sch76, EM65] and here we extend it to VPTs with rather simple constructions based on the concept of look-aheads. This characterization of functional finite-state transducers has been generalized to k-valued and k-ambiguous finite-state transducers [Web93] and with a better upper-bound [SdS10] based on lexicographic decomposition of transducers.

Finally, we discuss slightly different look-aheads. First, we consider look-aheads that are allowed to inspect the whole current prefix until the end of the word. We show that this does not add expressivity nor succinctness. Second, we show that restricting the look-ahead to the current prefix in between the current call and its matching return (*i.e.* the subtree rooted at the current node) is not sufficient to have a characterization of functional VPTs by deterministic $\mathsf{VPT}_{\mathsf{la}}$. All these results are new indications that the class of VPTs is robust.

8.1 Definitions

Assumptions. In order to simplify notations and proofs we suppose in this chapter that a structured alphabet has no internal symbols, that is $\Sigma = (\Sigma_c, \Sigma_r)$. Indeed, one can encode an internal symbol with a specific call symbol directly followed by a specific return symbol.

Moreover we also suppose in this chapter 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 to extend the results of this chapter to the general definitions of VPA and VPT given in Chapter 4 and 5.

Given a word w over Σ we denote by $\text{pref}_{\text{wn}}(w)$ the longest well-nested prefix of w. E.g. $\text{pref}_{\text{wn}}(crc) = cr$, $\text{pref}_{\text{wn}}(crcrrcrr) = crcr$. 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 without initial state. On a call symbol c, T can trigger the look-ahead from a state p of the VPA (which depends on the call transition). The look-ahead tests membership of the longest well-nested prefix of the current suffix (that starts by the letter c) to $L(A_p)$, where A_p is the VPA A with initial state 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).

We first define the notion of *look-ahead visibly pushdown automata*, they are VPA with no initial states. Each state q of a look-ahead VPA B defines a language:

the language of the VPA B_q which behaves just like B and whose initial state is q.

Definition 8.1.1. *A look-ahead* VPA *is a tuple $B = (Q^{la}, F^{la}, \Gamma^{la}, \delta^{la})$, where:*

- *Q^{la} is the finite set of states,*
- *$F^{la} \subseteq Q^{la}$ is the set of final states,*
- *Γ^{la} is the stack alphabet, and*
- *δ^{la} is the transition relation (as in a* VPA*).*

Moreover, for all $q \in Q^{la}$, we define the VPA *$B_q = (Q^{la}, \{q\}, F^{la}, \Gamma^{la}, \delta^{la})$.*

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 8.1.2 (VPA$_{\mathsf{la}}$)**.** *A* VPA *with look-ahead (*VPA$_{\mathsf{la}}$*) is a pair $A_{la} = (A, B)$ where $B = (Q^{la}, F^{la}, \Gamma^{la}, \delta^{la})$ is a look-ahead* VPA*, and A is a tuple $A = (Q, I, F, \Gamma, \delta)$ such that*

- *Q is the finite set of states,*
- *$I \subseteq Q$, resp. $F \subseteq Q$, is the set of initial, resp. final, states,*
- *Γ is the stack alphabet, and*
- *$\delta = \delta_c \uplus \delta_r$ is the transition relation such that:*
  - *$\delta_c \subseteq Q \times \Sigma_c \times Q^{la} \times \Gamma \times Q$,*
  - *$\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$.*

Let $u \in \Sigma^*$. A run of A_{la} on $u = a_1 \dots a_l$ is a sequence of transitions $\rho = t_1 \dots t_l \in \delta^*$ such that, if $t_k = (q, a_k, p, \gamma, q') \in \delta_c$, then we have $\mathrm{pref}_{\mathrm{wn}}(a_{k+1} \dots a_l) \in L(A_p)$, moreover ρ is a run of A considered as a VPA. The run ρ is accepting if $\sigma_0 = \sigma_l = \perp$ and $q_l \in F$. The size of a VPA$_{\mathsf{la}}$ $A^{la} = (A, B)$ is the size of A plus the size of B.

A 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-ahead 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 8.1.3 (Deterministic $\mathsf{VPA_{la}}$). *A $\mathsf{VPA_{la}}$ $A_{la} = (A, B)$ where $B = (Q^{la}, F^{la}, \Gamma^{la}, \delta^{la})$, and $A = (Q, I, F, \Gamma, \delta)$ is deterministic if the following conditions hold:*

- $|I| = 1$,
- *for all $(q, c, p_1, \gamma_1, q_1), (q, c, p_2, \gamma_2, q_2) \in \delta_c$, if $\gamma_1 \neq \gamma_2$ or $q_1 \neq q_2$ or $p_1 \neq p_2$, then $L(A_{p_1}) \cap L(A_{p_2}) = \varnothing$,*
- *for all $(q, r, \gamma, q_1), (q, r, \gamma, q_2) \in \delta_r$ then $q_1 = q_2$,*
- *B is deterministic.*

Note that deciding whether some $\mathsf{VPA_{la}}$ is deterministic can be done in PTIME. One has to check that for each state q and each call symbol c, the VPLs guarding the transition from state q and reading c are *pairwise* disjoint. By Theorem 4.3.2, as B is deterministic, this can be done in PTIME.

Following the definition of transducers in Chapter 3, we define a visibly pushdown transducer with look-ahead from the input alphabet Σ to the output alphabet Δ, as a $\mathsf{VPA_{la}}$ with an output morphism.

Definition 8.1.4 ($\mathsf{VPT_{la}}$). *A visibly pushdown transducer with look-ahead ($\mathsf{VPT_{la}}$) from Σ to Δ is a pair $T = (A, \Omega)$ where A is a $\mathsf{VPA_{la}}$ and Ω is the output morphism define from δ to Δ^*, where δ 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 $\mathsf{VPT_{la}}$.

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 8.1.5. *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*

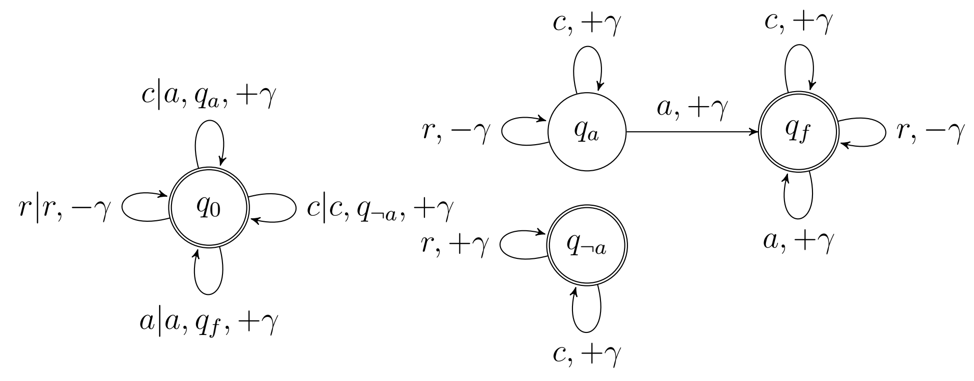


Figure 8.1: A $\mathsf{VPT}_{\mathsf{la}}$ (left) and its look-ahead (right) on $\Sigma_c = \{c, a\}$ and $\Sigma_r = \{r\}$

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 8.1 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 VPTs 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 8.2.1 a VPT that is equivalent to the $\mathsf{VPT}_{\mathsf{la}}$ defined in Example 8.1.5.

Example 8.2.1. *The* VPT $T = (Q, I, F, \Gamma, \delta)$ *defines the transduction of Example 8.1.5, 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:

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

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 8.2.1 relies heavily on non-determinism (*i.e.* on guesses). 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 Theorem 4.2.2), 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 $(q, \perp)$ is accessible from $(p, \perp)$ by reading w. In the next theorem, we use 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 8.2.2. *For any* $\mathsf{VPT_{la}}$*, resp.* $\mathsf{VPA_{la}}$*,* 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 $\mathsf{VPA_{la}}$. Let $A_{la} = (A, B)$ 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 on 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-ahead 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''.

Let us consider a word $wcw'r$ for some well-nested words w, w' (depicted on Fig. 8.2). 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

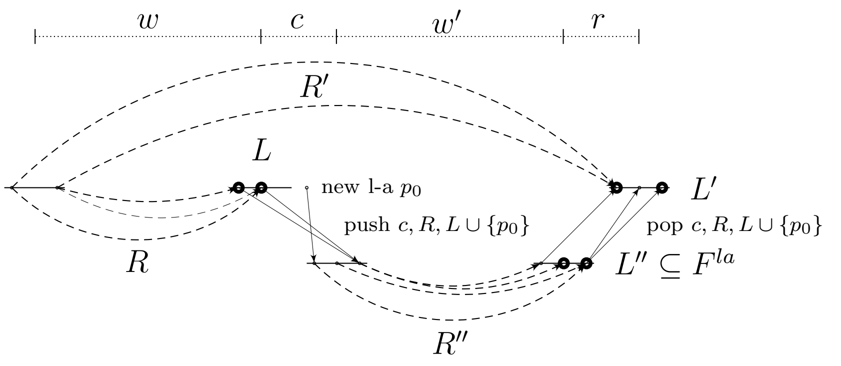


Figure 8.2:

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: $(q'', R'', L'') \xrightarrow{r,(\gamma,R,L,c)} (q', R', L') \in \delta'_r$ if the following conditions hold:

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

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

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

(*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 $sh(w) = \epsilon$.

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 us 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 us show that the construction also works for transducers. Let $T = (A_{la}, \Omega)$ where $A_{la} = (A, B)$ 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 VPA$_{\mathsf{la}}$ 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 $R(T) = R(T')$. □

Succinctness. The exponential blow-up in the construction of Theorem 8.2.2 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 $\Sigma = (\Sigma_c, \Sigma_r)$ has not internal) 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 n-th first 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 8.2.3 (Succinctness)**.** VPA$_{\mathsf{la}}$ *are exponentially more succinct than* VPA.

8.3 Functional VPTs and VPT$_{\mathsf{la}}$

While there is no known syntactic restriction on VPTs that captures all functional VPTs, we show that the class of deterministic VPT$_{\mathsf{la}}$ captures all functional VPTs. Given a functional VPT we construct an equivalent deterministic 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 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 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 [Eng78] 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 8.3.1. *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 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 $\mathsf{VPT}_{\mathsf{la}}$ $T' = ((A', B), \Omega')$, where (A', B) 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 $w = c_1w_1r_1w_2c_3w_3r_3$ where w_i are well-nested, this word is depicted on Fig. 8.3. 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. 8.3). 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.

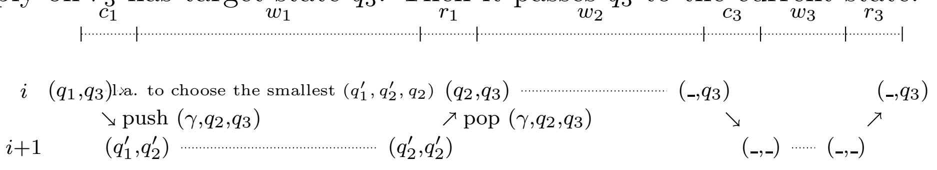


Figure 8.3:

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 VPTs accepts only words of the form cwr, where w is well-nested, so that one only needs to consider triples of states.

We now 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 $c_1w_1r_1w_3$ 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 $w_3 = \epsilon$ 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 $B_{q_1,q_1',q_2',q_2,q_3}$ 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 VPAs 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 VPAs, $A_{la} = \biguplus B_{q_1,q_1',q_2',q_2,q_3}$. We now define the call and return transitions of T' as follows, for all $c \in \Sigma_c, r \in \Sigma_r, \gamma \in \Gamma, q_1, q_1', q_2', q_3, q \in Q, u \in \Sigma^*$:

$$\begin{array}{llcccl}
T' \models & (q_1,q_3) & \xrightarrow{c|u,\ (\gamma,q_2,q_3),\ p_{q_1,q_1',q_2',q_2,q_3}} & (q_1',q_2') & \text{if} & T \models (q_1 \xrightarrow{c|u,\gamma} q_1') \\
T' \models & q_0 & \xrightarrow{c|u,\ (\gamma,q_3,q_3),\ p_{q_0,q_1',q_2',q_3,q_3}} & (q_1',q_2') & \text{if} & T \models (q_0 \xrightarrow{c|u,\gamma} q_1') \\
T' \models & (q_2',q) & \xrightarrow{r|u,(\gamma,q_2,q_3)} & (q_2,q_3) & \text{if} & T \models (q_2' \xrightarrow{r|u,\gamma} q_2)
\end{array}$$

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 $p_{q_1,q_1',q_2',q_2,q_3}$ and $p_{q_1,s_1',s_2',s_2,s_3}$. By definition of the look-aheads, we have $L(B_{q_1,q_1',q_2',q_2,q_3}) \cap L(B_{q_1,s_1',s_2',s_2,s_3}) = \varnothing$. 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 8.2.2 that removes the look-aheads, yields a nice characterization of functional VPTs:

Theorem 8.3.2. *For all functional* VPT T*, one can effectively construct an equivalent unambiguous* VPT T'.

8.4 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. VPTs, 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 VPTs.

Theorem 8.4.1. *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 [CDG+07].

Given n deterministic top-down binary tree automata $T_1, \dots, T_n$ over an alphabet Δ, one can construct in linear-time n *deterministic* VPAs $A_1, \dots, A_n$ that define the same languages as $T_1, \dots, T_n$ respectively, modulo the natural encoding of trees as nested words over the structured alphabet $\tilde{\Delta} = \{c_a \mid a \in \Delta\} \uplus \{r_a \mid a \in \Delta\}$ (See Section 6.4). 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 $w_{n,t} = c_1 r_1 \dots c_n r_n enc(t)$ 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 $\bigcap_i L(A_i) = \emptyset$.

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

Proof. For the ExpTime upper-bound, we first apply Theorem 8.2.2 to remove the look-aheads. This results in a VPT possibly exponentially bigger. Then functionality can be tested in PTime (Theorem 5.4.6).

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 (Theorem 4.3.2). 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 [CDG+07].

Theorem 8.4.3. *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 VPAs. 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 6.4.5. VPAs 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 [Gau09]. Look-aheads of VPAs inspect the longest well-nested prefix of the current suffix. This corresponds to subtrees in first-child next-sibling encodings of unranked trees. Therefore VPAs 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 [CDG+07].

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

8.5 Discussion on other Definitions of Look-ahead

In this section we discuss several other definitions of look-ahead. In particular we informally show that the closure by look-ahead (Theorem 8.2.2) and the equivalence between deterministic $\mathsf{VPT}_{\mathsf{la}}$ and functional VPT (Theorem 8.3.1) still hold when the look-ahead can inspect the whole suffix and can also be triggered on return transitions. However, when the look-ahead can inspect only the current well-nested prefix of the form cwr (corresponding to the first subtree of the current hedge in a tree), it is not sufficient to express all functional VPT with determinism.

Shorter look-aheads

Instead of inspecting the longest well-nested current prefix, we could restrict the look-ahead to visit only the current well-nested prefix of the form cwr, we call such look-ahead, the *short look-ahead.* Consider for example the word

$$cwc_1w_1r_1c_2w_2r_2r$$

when reading c_1 the short look-ahead reads $c_1w_1r_1$ instead of $c_1w_1r_1c_2w_2r_2$ for the visibly pushdown look-ahead (Definition 8.1.2). In a tree view of the nested word, this would correspond to the first subtree of the current hedge. While VPTs would still be closed by such look-aheads, we would not have a correspondence between deterministic $\mathsf{VPT}_{\mathsf{la}}$ and functional VPTs anymore. For instance, the family of transductions $(L_n)_n$ defined in Section 8.1 can clearly not be defined by a deterministic $\mathsf{VPT}_{\mathsf{la}}$ with this shorter look-head, although it is a functional transduction. Somehow, our definition of visibly pushdown look-ahead is the

	VPA	$\mathsf{VPA}_{\mathsf{la}}$
Emptiness	PTime	ExpTime-c
Universality	ExpTime-c	ExpTime-c
Inclusion	ExpTime-c	ExpTime-c
Equivalence	ExpTime-c	ExpTime-c

Table 8.1: Decision Problems for VPA and $\mathsf{VPA}_{\mathsf{la}}$

shortest look-ahead that yields the equivalence between deterministic $\mathsf{VPT}_{\mathsf{la}}$ and functional VPTs.

Longer look-aheads

Another way of adding look-aheads is to allow them to inspect the whole current suffix. Such look-aheads can be defined by visibly pushdown automata where return transitions on empty stack are allowed. The construction of Theorem 1 can be slightly modified to show that VPTs are still closed by such look-aheads. The idea is to extend the states with a new component $L_\perp \subseteq Q^{la}$ that corresponds to states of look-aheads that were started at a deeper position than the current position. For those states we apply only return transitions on empty stack when reading a return symbol. Obviously, VPTs with such look-aheads still satisfy the correspondence between functional VPTs and deterministic VPTs with look-ahead.

Look-aheads on return transitions.

One could also allow look-aheads on return transitions. Such look-aheads would inspect the longest well-nested prefix starting just after the current return symbol. This can be simulated (in PTime) by look-aheads on call transitions. One just delay one step the look-ahead as follows. If the next symbol is a return, then the look-ahead is stopped (before starting), as the longest well nested prefix is empty. If the next symbol is a call, then the look-ahead is coupled with the look-ahead of the call transition (by a product construction). Therefore our results still hold in this setting.

	VPT [FRR+10b]	$\mathsf{VPT}_{\mathsf{la}}$
Emptiness	PTime	ExpTime-c
Inclusion/equivalence of functional	ExpTime-c	ExpTime-c
Functionality	PTime	ExpTime-c

Table 8.2: Decision Problems for VPT,$\mathsf{VPT}_{\mathsf{la}}$

8.6 Conclusion

We have introduced visibly pushdown automata and transducers with look-ahead. We have shown that they are not more expressive than VPA, resp. VPT, but can be exponentially more succinct. Next, we have proven that the class of deterministic $\mathsf{VPT}_{\mathsf{la}}$ characterizes the functional VPT transductions, and as a consequence, so do the unambiguous VPT. Finally, we have shown that while they are exponentially more succinct, it comes with no cost in time complexity, except for emptiness and functionality. Table 8.1 and 8.2 summarizes our results. Note that universality of $\mathsf{VPA}_{\mathsf{la}}$ is in ExpTime as it amounts to check equivalence with the universal language. It is ExpTime-hard because universality of VPA is already ExpTime-hard. Finally, note that universality is not relevant to transducers as VPT are never universal, since every word has only a finite number of outputs.

An interesting extension would be to generalize those results to k-valued VPTs: is any k-valued VPT equivalent to a k-ambiguous $\mathsf{VPT}_{\mathsf{la}}$? This question is more difficult than for functional VPTs, as for k-valued VPTs, among a set of possible transitions it is necessary to choose for each output word (among at most k output words) exactly one transition, in order to turn k-valuedness into k-ambiguity. It is not clear how to use look-aheads to make such choices.

Related Works Regular look-aheads have been mainly considered for classes of tree transducers, where a transition can be fired provided the current subtree belongs to some regular tree language. For instance, regular look-aheads have been added to *top-down (ranked) tree transducers* in order to obtain a robust class of tree transducers that enjoys good closure properties wrt composition [Eng77], or to *macro tree transducers* (MTTs) [EV85]. For top-down tree transducers, adding regular look-ahead strictly increases their expressive power while MTTs are closed by regular look-ahead [EV85]. Another related result shows that every functional top-down tree transduction can be defined by a *deterministic* top-down tree transducer with look-ahead [Eng78]. However we saw in Section 6.4.5 that the expressiveness of top-down tree transducers for unranked transductions is incomparable with the expressiveness of VPT.

Macro tree transducers subsume VPTs (See Section 6.4) and as we said before, there is a correspondence between the two notions of look-aheads, for VPTs and MTTs respectively. However it is not clear how to derive our results on closure by look-aheads from the same result on MTTs, as the latter highly relies on parameters and it would require back-and-forth encodings between the two models. The direct constructions we give in this chapter are self-contained and allows one to derive the characterization of functional VPTs as unambiguous VPTs by a careful analysis of the construction.

Chapter 9

Conclusion

In this thesis, we have introduced the class of visibly pushdown transducers. We have shown that, contrarily to the more expressive class of pushdown transducers, this class enjoys good properties. Notably, we have shown that functionality is decidable in PTime and k-valuedness in co-NPTime. We also have shown that for functional VPT transductions the equivalence and inclusion problems are ExpTime-c and that they are decidable in PTime for deterministic VPT. All these problems are undecidable for pushdown transducers. Furthermore, we have shown that the class is closed under (visibly pushdown) look-ahead. Finally, we showed that the deterministic VPT with look-ahead, as well as the unambiguous VPT, characterize the functional VPT transductions. All these results provide evidences that this class is a robust generalization of finite state transducers.

However, while the class of finite state transducers is closed under composition and has a decidable type checking problem against NFA, the class of VPT is not closed under composition and its type checking problem against VPA is undecidable. We have identified the subclass of well-nested VPT and we have shown that it is closed under composition and its type checking problem against VPA is ExpTime-c. Moreover, we have highlighted the strong connection between these nested word transducers and various classes of tree transducers. We have shown that, compared to these tree transducers classes, the class of wnVPT enjoys limited expressivity due to the fact that it has no mechanism to duplicate or to move part of the input document. Nonetheless, while this duplication mechanism induces the main challenges for tree transducers, for wnVPT it is the ability to concatenate hedges that renders problems not trivial.

Finally, we have investigated the problem of performing efficiently the transductions defined by VPT. We presented the Eval algorithm that uses a compact data structure to represent runs. We exhibited a necessary and sufficient condition, decidable in co-NPTime, for a VPT transduction to be height-bounded memory, *i.e.* such that it can be performed with a memory that depends on the height of the input word but not on its length. We have shown that in the worst case the memory depends exponentially on the height of the input words.

We presented a second condition, called the twinning property for VPTs, such that the VPTs that satisfy it, called the twinned VPTs, can be evaluated with a memory that depends quadratically on the height of the word. This condition is not a necessary condition, we show that there are some VPTs that are not twinned, but whose transduction can be performed by a turing transducer in a memory linear in the height of the word. We showed that the class of twinned transducers is strictly larger than the class of determinizable VPT transductions, nevertheless, the twinned transductions can be performed with the same space complexity, *i.e.* polynomially in the height of the input word.

Open Problems

In spite of our utmost efforts, some problems remain open. We here provide a short list of the most important open problems.

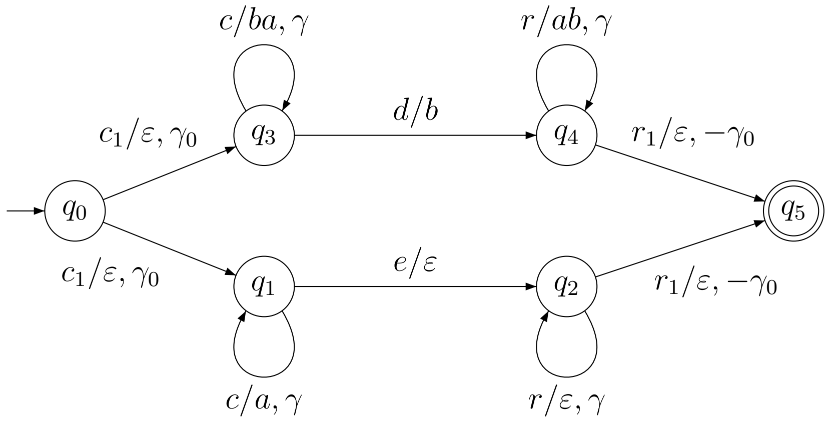


Figure 9.1: A determinizable functional VPT T_1.

Determinization of VPT. We have seen in Chapter 3 that some non-deterministic finite state transducers (NFT) are functional but have no equivalent deterministic ones (like for example the one that swap the first and last letter). The problem of

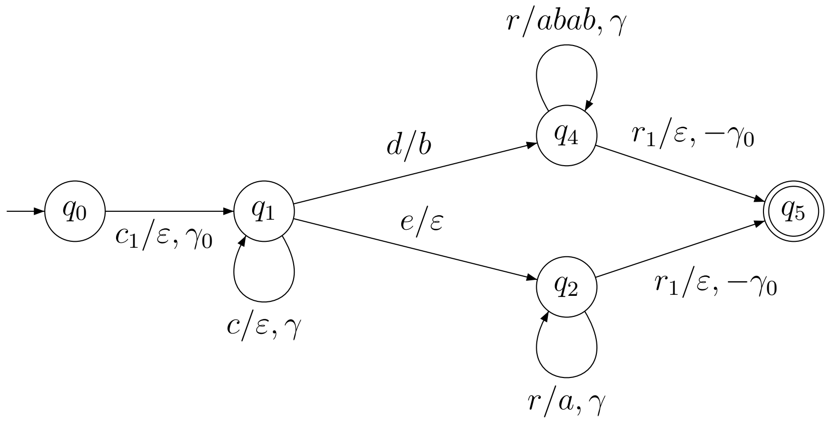


Figure 9.2: A deterministic VPT T'_d.

deciding whether a NFT is determinizable is decidable, it goes back to checking whether the so-called *twinning property* for NFT holds or not. If the twinning property holds, then there exists $k \in \mathbb{N}$ such that for any two runs on the same input word u, the size of the delay between both outputs is bounded by k. One can then perform a construction whose states are sets of pairs (q, v) where q is a state of the original transducer and v is the delay associated with q. This construction can be done with a fixed point algorithm. Because the size of this delay is bounded by a constant k (which depends on the size of the transducer) the state space of the deterministic NFT is finite and the determinization procedure terminates.

For VPT, the situation is more challenging for two reasons. First, there exist some determinizable VPT with unbounded delays. The VPT in Figure 9.1, has two possible runs on words of the form c_1c^n, the lower part produces a^n as output, while the upper part produces $(ba)^n$. The size of the delay is proportional to n, it can therefore be arbitrarily large. Nevertheless this VPT is determinizable, an equivalent deterministic VPT is illustrated in Figure 9.2. As a fact, in this case the arbitrarily large delay can somehow be stored on the stack, and then it can be bridged while reading the matching returns: the output of the loop transition on q_3, resp. q_1, is delayed until the matching loop of returns.

The second reason why VPTs are in a more challenging situation, is related to the output of a transition of the equivalent deterministic VPT. For NFT, the output is simply the longest common prefix of all active runs. For VPT, we might have to look ahead for common outputs. Take for example the determinizable

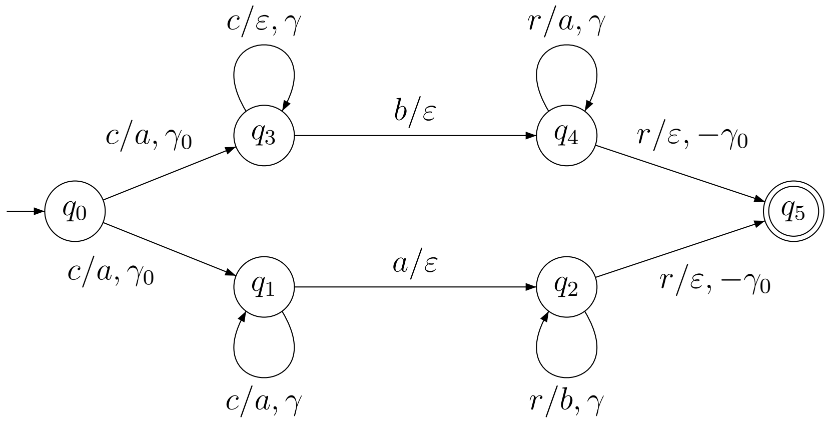


Figure 9.3: A determinizable functional VPT T_d.

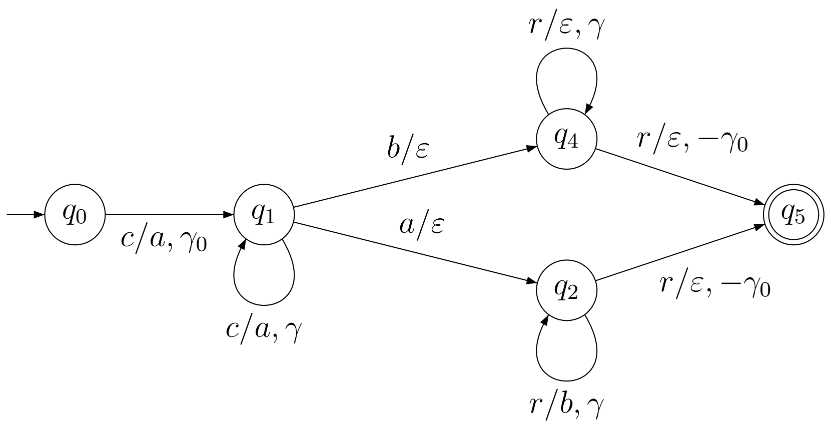


Figure 9.4: A deterministic VPT T'_d.

VPT of Figure 9.3, an equivalent deterministic VPT is illustrated Figure 9.4. For that VPT it is necessary to "advance" the output of the return loop (loop on state q_4), to the matching call loop. In that case, the output is the longest common prefix of all *accepting continuations*.

Finite valuedness of VPT. In Theorem 5.5.11, we showed that, for a fixed k, k-valuedness is decidable in co-NPTime. Some VPT might not be k-valued for any k. For example, the following transduction over $\Sigma = \{a, b\}$ is definable by an NFT but is not k-valued for any k:

$$\{(a^n, \alpha_1 \dots \alpha_n) \mid \alpha_i \in \Sigma\}$$

The finite-valuedness problem, or bounded-valuedness problem, asks, given a transducer T, whether there exists a $k \in \mathbb{N}$ such that T is k-valued.

This problem is decidable in PTIME for NFT [Web89]. The procedure relies on the detection of some patterns (reminiscent of the twinning properties) that induce unbounded valuedness. For top-down tree transducers it is also decidable, the difficult proof also relies on the detection of patterns [Sei94]. The difficult part of the proof consists in showing that the absence of these patterns implies finite-valuedness.

Equivalence of k-valued VPT. The equivalence of functional NFT transduction has been extended to k-valued NFT first in [Web93] with a double exponential procedure and then recently in [SdS10] with a single exponential bound. For VPT, we proved that the equivalence and inclusion problem of functional VPT is decidable in PTIME (See Theorem 5.4.7). However, extending these results to k-valued transducers remains challenging.

Regularity. We showed in Chapter 7, that it is decidable whether a given functional VPT transduction with a well-nested domain is equivalent to some deterministic NFT, *i.e.* whether it is bounded memory. The general problem, called the *regularity problem*, asks, given a VPT (whose domain is not necessarily well-nested) whether there exists an equivalent NFT, resp. DFT. This problem is open.

Note that, the regularity problem is decidable for deterministic pushdown *automaton* [Ste67]. As a result, as any VPA is determinizable and therefore equivalent to a deterministic PA, the regularity problem is decidable for VPA. It is however not clear how to take advantage of this result on VPA for deciding the regularity problem of VPT.

Two others related problems have been studied for VPA and can be generalized to VPT. A word is well-matched if it is well-nested and each call symbol is always matched with the same, predefined, return symbol, for instance, XML documents are examples of well-matched words. The problem studied in [SS07], resp. in [BLS06], asks, given a VPA, whether there exists an equivalent NFA on the set of well-matched, resp. well-nested, words. In other words, the problem asks, given a VPA, whether there is an equivalent NFA, assuming the words are well-matched, resp. well-nested. These problems are not equivalent to the regularity problem, since the non well-matched, resp. well-nested, words are ignored. The former problem for VPA is open, while the latter was solved for

VPA in [BLS06]. These two problems can be directly generalized to VPT: given a VPT, assuming all the input words are well-matched or well-nested, does an equivalent NFT, resp. DFT, exist ?

Lower bounds. We proved that k-valuedness and HBM are decidable for VPT in co-NPTime. However, we did not prove any lower bound. The proof of the upper bound is done through a reduction to the emptiness of reversal bounded counter machines. This later problem is co-NPTime hard (Theorem 5.5.6). Yet, it is not clear whether the ideas of this proof of hardness can be used to prove hardness of the k-valuedness problem or of the HBM problem for VPT.

Future Works

The results of this thesis may find a potential domain of application in the field of XML transformations. Visibly pushdown automata have been used to handle the validation of XML documents [PSZ11, Kum07]. Other works [Gau09, GNT09, Kum07, Cla08] used VPA to answer queries based on language such as XPath or XQuery. Those works have demonstrated that VPA are well-suited for efficiently processing XML. Further work may focus on investigating whether this proven efficiency of VPA for XML processing extends to visibly pushdown transducers.

Let us briefly discuss how our results translate in terms of XML. Visibly pushdown transducers can be used to model of XML transformations.

On the one hand, testing the equivalence of functional VPT can, for instance, be used for testing the correctness of a refactoring of an XML transformation, while on the other hand, testing inclusion could be used to test the backward compatibility of two XML transformations. Suppose that an old specification must be updated to handle new types of documents, while still handling as before the old types of documents. In that case the new XML transformation should contain the old one.

The type checking problem is clearly an XML inspired problem. Given a set of input XML documents and a set of output documents, specified for example by an input and output XML Schema [WF04], and given an XML transformation, for instance specified in a language such as XSLT, the type checking problem asks whether all valid input documents are transformed into a valid output document. VPA capture the structural part of XML Schema [PSZ11] (*i.e.* a VPA cannot test the values domain, nor the uniqueness of keys). Therefore Theorem 6.3.1 states that it is decidable whether an XML transformation, specified by a wnVPT, type-checks against the structural part of some XML Schema. With that in mind, in [PSZ11], we presented an implementation of the conversion of XML schema into equivalent VPA. This conversion procedure is a first step towards a framework based on visibly pushdown machines for type checking XML transformations.

The evaluation algorithm Eval presented in Chapter 7 is a memory efficient algorithm that can be directly applied to XML transformations specified by functional VPT. Furthermore, deciding whether such transductions are HBM or twinned makes sense when one wants to ensure that the memory used is reasonable, besides, in the case of XML, a vast majority of documents are shallow [BMV05].

Nonetheless, the expressive power of VPT is quite low with respect to what the main XML transformation languages, such as XSLT [Cla99], XQuery [RCD$^+$] or XDuce [Fri06], can do. Indeed, VPT do not have the ability to swap or duplicate parts of the document. We showed in Section 6.4, that increasing the expressiveness can be achieved by adding a macro mechanism (*i.e.* parameters) to VPT in a similar manner that it is done in macro tree or forest transducers [EV85, PS04]. With such a macro mechanism, the expressive power of VPT should be similar to that of macro tree transducers, in fact we conjecture that these models are equivalent. The macro tree transducers capture a significant part of the XSLT language [MBPS05].

We would therefore like to extend our results to macro visibly pushdown transducers in the following directions.

First, we would like to extend the evaluation algorithm Eval of Chapter 7 to handle the parameters of macro VPT. To be efficient this evaluation algorithm should perform the transformation in a streaming mode, *i.e.* by reading the input stream only once, from left to right.

Clearly, some macro VPT transductions can be memory consuming, it would therefore be interesting to extend the decision procedure of Chapter 7, that decides whether a transformation can be performed with a memory somehow bounded by the height of the input document. Compared to VPT, the additional challenge with macro VPT lies in the fact that parameters can be used to move or duplicate parts of the input document, requiring an amount of memory proportional to the size of the part that is moved or duplicated. Another additional difficulty is that there might be an unbounded number of parameters evaluating at a given time, and thus requiring a possibly unbounded amount of memory. We conjecture that the macro VPT transductions that can be evaluated with height bounded memory are exactly the HBM VPT transductions.

Finally, we also would like to look into how to extend to macro VPT the results of Chapter 5 and Chapter 6: deciding functionality, equivalence of functional VPT and type checking.

All these extensions are obviously very challenging but they have already been tackled, in part, by various related works.

Indeed, in a first trend of research, several models of tree transducers have been proposed as formal models of XML transformations [MN00, MN03, MBPS05, PS04, EHS07]. Both the expressiveness of the model and the type checking problems have been extensively investigated for these models [FH07, MNG08, MN07, MSV03]. They however do not tackle the problem of efficiently performing the transformation. A second trend of related researches looks for an efficient model for performing XML transformations [DZ09, DZ08, FN07]. Clearly, first and foremost we would have to assess the exact expressiveness of macro VPT with regards to all those existing models.

We hope that the techniques exposed in the present thesis, contributed to shedding some light on how to tackle problems related to XML transformations, and provided some interesting baseline thoughts for further work.

Appendix A

Proof of Theorem 7.5.3

In this section we prove Theorem 7.5.3. We extend the concatenation to pairs of words and denote it by $\cdot$, i.e. $(u,v)\cdot(u',v') = (uu',vv')$.

Proof of Theorem 7.5.3

Proof. We assume that T_1 is not twinned and show that T_2 is not twinned either. By definition of the TP there are two runs of the form

$$\begin{cases} (i_1,\perp) \xrightarrow{u_1|v_1} (p_1,\sigma_1) \xrightarrow{u_2|v_2} (p_1,\sigma_1\beta_1) \xrightarrow{u_3|v_3} (q_1,\sigma_1\beta_1) \xrightarrow{u_4|v_4} (q_1,\sigma_1) \\ (i_1',\perp) \xrightarrow{u_1|v_1'} (p_1',\sigma_1') \xrightarrow{u_2|v_2'} (p_1',\sigma_1'\beta_1') \xrightarrow{u_3|v_3'} (q_1',\sigma_1'\beta_1') \xrightarrow{u_4|v_4'} (q_1',\sigma_1') \end{cases}$$

such that (q,σ_1) and (q',σ_1') are co-accessible and $\Delta(v_1v_3, v_1'v_3') \neq \Delta(v_1v_2v_3v_4, v_1'v_2'v_3'v_4')$. We will prove that by pumping the loops on u_2 and u_4 sufficiently many times we will get a similar situation in T_2, proving that T_2 is not twinned. It is easy to show that there exist $k_2 > 0$, $k_1,k_3 \geq 0$, $w_i, w_i' \in \Sigma^*$, $i \in \{1,\dots,4\}$, some states $i_2,p_2,q_2,i_2',p_2',q_2'$ of T_2 and some stack contents $\sigma_2,\beta_2,\sigma_2',\gamma_2'$ of T_2 such that we have the following runs in T_2:

$$\begin{cases} (i_2,\perp) \xrightarrow{u_1u_2^{k_1}|w_1} (p_2,\sigma_2) \xrightarrow{u_2^{k_2}|w_2} (p_2,\sigma_2\beta_2) \xrightarrow{u_2^{k_3}u_3u_4^{k_3}|w_3} (q_2,\sigma_2\beta_2) \xrightarrow{u_4^{k_2}|w_4} (q_2,\sigma_2) \\ (i_2',\perp) \xrightarrow{u_1u_2^{k_1}|w_1'} (p_2',\sigma_2') \xrightarrow{u_2^{k_2}|w_2'} (p_2',\sigma_2'\beta_2') \xrightarrow{u_2^{k_3}u_3u_4^{k_3}|w_3'} (q_2',\sigma_2'\beta_2') \xrightarrow{u_4^{k_2}|w_4'} (q_2',\sigma_2') \end{cases}$$

such that (q_1,σ_1) and (q_2,σ_2) are co-accessible with the same input word u_5, and (q_1',σ_1') and (q_2',σ_2') are co-accessible with the same input word u_5'. Now for all

$i \geq 0$, we let

$$\begin{array}{ll} V^{(i)} = v_1(v_2)^{k_1+ik_2+k_3}v_3(v_4)^{k_1+ik_2+k_3} & W^{(i)} = w_1(w_2)^i w_3(w_4)^i \\ V'^{(i)} = v'_1(v'_2)^{k_1+ik_2+k_3}v'_3(v'_4)^{k_1+ik_2+k_3} & W'^{(i)} = w'_1(w'_2)^i w'_3(w'_4)^i \\ D_1(i) = \Delta(V^{(i)}, V'^{(i)}) & D_2(i) = \Delta(W^{(i)}, W'^{(i)}) \end{array}$$

In other words, $D_1(i)$ (resp. $D_2(i)$) is the delay in T_1 (resp. T_2) accumulated on the input word $u_1(u_2)^{k_1+ik_2+k_3}u_3(u_4)^{k_1+ik_2+k_3}$ by the two runs of T_1 (resp. T_2).

There is a relation between the words $V^{(i)}$ and $W^{(i)}$. Indeed, since T_1 and T_2 are equivalent and (q_1, σ_1) and (q_2, σ_2) are both co-accessible by the same input word, for all $i \geq 1$, either $V^{(i)}$ is a prefix of $W^{(i)}$ or $W^{(i)}$ is a prefix of $V^{(i)}$, i.e. there exist $X \in \Sigma^*$ such that: for all $i \geq 1$, $V^{(i)} = W^{(i)}X$ or for all $i \geq 1$ $V^{(i)}X = W^{(i)}$. Similarly, there exists $X' \in \Sigma^*$ such that for all $i \geq 1$, $V'^{(i)} = W'^{(i)}X'$ or for all $i \geq 1$, $V'^{(i)}X' = W'^{(i)}$.

We now prove the following key result: for all $i, j \geq 1$,

$$D_1(i) \neq D_1(j) \implies D_2(i) \neq D_2(j)$$

We consider two cases (the other ones being symmetric):

- for all $\ell \geq 1$, $V^{(\ell)} = W^{(\ell)}X$ and $V'^{(\ell)} = W'^{(\ell)}X'$. Then we have:

$$\begin{array}{llcll} & \Delta(V^{(i)}, V'^{(i)}) & \neq & \Delta(V^{(j)}, V'^{(j)}) & \\ \Rightarrow & \Delta(W^{(i)}X, W'^{(i)}X') & \neq & \Delta(W^{(j)}X, W'^{(j)}X') & \\ \Rightarrow & \Delta(\Delta(W^{(i)}, W'^{(i)}) \cdot (X, X')) & \neq & \Delta(\Delta(W^{(j)}, W'^{(j)}) \cdot (X, X')) & \text{(Lemma 7.3.10)} \\ \Rightarrow & \Delta(W^{(i)}, W'^{(i)}) & \neq & \Delta(W^{(j)}, W'^{(j)}) & \end{array}$$

- for all $\ell \geq 1$, $V^{(\ell)} = W^{(\ell)}X$ and $V'^{(\ell)}X' = W'^{(\ell)}$. Then we have:

$$\begin{array}{llcll} & \Delta(V^{(i)}, V'^{(i)}) & \neq & \Delta(V^{(j)}, V'^{(j)}) & \\ \Rightarrow & \Delta(W^{(i)}X, V'^{(i)}) & \neq & \Delta(W^{(j)}X, V'^{(j)}) & \\ \Rightarrow & \Delta(\Delta(W^{(i)}, V'^{(i)}) \cdot (X, \epsilon)) & \neq & \Delta(\Delta(W^{(j)}, V'^{(j)}) \cdot (X, \epsilon)) & \text{(Lemma 7.3.10)} \\ \Rightarrow & \Delta(W^{(i)}, V'^{(i)}) & \neq & \Delta(W^{(j)}, V'^{(j)}) & \\ \Rightarrow & \Delta(\Delta(W^{(i)}, V'^{(i)}) \cdot (\epsilon, X')) & \neq & \Delta(\Delta(W^{(j)}, V'^{(j)}) \cdot (\epsilon, X')) & \text{(Lemma A.0.1)} \\ \Rightarrow & \Delta(W^{(i)}, V'^{(i)}X') & \neq & \Delta(W^{(j)}, V'^{(j)}X') & \text{(Lemma 7.3.10)} \\ \Rightarrow & \Delta(W^{(i)}, W'^{(i)}) & \neq & \Delta(W^{(j)}, W'^{(j)}) & \end{array}$$

Now by Lemma A.0.3, since $\Delta(v_1v_3, v'_1v'_3) \neq \Delta(v_1v_2v_3v_4, v'_1v'_2v'_3v'_3)$, there exists $i_0 \geq 1$ such that for all $i, j \geq i_0$, if $i \neq j$ then $\Delta(v_1(v_2)^i v_3(v_4)^i, v'_1(v'_2)^i v'_3(v'_4)^i) \neq \Delta(v_1(v_2)^j v_3(v_4)^j, v'_1(v'_2)^j v'_3(v'_4)^j)$. In particular since $k_2 \geq 1$, we have $D_1(i) \neq D_1(j)$ for all $i, j \geq i_0$ and $i \neq j$. By the last intermediate result, we get $D_2(i_0) \neq D_2(i_0 + 1)$. Therefore the TP does not hold for T_2. $\square$

Lemma A.0.1. *For all $u, u', v, v', w, w' \in \Sigma^*$, we have*

$$\Delta(\Delta(u,u') \cdot (w,w')) = \Delta(\Delta(v,v') \cdot (w,w')) \text{ iff } \Delta(u,u') = \Delta(v,v')$$

Proof. There exists A, B, C, D and X, Y such that:

$$\Delta(u,u') = (A,B) \quad \Delta(v,v') = (C,D) \quad u = XA \quad u' = XB \quad v = YC \quad v' = YD$$

Let also E, F, G, H such that $\Delta(Aw, Bw') = (E,F)$ and $\Delta(Cw, Dw') = (G,H)$. Clearly, if $A = C$ and $B = D$, we have $E = G$ and $F = H$.

Conversely, suppose that $A \neq C$ (the case $B \neq D$ is symmetric). We show that $E \neq G$ or $F \neq H$. By definition of the delay, we know that $A \wedge B = \epsilon$, and $C \wedge D = \epsilon$. Therefore we have the following cases, for some words A', B', C', D' and letters a, b, c, d such that $a \neq b$ and $c \neq d$:

1. $A = aA'$ and $B = bB'$, $C = cC'$ and $D = dD'$ for some A', B', C', D'. Therefore $\Delta(Aw, Bw') = (Aw, Bw') = (E,F)$ and $\Delta(Cw, Dw') = (Cw, Dw') = (G,H)$. Since $A \neq C$, we get $E \neq G$;

2. $A = aA'$ and $B = bB'$, and $D = \epsilon$. Therefore $\Delta(Aw, Bw') = (Aw, Bw') = (E,F)$ and $\Delta(Cw, Dw') = \Delta(Cw, w') = (G,H)$. We have necessarily $|H| \leq |w'|$. Since $B \neq \epsilon$, we have $|F| = |Bw'| > |w'| \geq |H|$. Therefore $F \neq H$;

3. $A = aA'$ and $B = bB'$ and $C = \epsilon$. We can apply the same argument as case 2;

4. $A = \epsilon$ and $C = cC'$ and $D = dD'$. This case is symmetric to case 2;

5. $B = \epsilon$ and $C = cC'$ and $D = dD'$. This case is symmetric to case 2;

6. $A = \epsilon$ and $C = \epsilon$. This case is not possible since we have assumed $A \neq C$;

7. $A = \epsilon$ and $D = \epsilon$ ($C \neq \epsilon$). We have $\Delta(Aw, Bw') = \Delta(w, Bw')$ and $\Delta(Cw, Dw') = \Delta(Cw, w')$. Suppose that $E = G$ and $F = H$. Then there exists Z, Z' such that $w = ZE$, $Bw' = ZF$, $Cw = Z'E$ and $w' = Z'F$. Therefore $Bw' = BZ'F = ZF$, and $BZ' = Z$, so that $w = BZ'E$ and $Cw = CBZ'E = Z'E$, i.e. $CB = \epsilon$, which contradicts $C \neq \epsilon$;

8. $B = \epsilon$ and $C = \epsilon$. This case is symmetric to the previous case;

9. $B = \epsilon$ and $D = \epsilon$. Then we have $\Delta(Aw, Bw') = \Delta(Aw, w')$ and $\Delta(Cw, Dw') = \Delta(Cw, w')$. Again suppose that $E = G$ and $F = H$, therefore there exists Z, Z' such that $Aw = ZE$, $w' = ZF$, $Cw = Z'E$ and $w' = Z'F$. Therefore $Z = Z'$, which implies $Aw = Cw$. This contradicts $A \neq C$.

□

Lemma A.0.2. *Let $v_1, v_2, v_3, v_4, w_1, w_2, w_3, w_4 \in \Sigma^*$ and for all $i \geq 0$, let*

$$V^{(i)} = v_1(v_2)^i v_3(v_4)^i \qquad\qquad W^{(i)} = w_1(w_2)^i w_3(w_4)^i.$$

If there exist $K \geq 0$ and $X \in \Sigma^*$ such that for all $i \geq K$, $V^{(i)} = W^{(i)}X$, then for all $i \geq 0$, $V^{(i)} = W^{(i)}X$.

Proof. First note that we have $|v_2v_4| = |w_2w_4|$, this is a straight consequence from the fact that for all $i \geq K$, $V^{(i)} = W^{(i)}X$. We consider two cases:

- $|v_2| \neq |v_4|$: in that case we can show that the primitive roots of v_2, v_4, w_2, w_4 are conjugate (see for example [FRR+10a]) and therefore have the same length. Therefore we can apply Theorem 1 of [HK07] which yields the result.

- $|v_2| = |v_4|$: in that case we also have $|w_2| = |w_4|$, and suppose $|v_1| \geq |w_1|$ (the case $|v_1| < |w_1|$ is similar). There exists $Y \in \Sigma^*$, such that for any $i \geq K$ we have $v_1{v_2}^i = w_1{w_2}^iY$ and $Yv_3{v_4}^i = w_3{w_4}^iX$. Therefore we can apply Theorem 2 of [HK07] which shows that these equalities hold for any i. For any i we have $V^{(i)} = v_1{v_2}^iv_3{v_4}^i = w_1{w_2}^iYv_3{v_4}^i = w_1{w_2}^iw_3{w_4}^iX = W^{(i)}X$.

□

Lemma A.0.3. *Let $v_1, v_2, v_3, v_4, w_1, w_2, w_3, w_4 \in \Sigma^*$ and for all $i \geq 0$, let*

$$V^{(i)} = v_1(v_2)^i v_3(v_4)^i \qquad\qquad W^{(i)} = w_1(w_2)^i w_3(w_4)^i.$$

If $\Delta(V^{(0)}, W^{(0)}) \neq \Delta(V^{(1)}, W^{(1)})$, then there exists $i_0 \geq 1$ such that for all $i, j \geq i_0$, if $i \neq j$ then $\Delta(V^{(i)}, W^{(i)}) \neq \Delta(V^{(j)}, W^{(j)})$.

Proof. First note that since $\Delta(V^{(0)}, W^{(0)}) \neq \Delta(V^{(1)}, W^{(1)})$, we clearly have $|v_2v_4| \neq \epsilon$ or $|w_2w_4| \neq \epsilon$, We write $u \preceq v$ if u is a prefix of v, and $u||v$ if u and v are incomparable, i.e. $u \wedge v = \epsilon$. We consider several cases:

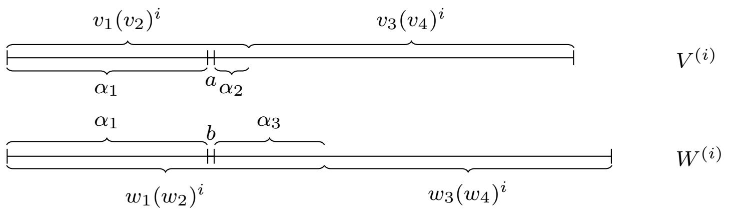
1. there is $K \geq 1$ such that for all $i \geq K$, $V^{(i)} \preceq W^{(i)}$ or $W^{(i)} \preceq V^{(i)}$. Consider the lengths of $V^{(i)}$ and $W^{(i)}$. When K is large enough, one of $V^{(i)}$ and $W^{(i)}$ is always the prefix of the other, for $i \geq K$: $V^{(i)} \preceq W^{(i)}$ if $|v_2v_4| \leq |w_2w_4|$, and $W^{(i)} \preceq V^{(i)}$ otherwise. Let us assume that $|v_2v_4| \leq |w_2w_4|$, i.e., for all $i \geq K$, there exists X_i such that $W^{(i)} = V^{(i)}X_i$. The other case is symmetric. We have $|W^{(i+1)}| - |W^{(i)}| = |w_2w_4| = |V^{(i+1)}| + |X_{i+1}| - |V^{(i)}| - |X_i|$, i.e. $|w_2w_4| = |V^{(i)}| + |v_2v_4| + |X_{i+1}| - |V^{(i)}| - |X_i|$, i.e. $|X_{i+1}| - |X_i| = |w_2w_4| - |v_2v_4|$. We again consider several cases:

   1.1 $|w_2w_4| > |v_2v_4|$. We have $|X_K| < |X_{K+1}| < |X_{K+2}| \ldots$, and by definition of the delay, $\Delta(V^{(i)}, W^{(i)}) = (\epsilon, X_i)$. Therefore the delay always increases in size as i icreases and we get the result;

   1.2 $|w_2w_4| = |v_2v_4|$. We show that this case is not possible. Indeed, it implies that $|X_K| = |X_{K+1}| = |X_{K+2}| \ldots.$ Therefore by definition of $V^{(i)}$ and $W^{(i)}$, there exists $K' \geq K$ and $X \in \Sigma^*$ such that $X = X_{K'} = X_{K'+1} = X_{K'+2} \ldots.$ By Lemma A.0.2, we get $W^{(0)} = V^{(0)}X$ and $W^{(1)} = V^{(1)}X$, which contradicts $\Delta(W^{(0)}, V^{(0)}) \neq \Delta(W^{(1)}, V^{(1)})$.

2. for all $K \geq 1$, there is $i \geq K$ such that $V^{(i)}||W^{(i)}$. We show in this case that one of the two components of the delay always increases in size when i increases. We consider several cases depending on where the first difference between $V^{(i)}$ and $W^{(i)}$ occurs. The cases we consider also depend on K. In particular, by taking a large K it can reduce the number of cases we have to consider. For some $\alpha_1, \alpha_2, \alpha_3 \in \Sigma^*$ and $a, b \in \Sigma$ such that $a \neq b$, and for a K large enough, one of the following condition holds:

   2.1 there is $i \geq K$ such that $v_1(v_2)^i = \alpha_1 a \alpha_2$ and $w_1(w_2)^i = \alpha_1 b \alpha_3$, as illustrated below.

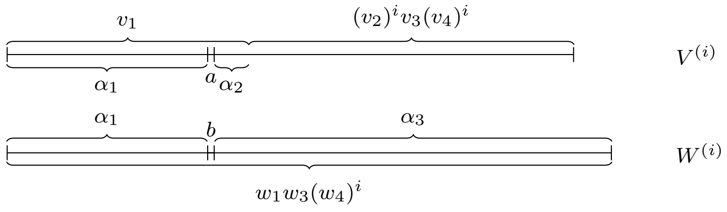


   Then for all $j \geq i$:

$$\Delta(V^{(j)}, W^{(j)}) = (a\alpha_2(v_2)^{j-i}v_3(v_4)^j, b\alpha_3(w_2)^{j-i}w_3(w_4)^j)$$

Since $|v_2v_4| \neq \epsilon$ or $|w_2w_4| \neq \epsilon$, some of the two components of the delays is always increasing in size as j icreases, which proves the result;

2.2 $w_2 = \epsilon$ and $v_1 = \alpha_1 a \alpha_2$ and there is $i \geq K$ such that $w_1 w_3 (w_4)^i = \alpha_1 b \alpha_3$, as illustrated below.

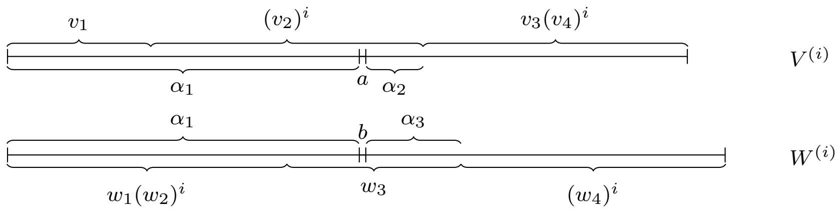


Then for all $j \geq i$:

$$\Delta(V^{(j)}, W^{(j)}) = (a\alpha_1 v_2^j v_3 (v_4)^j, b\alpha_3 (w_4)^{j-i})$$

Since $v_2v_4 \neq \epsilon$ or $w_4 \neq \epsilon$, one of the two components of the delays is always increaing in size;

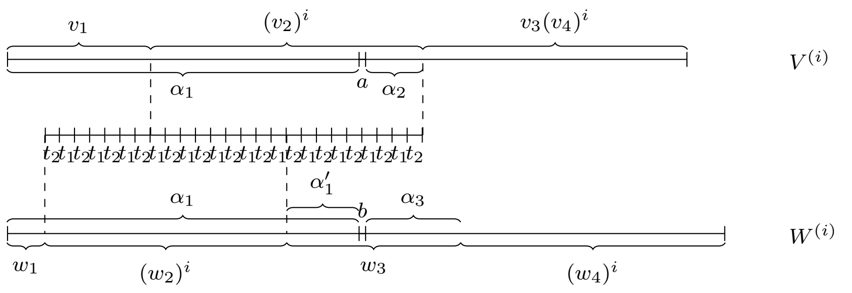
2.3 there is $i \geq K$ such that $v_1(v_2)^i = \alpha_1 a \alpha_2$ and $w_1(w_2)^i w_3 = \alpha_1 b \alpha_3$, with $|\alpha_1| \geq |v_1|$ and $|\alpha_1| \geq |w_1(w_2)^i|$ (otherwise it is case 2.1). We also assume that $w_2 \neq \epsilon$ (otherwise it can be proved similarly as case 2.1). Therefore $v_2 \neq \epsilon$. This case is illustrated below:



We have taken K large enough, so that v_2^i and w_2^i have a common factor of length at least $|v_2| + |w_2|$. The strong theorem of Fine and Wilf [Lot97] implies that the primitive roots of v_2 and w_2 are conjugate: there are $t_1, t_2 \in \Sigma^*$ and $n, p \geq 1$ such that $v_2 = (t_1t_2)^n$ and $w_2 = (t_2t_1)^p$. It can be shown (see for instance [FRR$^+$10a]) that we can choose t_1 and t_2 such that there exist $k_1, k_2 \geq 0$ verifying:

$$v_1 = (v_1 \wedge w_1)(t_2t_1)^{k_1} t_2 \qquad w_1 = (v_1 \wedge w_1)(t_2t_1)^{k_2}$$

Let α_1' be such that $\alpha_1 = w_1(w_2)^i \alpha_1'$. There exist k_3, X and Y such that $\alpha_1' = (t_2t_1)^{k_3} X$ with $t_2t_1 = XaY$, and $w_3 = \alpha_1' b \alpha_3$, as illustrated below (we assume that $|w_1| < |v_1|$ on the picture).



We have for all $j \geq i$:

$$\begin{aligned}
& \Delta(V^{(j)}, W^{(j)}) \\
= \; & \Delta(v_1(v_2)^j v_3(v_4)^j, w_1(w_2)^j w_3(w_4)^j) \\
= \; & \Delta((v_1 \wedge w_1)(t_2t_1)^{k_1} t_2(t_1t_2)^{jn} v_3(v_4)^j, w_1(w_2)^j w_3(w_4)^j) \\
= \; & \Delta((v_1 \wedge w_1)t_2(t_1t_2)^{k_1+jn} v_3(v_4)^j, w_1(w_2)^j w_3(w_4)^j) \\
= \; & \Delta((v_1 \wedge w_1)t_2(t_1t_2)^{k_1+jn} v_3(v_4)^j, (v_1 \wedge w_1)(t_2t_1)^{k_2}(w_2)^j w_3(w_4)^j) \\
= \; & \Delta(t_2(t_1t_2)^{k_1+jn} v_3(v_4)^j, (t_2t_1)^{k_2+jp} w_3(w_4)^j) \\
= \; & \Delta((t_2t_1)^{k_1+jn} t_2 v_3(v_4)^j, (t_2t_1)^{k_2+jp} w_3(w_4)^j)
\end{aligned}$$

We now consider the following subcases:

2.3.1 $|v_2| > |w_2|$. As a consequence, $n > p$, and we can assume that K is big enough, so that $k_1 - k_2 + j(n-p) > 0$. We get:

$$\begin{aligned}
\Delta(V^{(j)}, W^{(j)}) & = \Delta((t_2t_1)^{k_1-k_2+j(n-p)} t_2 v_3(v_4)^j, w_3(w_4)^j) \\
& = \Delta((t_2t_1)^{k_1-k_2+j(n-p)} t_2 v_3(v_4)^j, \alpha_1' b \alpha_3(w_4)^j) \\
& = \Delta((t_2t_1)^{k_1-k_2+j(n-p)} t_2 v_3(v_4)^j, (t_2t_1)^{k_3} X b \alpha_3(w_4)^j)
\end{aligned}$$

We can take j such that $k_1 - k_2 + j(n-p) > k_3$, and therefore we finally have:

$$\begin{aligned}
& \Delta(V^{(j)}, W^{(j)}) \\
= \; & \Delta((t_2t_1)^{k_1-k_2+j(n-p)-k_3} t_2 v_3(v_4)^j, X b \alpha_3(w_4)^j) \\
= \; & \Delta(XaY(t_2t_1)^{k_1-k_2+j(n-p)-k_3-1} t_2 v_3(v_4)^j, X b \alpha_3(w_4)^j) \\
= \; & (aY(t_2t_1)^{k_1-k_2+j(n-p)-k_3-1} t_2 v_3(v_4)^j, b \alpha_3(w_4)^j)
\end{aligned}$$

Since $t_2t_1 \neq \epsilon$, we get that the first component of the delay always increases in size when j increases;

2.3.2 $|v_2| < |w_2|$. We can take K large enough such that this case never happens, i.e. $(v_2)^i$ and w_3 do not overlap.

2.3.3 $|v_2| = |w_2|$. Therefore $n = p$, and we have for all $j \geq i$:

$$\Delta(V^{(j)}, W^{(j)}) = \Delta((t_2t_1)^{k_1}t_2v_3(v_4)^j, (t_2t_1)^{k_2}w_3(w_4)^j)$$

As case 2.3.1, since by hypothesis there is an overlap between $(v_2)^i$ and w_3, we have $k_1 > k_2 + k_3$, and we get:

$$\begin{array}{rcl}\Delta(V^{(j)}, W^{(j)}) & = & \Delta((t_2t_1)^{k_1-k_2}t_2v_3(v_4)^j, w_3(w_4)^j) \\ & = & \Delta((t_2t_1)^{k_1-k_2}t_2v_3(v_4)^j, (t_2t_1)^{k_3}Xb\alpha_3(w_4)^j) \\ & = & \Delta((t_2t_1)^{k_1-k_2-k_3}t_2v_3(v_4)^j, Xb\alpha_3(w_4)^j) \\ & = & \Delta(XaY(t_2t_1)^{k_1-k_2-k_3-1}t_2v_3(v_4)^j, Xb\alpha_3(w_4)^j) \\ & = & (aY(t_2t_1)^{k_1-k_2-k_3-1}t_2v_3(v_4)^j, b\alpha_3(w_4)^j)\end{array}$$

Therefore if $v_4 \neq \epsilon$ and $w_4 \neq \epsilon$, we are done as one of the two components of the delay will increase in size when j increases. If $v_4 = w_4 = \epsilon$ we can explicitly give the form of $\Delta(V^{(0)}, W^{(0)})$ and $\Delta(V^{(1)}, W^{(1)})$:

$$\begin{array}{rcl}\Delta(V^{(0)}, W^{(0)}) & = & \Delta((t_2t_1)^{k_1}t_2v_3, (t_2t_1)^{k_2+k_3}Xb\alpha_3) \\ & = & \Delta((t_2t_1)^{k_1-k_2-k_3}t_2v_3, Xb\alpha_3) \\ & = & \Delta((t_2t_1)^{k_1}t_2(t_1t_2)^n v_3, (t_2t_1)^{k_2+k_3+n}Xb\alpha_3) \\ & = & \Delta(V^{(1)}, W^{(1)})\end{array}$$

This is excluded by hypothesis, so this case is not possible.

2.4 the other cases (the first difference occurs between $(v_2)^i$ and $(w_4)^i$, or between v_3 and w_3, or between v_3 and $(w_4)^i$, or between $(v_4)^i$ and $(w_4)^i$) are proved similarly as case 2.3 by decomposing the words as power of their primitive roots. For instance, if the first difference occurs between v_3 and w_3, then either $v_2 = w_2 = \epsilon$ and it is the same as case 2.1, or $v_2 \neq \epsilon$ and $w_2 = \epsilon$ but we can take K large enough so that this case is impossible, or $v_2 \neq \epsilon$ and $w_2 \neq \epsilon$. In this latter case we can take K large enough so that the primitive roots of v_2 and w_2 are conjugate. We have again to distinguish several cases on the relative lengths of v_2 and w_2 (as for 2.3) but the proofs are similar. Similar techniques were already applied to prove that functionality is decidable for VPTs [FRR+10a].

$\square$

Bibliography

[ABB97] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and pushdown automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of formal languages, vol. 1*, pages 111–174. Springer-Verlag, 1997.

[AD11] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. Available on: `http://www.cis.upenn.edu/~alur/`, 2011. Submitted.

[ALH04] Lauren Wood Gavin Nicol Jonathan Robie Mike Champion Steve Byrne Arnaud Le Hors, Philippe Le Hégaret. Document object model (dom), W3C recommendation, 2004.

[Alu07] Rajeev Alur. Marrying words and trees. In *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2007*, volume 5140 of *LNCS*, pages 233–242, 2007.

[Alu11] Rajeev Alur. Nested words and visibly pushdown languages `http://www.cis.upenn.edu/~alur/nw.html`, June 2011.

[AM04] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *36th ACM symposium on Theory of computing, STOC 2004*, pages 202–211, 2004.

[AM06] Rajeev Alur and Parthasarathy Madhusudan. Adding nesting structure to words. In *10th International Conference on Developments in Language Theory, DLT 2006*, pages 1–13, 2006.

[AM09] Rajeev Alur and Parthasarathy Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.

[BC02] Marie-Pierre Béal and Olivier Carton. Determinization of transducers over finite and infinite words. *Theoretical Computer Science*, 289(1):225–251, 2002.

[BCPS03] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch. Squaring transducers: an efficient procedure for deciding functionality and sequentiality. *Theoretical Computer Science*, 292(1):45–63, 2003.

[Ber09] Jean Berstel. Transductions and context-free languages `http://www-igm.univ-mlv.fr/~berstel/`, December 2009.

[BG06] Andreas Blass and Yuri Gurevich. A note on nested words. Technical Report MSR-TR-2006-139, Microsoft Research, October 2006.

[BJ07] Michael Benedikt and Alan Jeffrey. Efficient and expressive tree filters. In *27th Conference on Foundations of Software Technology and Theoretical Computer Science, FST TCS 2007*, volume 4855 of *LNCS*, pages 461–472, 2007.

[BLS06] Vince Bárány, Christof Löding, and Olivier Serre. Regularity problems for visibly pushdown languages. In *23rd Annual Symposium on Theoretical Aspects of Computer Science, STACS 2006*, pages 420–431, 2006.

[BMV05] Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the XML web: Gathering statistics from an XML sample. *14th international conference on World Wide Web, WWW 2005*, 8(4):413–438, 2005.

[BYFJ05] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over XML streams. In *24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2005*, pages 216–227, 2005.

[Cau06] Didier Caucal. Synchronization of pushdown automata. In *10th International Conference on Developments in Language Theory, DLT 2006*, pages 120–132, 2006.

[CDG$^+$07] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2007.

[CH08] Didier Caucal and Stéphane Hassen. Synchronization of grammars. In *3rd international conference on Computer science: theory and applications, CSR 2008*, pages 110–121, 2008.

[Cho77] Christian Choffrut. Une caractérisation des fonctions séquentielles et des fonctions sous-sq́uentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325–337, 1977.

[Cla99] James Clark. XSL Transformations (XSLT) version 1.0, W3C recommendation, 1999.

[Cla08] Robert Clark. Querying streaming xml using visibly pushdown automata. Technical Report 2008-3008, ideals, 2008.

[CRT11] Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. A polynomial procedure for trimming visibly pushdown automata. Technical Report hal-00606778, HAL, 2011.

[DZ08] Jana Dvoráková and Filip Zavoral. Xord: An implementation framework for efficient xslt processing. In *2nd International Symposium on Intelligent Distributed Computing, IDC 2008*, pages 95–104, 2008.

[DZ09] Jana Dvoráková and Filip Zavoral. Using input buffers for streaming xslt processing. In *First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA 2009*, pages 50–55, 2009.

[EHS07] Joost Engelfriet, Hendrik Jan Hoogeboom, and Bart Samwel. Xml transformation by tree-walking transducers with invisible pebbles. In *26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2007*, pages 63–72, 2007.

[Eil74] Samuel Eilenberg. *Automata, Languages, and Machines.* Academic Press, 1974.

[EM65] Calvin C. Elgot and Jorge E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9:47–68, 1965.

[EM03] Joost Engelfriet and Sebastian Maneth. Macro tree translations of linear size increase are MSO definable. *SIAM Journal on Computing*, 32:950–1006, 2003.

[Eng77] Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303, 1977.

[Eng78] Joost Engelfriet. On tree transducers for partial functions. *Information Processing Letters*, 7(4):170–172, 1978.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985.

[FH07] Alain Frisch and Haruo Hosoya. Towards practical typechecking for macro tree transducers. In *15th International Workshop on Database Programming Languages, DBPL 2007*, pages 246–260, 2007.

[FN07] Alain Frisch and Keisuke Nakano. Streaming xml transformation using term rewriting. In *5th ACM SIGPLAN Workshop on Programming Language Technologies for XML, PLAN-X 2007*, pages 2–13, 2007.

[Fri06] Alain Frisch. Ocaml + xduce. In *11th ACM SIGPLAN international conference on Functional programming, ICFP 2006*, pages 192–200, 2006.

[FRR$^+$10a] Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. On functionality of visibly pushdown transducers. *CoRR*, abs/1002.1443, 2010.

[FRR$^+$10b] Emmanuel Filiot, Jean-François Raskin, Pierre-Alain Reynier, Frédéric Servais, and Jean-Marc Talbot. Properties of visibly pushdown transducers. In *35th International Symposium on Mathematical Foundations of Computer Science, MFCS 2010*, pages 355–367, 2010.

[Gau09] Olivier Gauwin. *Streaming Tree Automata and XPath.* PhD thesis, Université Lille 1, 2009.

[GHS09] Martin Grohe, Andre Hernich, and Nicole Schweikardt. Lower bounds for processing data with few random accesses to external memory. *Journal of the ACM*, 56(3):12:1–12:58, 2009.

[GI83] Eitan M. Gurari and Oscar H. Ibarra. A note on finite-valued and finitely ambiguous transducers. *Theory of Computing Systems*, 16(1):61–66, 1983.

[GNR08] Olivier Gauwin, Joachim Niehren, and Yves Roos. Streaming tree automata. *Information Processing Letters*, 109(1):13–17, 2008.

[GNT09] O. Gauwin, J. Niehren, and S. Tison. Earliest query answering for deterministic nested word automata. In *17th International Symposium on Fundamentals of Computation Theory, FCT 2009*, volume 5699 of *LNCS*, pages 121–132, 2009.

[Gri68] Timothy V. Griffiths. The unsolvability of the equivalence problem for lambda-free nondeterministic generalized machines. *Journal of the ACM*, 15(3):409–413, 1968.

[Har78] Michael A. Harrison. *Introduction to formal language theory.* Addison-Wesley, 1978.

[HIKS02] Tero Harju, Oscar H. Ibarra, Juhani Karhumaki, and Arto Salomaa. Some decision problems concerning semilinearity and commutation. *Journal of Computer and System Sciences*, 65:278–294, 2002.

[HK99] Ismo Hakala and Juha Kortelainen. On the system of word equations $x_0u_1^ix_1u_2^ix_2u_3^ix_3 = y_0v_1^iy_1v_2^iy_2v_3^iy_3 (i = 0, 1, 2, \ldots)$ in a free monoid. *Theoretical Computer Science*, 225(1-2):149–161, 1999.

[HK07] Stepan Holub and Juha Kortelainen. On systems of word equations with simple loop sets. *Theoretical Computer Science*, 380(3):363–372, 2007.

[HL11] Matthew Hague and Anthony Widjaja Lin. Model checking recursive programs with numeric data types. In *23rd International Conference on Computer Aided Verification, CAV 2011*, pages 743–759, 2011.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley, 2003.

[Iba78] Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, 1978.

[IK86] Karel Culik II and Juhani Karhumäki. The equivalence of finite valued transducers (on hdtol languages) is decidable. In *11th International Symposium on Mathematical Foundations of Computer Science, MFCS 1986*, pages 264–272, 1986.

[KM10] Christian Konrad and Frédéric Magniez. The streaming complexity of validating XML documents. Technical Report 1012.3311, arXiv, 2010.

[KS07] Christoph Koch and Stefanie Scherzinger. Attribute grammars for scalable query processing on XML streams. *International Journal on Very Large Data Bases*, 16(3):317–342, 2007.

[Kum07] Visibly pushdown automata for streaming XML. In *16th international conference on World Wide Web, WWW 2007*, pages 1053–1062, 2007.

[Lot97] M Lothaire. *Combinatorics on words.* Cambridge University Press, 1997.

[Man03] Sebastian Maneth. The macro tree transducer hierarchy collapses for functions of linear size increase. In *23st Conference on Foundations of Software Technology and Theoretical Computer Science, FST TCS 2003*, pages 326–337, 2003.

[MBPS05] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML type checking with macro tree transducers. In *24th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS 2005*, pages 283–294, 2005.

[Min67] Marvin Lee Minsky. *Finite and Infinite Machines.* Prentice-Hall, 1967.

[MN00] Sebastian Maneth and Frank Neven. Structured document transformations based on XSL. In *7th International Workshop on Database Programming Languages, DBPL 1999*, volume 1949 of *LNCS*, pages 80–98, 2000.

[MN03] Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *9th International Conference on Database Theory, ICDT 2003*, pages 64–78, 2003.

[MN04] Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple xml transformations. In *23th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2004*, pages 23–34, 2004.

[MN05] Wim Martens and Frank Neven. On the complexity of typechecking top-down xml transformations. *Theoretical Computer Science*, 336(1):153–180, 2005.

[MN07] Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple xml transformations. *Journal of Computer and System Sciences*, 73(3):362–390, 2007.

[MNG08] Wim Martens, Frank Neven, and Marc Gyssens. Typechecking top-down xml transformations: Fixed input or output schemas. *Information and Computation*, 206(7):806–827, 2008.

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

[MV09] P. Madhusudan and M. Viswanathan. Query automata for nested words. In *34th International Symposium on Mathematical Foundations of Computer Science, MFCS 2009*, volume 5734 of *LNCS*, pages 561–573. Springer Berlin / Heidelberg, 2009.

[NS07] Dirk Nowotka and Jirí Srba. Height-deterministic pushdown automata. In *32th International Symposium on Mathematical Foundations of Computer Science, MFCS 2007*, pages 125–134, 2007.

[Pap81] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28:765–768, October 1981.

[Pla94] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *Second Annual European Symposium on Algorithms, ESA 1994*, pages 460–470, 1994.

[Pos46] Emil Post. A Variant of a Recursively Unsolvable Problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

[PS04] Thomas Perst and Helmut Seidl. Macro forest transducers. *Information Processing Letters*, 89(3):141–149, 2004.

[PSZ11] François Picalausa, Frédéric Servais, and Esteban Zimányi. Xevolve: an xml schema evolution framework. In *26th ACM Symposium on Applied Computing, SAC 2011*, pages 1645–1650, 2011.

[Ram10] Prakash Ramanan. Memory lower bounds for XPath evaluation over XML streams. *Journal of Computer and System Sciences*, In Press, Corrected Proof:–, 2010.

[RCD$^+$] Jonathan Robie, Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, and JÈrÙme SimÈon. XQuery Update Facility 1.0, W3C Recommendation 17 March 2011.

[RS08] Jean-François Raskin and Frédéric Servais. Visibly pushdown transducers. In *35th International Colloquium on Automata, Languages and Programming, ICALP 2008*, volume 5126 of *LNCS*, pages 386–397, 2008.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, 2009.

[Sch75] Marcel Paul Schützenberger. Sur les relations rationnelles. In *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 209–213, 1975.

[Sch76] Marcel Paul Schützenberger. Sur les relations rationnelles entre monoides libres. *Theoretical Computer Science*, 3(2):243–259, 1976.

[SdS08] Jacques Sakarovitch and Rodrigo de Souza. On the decidability of bounded valuedness for transducers. In *33th International Symposium on Mathematical Foundations of Computer Science, MFCS 2008*, pages 588–600, 2008.

[SdS10] Jacques Sakarovitch and Rodrigo de Souza. Lexicographic decomposition of k -valued transducers. *Theory of Computing Systems*, 47(3):758–785, 2010.

[Sei92] Helmut Seidl. Single-valuedness of tree transducers is decidable in polynomial time. *Theoretical Computer Science*, 106(1):135–181, 1992.

[Sei94] Helmut Seidl. Equivalence of finite-valued tree transducers is decidable. *Mathematical Systems Theory*, 27(4):285–346, 1994.

[Sén97] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *24th International Colloquium on Automata, Languages and Programming, ICALP 1997*, pages 671–681, 1997.

[Sén99] Gérard Sénizergues. T(A) = T(B)? In *26th International Colloquium on Automata, Languages and Programming, ICALP 1999*, volume 1644 of *LNCS*, pages 665–675, 1999.

[SLLN09] Slawomir Staworko, Grégoire Laurence, Aurélien Lemay, and Joachim Niehren. Equivalence of deterministic nested word to word transducers. In *17th International Symposium on Fundamentals of Computation Theory, FCT 2009*, volume 5699 of *LNCS*, pages 310–322, 2009.

[SS07] Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming xml documents against dtds. In *13th International Conference on Database Theory, ICDT 2007*, pages 299–313, 2007.

[Ste67] Richard Edwin Stearns. A regularity test for pushdown machines. *Information and Control*, 11(3):323–340, 1967.

[SV02] Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *21th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002*, pages 53–64, 2002.

[Tan09] Nguyen Van Tang. A tighter bound for the determinization of visibly pushdown automata. In *11th International Workshop on Verification of Infinite-State Systems, INFINITY 2009*, pages 62–76, 2009.

[TVY08] Alex Thomo, Srinivasan Venkatesh, and Ying Ying Ye. Visibly pushdown transducers for approximate validation of streaming XML. In *5th international conference on Foundations of information and knowledge systems, FoIKS 2008*, pages 219–238, 2008.

[VSS05] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. On the complexity of equational horn clauses. In *20th International*

Conference on Automated Deduction, CADE 2005, volume 3632 of *LNCS*, pages 337–352, 2005.

[Web88] Andreas Weber. A decomposition theorem for finite-valued tranducers and an application to the equivalence problem. In *13th International Symposium on Mathematical Foundations of Computer Science, MFCS 1988*, pages 552–562, 1988.

[Web89] Andreas Weber. On the valuedness of finite transducers. *Acta Informatica*, 27(8):749–780, 1989.

[Web93] Andreas Weber. Decomposing finite-valued transducers and deciding their equivalence. *SIAM Journal on Computing*, 22(1):175–202, 1993.

[WF04] Priscilla Walmsley and David C. Fallside. XML Schema Part 0: Primer Second Edition. W3C recommendation, W3C, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`.

[WK95] Andreas Weber and Reinhard Klemm. Economy of description for single-valued transducers. *Information and Computation*, 118(2):327–340, 1995.