PROGRAMMING USING AUTOMATA AND TRANSDUCERS

Loris D'Antoni

A DISSERTATION

in

Computer Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfilment of the Requirements for the

Degree of Doctor of Philosophy

2015

To Christian, Marinella, Monica, and Stefano

Acknowledgements

The material described in this dissertation is the culmination of an extended collaboration with Rajeev Alur and Margus Veanes, with additional contributions from Benjamin Livshits and David Molnar. I am lucky to count these talented individuals as colleagues and friends, and I want to acknowledge their contributions to this dissertation.

Throughout my time at Penn, my adviser Rajeev Alur has granted me freedom to pursue the projects I believed in and great support when I needed to brainstorm, work on a proof, or simply decide on what to do next. Thank you for encouraging, nurturing, and providing help along this academic journey. To NSF, thank you for the Expeditions in Computing grant CCF 1138996, which funded most of my research.

Sampath Kannan, Benjamin Pierce, Val Tannen, and Margus Veanes have all graciously agreed to serve on my dissertation committee. I am grateful for the time they devoted to reading both my proposal and this dissertation. Their many insightful comments and suggestions greatly improved it.

Mariangiola Dezani-Ciancaglini introduced me to research when I was an undergraduate nearly 8 years ago and was the first person to truly believe in my capabilities, for which I am eternally grateful. Margus Veanes mentored me for two productive internships at Microsoft Research Redmond during the summers of 2012 and 2013. Since then, Margus has been a great colleague, friend, and research collaborator.

To the formal methods, programming languages, and verification students at Penn, thank you for creating a friendly and stimulating research environment. To the Graduate Student in Engineering Group, thank you for all the social events and pleasant distractions from work.

To Adam, thank you for being an honest and supportive friend.

To my family, especially my parents Marinella and Stefano, thank you for supporting me and letting me pursue what I believed in. To my brother Christian, thank you for helping me push my potential to the limit.

To Monica, thank you for being with me and making my life complete.

ABSTRACT

PROGRAMMING USING AUTOMATA AND TRANSDUCERS

Loris D'Antoni

Dr. Rajeev Alur

Automata, the simplest model of computation, have proven to be an effective tool in reasoning about programs that operate over strings. Transducers augment automata to produce outputs and have been used to model string and tree transformations such as natural language translations. The success of these models is primarily due to their closure properties and decidable procedures, but good properties come at the price of limited expressiveness. Concretely, most models only support finite alphabets and can only represent small classes of languages and transformations.

We focus on addressing these limitations and bridge the gap between the theory of automata and transducers and complex real-world applications: *Can we extend automata and transducer models to operate over structured and infinite alphabets? Can we design languages that hide the complexity of these formalisms? Can we define executable models that can process the input efficiently?*

First, we introduce succinct models of transducers that can operate over large alphabets and design BEX, a language for analysing string coders. We use BEX to prove the correctness of UTF and BASE64 encoders and decoders. Next, we develop a theory of tree transducers over infinite alphabets and design FAST, a language for analysing tree-manipulating programs. We use FAST to detect vulnerabilities in HTML sanitizers, check whether augmented reality taggers conflict, and optimize and analyze functional programs that operate over lists and trees. Finally, we focus on laying the foundations of stream processing of hierarchical data such as XML files and program traces. We introduce two new efficient and executable models that can process the input in a left-to-right linear pass: symbolic visibly pushdown automata and streaming tree transducers. Symbolic visibly pushdown automata are closed under Boolean operations and can specify and efficiently monitor complex properties for hierarchical structures over infinite alphabets. Streaming tree transducers can express and efficiently process complex XML transformations while enjoying decidable procedures.

Contents

List of Tables

List of Figures

"When you do things right, people won't be sure you've done anything at all."

— The cosmic entity, *Futurama*

Chapter 1

Introduction

"We forget old stories, but those stories remain the same."
— Dejan Stojanovic, *The Sun Watches the Sun*

1.1 Preamble

How do we build reliable software? How do we increase programmers' productivity? With more than 18 million software developers worldwide, answering these questions is becoming increasingly important.[1] Although there has been huge progress in this area, reasoning about general purpose programs remains a hard task.

Programmers and end-users often need to solve tricky domain-specific tasks, such as writing text and XML transformations, that do not require the full power of general purpose programming languages. In fact, despite their complexity, these tasks often only require a few specialized programming constructs. This aspect opens the opportunity for creating simpler domain-specific tools that do not require the power of general-purpose programming. The ultimate goal of designing programming languages and techniques that make the programming of domain-specific tasks simpler, less error-prone, and more efficient is the main inspiration behind this dissertation.

In this dissertation we focus on programs that operate over strings, lists, and trees, and leverage the fact that finite state machines offer a suitable theoretical foundation for reasoning about these programs. Many variants of these finite state machines have been proposed and applied to different domains, such as program monitoring [LG07], natural language processing [MGHK09, Moh97, MK08], and XML processing [HP03,

[1]`http://www.infoq.com/news/2014/01/IDC-software-developers`

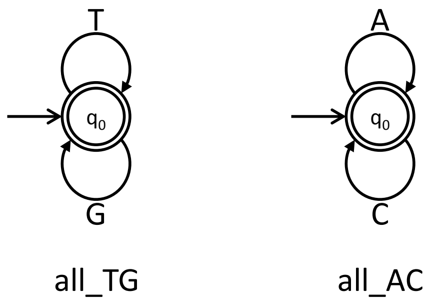


FIGURE 1.1: Examples of finite automata.

IH08, MSV00, BDM$^+$06, Bec03, MBPS05]. We build on these models and identify novel ones that can be used to analyze many real-world programs.

1.2 Automata, languages, and program properties

Finite automata define sets of strings, also called languages. They are used in many applications, most notably in analysis of regular expressions and program monitoring [LG07]. Intuitively a finite automaton is a labeled graph with initial and final states (nodes). The labels of the edges are symbols over an input alphabet and each path through the graph corresponds to a string. The language defined by a particular automaton is the set of all strings for which there is a path from an initial to a final state. For example, the automaton all_TG in Figure 1.1 accepts the language of all strings containing only the symbols T and G.

Automata can capture computations. In particular, they can be seen as programs that map lists to Boolean values. Examples of such programs can be seen in Figure 1.2. In particular, the programs all_TG and all_AC (lines 11–20) can be modeled using the automata depicted in Fig. 1.1 that accept strings over the alphabet base = {A, T, G, C}. Similarly, the programs is_dna and is_empty (lines 4–10) can be captured by two automata accepting all strings and only the empty string respectively.

What is the advantage of representing programs as automata? Unlike general purpose programs, finite automata enjoy many properties that can enable forms of program analysis. Automata can be complemented, intersected, determinized, and minimized, and it is decidable whether two automata accept the same languages. This opens a door of opportunities on what type of properties we can check for our programs in

Figure 1.2. For example, we can prove that the programs all_TG and all_AC accept disjoint languages by checking that the intersection of their corresponding automata is equivalent to the empty language.

1.3 Transducers, transformations, and programs

Although automata enjoy many useful properties, they can only model very limited classes of programs. Finite transducers extend finite automata to model partial functions from strings to strings, also called string transformations. This is done by adding an output component to transitions. A finite state transducer is a labeled graph with initial and final states (nodes), where each edge carries an input symbol and a sequence of output symbols. Each path corresponding to an input string also produces an output string by outputting sequences of characters at each transition. The concatenation of such sequences is the output corresponding to the traversed path. The transformation defined by a transducer is the set of all input/output string pairs for which there is a path from an initial to a final state. For example, the transducer map_base in Figure 1.3 defines the transformation that outputs the DNA sequence matching the input one (maps each base to the matching one, e.g. A to T).

Transducers can be seen as programs that map lists into lists. Examples of such programs can be seen in Figure 1.2. The programs map_base and filter_AC (lines 21–35) can be modeled using the transducers depicted in Figure 1.3.

Again, what is the advantage of modeling programs using transducers? Finite transducers also enjoy a variety of properties that enable powerful analysis. In particular, transducers are closed under sequential composition, and it is decidable whether two transducers define the same transformations. Also, given a transducer $\mathcal{T}$, and two automata I and O, it is decidable whether for every string accepted by the automaton I, the transducer $\mathcal{T}$ produces an output accepted by O. This problem is called regular type-checking. Consider the programs in Fig. 1.2. We can use the transducers' closure under sequential composition to automatically generate a transducer m_f_DNA that processes the input in a single pass without producing intermediate results (on the right Fig. 1.3). Similarly, we can build a transducer for the transformation f_m_f_DNA. We can then use type-checking to statically prove that for every list for which dnaseq returns true, the function f_m_f_DNA outputs a list in the language is_empty (i.e. the empty list).

```
(* Types for DNA bases and sequences *)
type base = A | T | G | C
type dnaseq = list base
(* True for any DNA sequence l *)
let is_dna (l: dnaseq) : bool = true
(* True for empty sequence *)
let is_empty (l: dnaseq) : bool =
    match l with
        [] -> true
      | _  -> false
(* True if all the elements of the list l are T or A *)
let rec all_TG (l: dnaseq) : bool  =
    match l with
        []   -> true
      | h::t -> (h = T || h = G) && (all_TG t)
(* True if all the elements of the list l are even *)
let rec all_AC (l: dnaseq) : bool =
    match l with
        []   -> true
      | h::t -> (h = A || h = C) && (all_AC t)
(* Replaces each base with the matching one*)
let rec map_base (l: dnaseq) : dnaseq =
    match l with
        []   -> []
      | A::t -> T::(map_base t)
      | T::t -> A::(map_base t)
      | G::t -> C::(map_base t)
      | C::t -> G::(map_base t)
(* Removes all the T and G elements from a list l *)
let rec filter_AC (l: dnaseq) : dnaseq = match l with
     []   -> []
   | A::t -> A::(filter_AC t)
   | C::t -> C::(filter_AC t)
   | T::t -> filter_AC t
   | G::t -> filter_AC t
(* Applies map_base then filter_AC to l *)
let m_f_DNA (l: dnaseq) : dnaseq = filter_AC (map_base l)
(* Applies filter_AC then map_base then filter_AC to l *)
let f_m_f_DNA (l: dnaseq) : dnaseq = m_f_DNA (filter_AC l)
(* Reverses a list*)
let rec reverse (l: dnaseq): dnaseq =
    match l with
        []   -> []
      | h::t -> (reverse t) @ [h]
```

FIGURE 1.2: Functional programs that can be modeled using transducers.

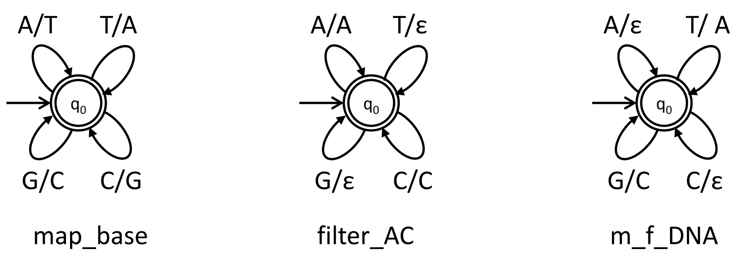


FIGURE 1.3: Examples of finite state transducers.

1.4 Limitations of existing models

Interesting programs can be modeled using finite automata and transducers. However, the capabilities of these formalisms are limited: they only operate over strings, they can only capture limited classes of languages and transformations (e.g. reverse in Fig. 1.2 cannot be modeled as a finite state transducer), and they are limited to finite alphabets.

Many models and techniques have been proposed to attenuate these limitations. Automata and transducers have been extended to trees [Eng75, Eng77, CDG$^+$07], richer transducer models have been presented to handle more complex functions [AC10, AC11, EH01], there have been extensions to support infinite and structured alphabets [VHL$^+$12, KF94], and a few programming languages were implemented to simplify programming in these models [PS12, MK08]. Although all these extensions have progressed towards addressing many of the limitations, there are simple domains that could potentially benefit from transducer-based analysis that cannot be modeled using existing models.

The following limitations still exist:

Alphabet expressiveness: Although several extensions have been proposed, many tree and string transformations and XML languages over infinite domains cannot be expressed using existing models;

Executable models: Most models proposed to capture larger classes of functions are not efficiently executable. In particular, they typically require multiple passes over the input.

Usability: Automata and transducers are complex to use for an average user and the integration between programming languages and these models is still limited.

1.5 Contributions

The goal of this dissertation is to bridge part of the gap between practical applications and the theory of automata and transducers. Its contributions are divided between the following areas:

Foundational results: We define novel automata and transducer models that can express large classes of languages and transformations. For many of these models we provide strong theoretical guarantees and exactly characterize their expressiveness.

Implementation: We implement two programming languages, FAST and BEX, that have their semantics given in terms of automata and transducers. Using these semantics, programs written in FAST and BEX can be statically analyzed. We also develop the open-source automata library SVPAlib, for coding automata over infinite alphabets.

Applications: We show how transducer-based languages are beneficial in proving properties of real-world programs. We use BEX to prove the correctness of BASE64 and UTF8 coders, and FAST to prove the absence of malicious outputs in HTML sanitizers, to check interference of augmented reality applications, to optimize functional language compilation, and to analyze functional programs over trees.

The rest of this chapter expands on each of these contributions in detail.

1.5.1 Foundational results

Automata are the pillars of computability and they are used in many domains thanks to their decidable procedures and closure properties. Our first contribution is to extend existing automata and transducer models to capture richer classes of languages and transformations. For these extended versions we prove closure properties and provide upper bounds for several decision procedures. Chapter 2 introduces symbolic extended finite automata and transducers, which extend finite automata and transducer with predicates to support infinite alphabets and the ability to process multiple adjacent symbols in a single transition. We prove negative properties for these models and show that for a restricted version, called Cartesian, checking equivalence is decidable. Chapter 3 defines symbolic tree transducers with regular look-ahead, which extend their classic counterpart with predicates to support infinite alphabets. We show that this model is closed under composition under the same assumptions that are necessary in the non-symbolic setting. Chapter 4 presents symbolic visibly pushdown automata, which extend visibly pushdown automata predicates to support infinite alphabets. We show that this expressive model is closed under Boolean operations, can be determinized, and enjoys decidable emptiness. Chapter 5 defines streaming tree transducers together with many equivalent variants. We show how streaming tree transducers capture the class of monadic-second-order-definable tree transformations, are closed under compositions and regular look-ahead, and enjoy decidable equivalence. For this last problem we provide a NExpTime upper bound, the first elementary upper bound for this class of transductions.

1.5.2 Language design and implementation

Automata and transducers are elegant models but are hard to program with. Our second contribution is the design and implementation of two programming languages, Bex and Fast, that act as frontend for automata and transducer models. Bex is a rule-based language described in Chapter 2, and it is a frontend for extended symbolic finite transducers over the theory of bit-vectors. This language allows the user to naturally program complex string encoders such as Utf8 and Base64 encoders and decoders using less than 50 lines of code. Bex supports regular-expression-based pattern matching, bit-vector operations such as bit-shift, bit-or, and bit-extract, and transducer operations such as equivalence and composition. Bex is also available as a runnable web interface at `http://rise4fun.com/Bex/`. Fast is a functional language described in Chapter 3, and it is a frontend for symbolic tree transducers with regular look-ahead over the theories supported by the satisfiability module theory (SMT) solver Z3 [DMB08]. These include theories over bit-vectors, integers, reals, and data-types. Fast also supports Boolean operations over languages, language equivalence, transducer composition, and regular type-checking. Fast is also available as a runnable interface at `http://rise4fun.com/fast/`. Finally, we implemented SVPAlib, an open-source symbolic automata library that supports typical automata operations for automata over infinite alphabets (Chapter 4). The SVPAlib library is available at `https://github.com/lorisdanto/symbolicautomata`.

1.5.3 Applications

Although elegant automata and transducer models are already rewarding by themselves, applying them to solve problems gives them new life. My third contribution is the use of automata and transducer models to solve real-world problems. In Chapter 2 we use Bex to program efficient versions of Base64 and Utf8 encoders and decoders that make heavy use of bit-level manipulations. Using Bex we prove the correctness of these encoders. These efficient implementations, thanks to their transducer-based semantics, have been used to generate efficient data-parallel code [VMML15]. In Chapter 3 we use FAST to 1) prove safety properties for HTML sanitizers in the context of web security; 2) analyze whether two augmented reality taggers ever try to annotate the same node of an input tree; 3) efficiently compose functional programs using transducer composition as a deforestation technique; and 4) prove pre-post condition properties of functional programs that transform lists and trees over complex domains. Finally, in Chapter 4 we show how to efficiently monitor complex properties on XML documents and structured program traces using the library SVPAlib.

1.6 Acknowledgements

This dissertation describes work performed in cooperation with many different colleagues and portions of it are based on papers written in collaboration with them. In particular, the presentation of the language BEX described in Chapter 2 is a revised version of two papers by D'Antoni and Veanes [DV13b, DV13a, DV15]. The language FAST was designed in collaboration with Margus Veanes and the description presented in Chapter 3 is an extended version of a paper by D'Antoni, Veanes, Livshits, and Molnar [DVLM14]. Chapter 4 is a revised version of a paper by D'Antoni and Alur [DA14]. Chapter 5 is an extended version of a paper by Alur and D'Antoni [AD12].

Part I

Transducer-based programming languages

Chapter 2

BEX: a language for verifying string coders

"Words are the source of misunderstandings."
— Antoine de Saint-Exupéry, *The Little Prince*

2.1 Introduction

String coders enable communication across different applications by translating between different string formats. String coders are ubiquitous: emails sent through SMTP are encoded using BASE64, text entered inside HTML forms is encoded using URL encode, and Unicode text files are converted from UTF8 to UTF16 when stored in memory.

Despite the wide adoption of these encodings, their analysis is difficult, and carefully crafted *invalid* UTF8 sequences have been used to bypass security validations. Several attacks have been demonstrated based on over-encoding the characters '`.`' and '`/`' in malformed URLs.[1] For example, the invalid sequence $[\texttt{C0}_{16}, \texttt{AF}_{16}]$ (that decodes to '`/`') has been used to bypass a literal check in the Microsoft IIS server (in unpatched Windows 2000 SP1) to determine if a URL contains "`../../`" by encoding it as "`..%C0%AF../`". Similar vulnerabilities exists in Apache Tomcat ($\leq$ 6.0.18), where "`%C0%AE`" has been used for encoding '`.`'.[2] Further attacks have used double-encoding, where the input string is encoded twice to violate some of the decoder assumptions.[3]

[1] `http://www.sans.org/security-resources/malwarefaq/wnt-unicode.php`
[2] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-2938`
[3] `https://www.owasp.org/index.php/Double_Encoding`

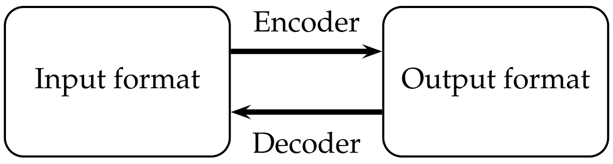


FIGURE 2.1: Role of string encoders and decoders.

2.1.1 Challenges in verifying real-world coders

To verify string coders we need to check whether the encoder (E) and decoder (D) invert each other:

$$E \circ D \equiv I \wedge D \circ E \equiv I$$

Here I denotes the identity function. This property guarantees that there exists a bijection between the input and the output format. This implies that there is only one way to encode each string, preventing attacks such as the one used in Apache Tomcat, in which the same symbol can be encoded in two different ways. Checking this property requires the use of sequential composition and functional equivalence. Luckily, finite state transducers are closed under composition and enjoy decidable equivalence.

Unfortunately the problem is not so simple. Despite their small sizes, string coders are hard to verify using finite state transducers due to the following reasons:

Large alphabets: coders operate over large alphabets containing up to 2^{32} characters;

Bit-vectors: coder implementations perform complex bit-vector operations to achieve efficiency;

Look-ahead: each output symbol can depend on multiple adjacent input symbols.

These challenges can already be observed in the simple case of BASE64 encoding (Figure 2.2). A BASE64 encoder transforms sequences of bytes into sequences of 6-bit characters. This is done by reading 3 characters at a time, splitting the resulting sequence of 24 bits into groups of 6 bits, and applying a character mapping to the each 6-bit number. Even in this simple setting the input alphabet contains 256 characters, the output is computed using bit-vector operation to isolate particular bits, and each output symbol may depend on different input symbols. For example in Figure 2.2 the output letter 'W' depends on the last two bits of the letter 'M' and the first four bits of the letter 'a'.

Modeling this encoder with a finite state transducer requires storing part of the input symbols in the state component. For example, after reading the letter 'a' in the example of Figure 2.2, the transducer will have to remember in its state the last four bits of 'a' and use them in the next step to build the corresponding output symbol. For large alphabets, this would cause an explosion in the number of states and transitions. Thus, finite state transducers are not a good candidate for verifying string coders as they cannot naturally represent bit-vector semantics, and they would face a state-space explosion in the case of large alphabets.

Text content	M								a								n							
ASCII	77 ($4D_{16}$)								97 (61_{16})								110 ($6E_{16}$)							
Bit pattern	0	1	0	0	1	1	0	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0
Index	19						22						5						46					
BASE64	T						W						F						u					

FIGURE 2.2: BASE64 encoding

2.1.2 Contributions

We introduce Symbolic Extended Finite Transducers (S-EFT) as a model for verifying string coders. S-EFTs differ from finite state transducers in the following aspects:

Symbolic alphabets: transitions operate over the alphabet symbolically via predicates and functions;

Finite look-ahead: transitions can read more than one input symbol at a time.

These aspects address the limitations of finite state transducers and allow us to naturally model coders like the one shown in Figure 2.2.

Unfortunately, we show that S-EFTs are not closed under composition and the equivalence problem is undecidable. We therefore introduce a subclass of S-EFTs called Cartesian for which we provide a practical composition algorithm and show equivalence to be decidable. In a Cartesian S-EFT, the predicates appearing in each transition are only allowed to contain conjunctions of unary predicates. We demonstrate how this restriction is enough to model string coders.

Finally, we design BEX, an S-EFT-based language for verifying string coders, and we use it to prove verify efficient implementations of BASE16, BASE32, BASE64, and UTF8 encoders and decoders.

This chapter is structured as follows:

- in Section 2.2 we give an overview of BEX and present a small case-study;
- in Section 2.3 we define Symbolic Extended Finite Automata (S-EFAs), Symbolic Extended Finite Transducers (S-EFTs), and their subclasses;

- in Section 2.4 we study of the closure and decidability properties of S-EFAs;
- in Section 2.5 we study the equivalence problem for S-EFTs and we show that:
  - the equivalence of single-valued S-EFTs is undecidable;
  - the equivalence of single-valued Cartesian S-EFTs is decidable;
- in Section 2.6 we show that S-EFTs are not closed under composition and provide a sound but incomplete composition algorithm;
- in Section 2.7 we show how BEX can be used to program real-world coders;
- in Section 2.8 we discuss related work.

2.2 Verifying BASE64 in BEX

We give an overview of the language BEX and illustrate how we can use it to verify an implementation of a BASE64 encoder and a decoder (Figure 2.3). BASE64 is used to transfer binary data in textual format, e.g., in emails via the protocol MIME. The digits of the encoding are chosen in the safe ASCII range of characters that remain unmodified during transport over textual media.

Figure 2.3 contains BEX implementations of base64encode and base64decode. The encoder reads characters (bytes) in the range 0 and 255 and produces characters in the set $[a\text{–}zA\text{–}Z0\text{–}9+/]$. The decoder does the opposite.

The program 'base64encode' reads 3 characters at a time and applies the operation we illustrated in Figure 2.2. Each 6 bit character obtained after the split is transformed using the function 'em'. If the string has a length that is not divisible by 3, one or two '=' symbols are added at the end of the output (in the figure the character '\$' stands for end of string). We explain the syntax of BEX using the rule in lines 12–14. The rule can be read as: if the first two symbols of the input are two bytes (characters between 0 and FF), then output the four comma-separated characters appearing on the right-hand side of the rule. The symbols #0 and #1 respectively represent the first and second character being read. In the figure 'bits(i, j, s)' is the function that, given a bit-vector s, extracts the bit-vector starting at index i (from right to left) and ending at index j. For example 'bits$(2, 1, 0110)$=11'. The function 'ite(b, s_1, s_2)' stands for 'if b then s_1 else s_2'. After applying a rule that reads n symbols of a string $c_1 \ldots c_n c_{n+1} \ldots c_k$, BEX produces the rule's output and then calls the same function again on the suffix $c_{n+1} \ldots c_k$.

The program 'base64decode' inverts the transformation by reading 4 characters at a time. Notice how the rules in lines 23–25 take care of the case in which the encoder

```
1   // Character maps
2   function em(x) := ite(x <= 25, x + 65, ite(x <= 51, x + 71,
3                          ite(x <= 61, x − 4, ite(x == 62, '+', '/'))))
4   function dm(x) := ite(x == '/', 63, ite(x == '+', 62,
5                          ite(x <= '9', x + 4, ite(x <= 'Z', x − 65, x − 71))))
6   // Base64 encoder
7   program base64encode := repeat {
8       [\0–\xFF]{3}   ⇒  [em(bits(7,2,#0)),
9                           em((bits(1,0,#0) << 4)|bits(7,4,#1)),
10                            em((bits(3,0,#1) << 2)|bits(7,6,#2)),
11                              em(bits(5,0,#2))];
12      [\0–\xFF]{2}$  ⇒  [em(bits(7,2,#0)),
13                          em((bits(1,0,#0) << 4)|bits(7,4,#1)),
14                            em(bits(3,0,#1) << 2),'='];
15      [\0–\xFF]{1}$  ⇒  [em(bits(7,2,#0)),
16                          em((bits(1,0,#0) << 4)|bits(7,4,#1)),'=','='];
17  }
18  // Base64 decoder
19  program base64decode := repeat {
20      [a–zA–Z0–9+/]{4}     ⇒  [(dm(#0) << 2)|bits(5,4,dm(#1)),
21                                (bits(3,0,dm(#1)) << 4)|bits(5,2,dm(#2)),
22                                  (bits(1,0,dm(#2)) << 6)|dm(#3)];
23      [a–zA–Z0–9+/]{3}=$   ⇒  [(dm(#0) << 2)|bits(5,4,dm(#1)),
24                                (bits(3,0,dm(#1)) << 4)|bits(5,2,dm(#2))];
25      [a–zA–Z0–9+/]{2}==$  ⇒  [(dm(#0) << 2)|bits(5,4,dm(#1))];
26  }
27  // Identity
28  program identity { [\0–\xFF] ⇒ [#0];}
29  // Check correctness
30  assert-true (comp(base64encode, base64decode), eq(identity));
```

FIGURE 2.3: BASE64 analysis in BEX.

added equal ('=') symbols at the end of the string. The program 'identity' applies the identity function to strings over characters in the range 0 and 255.

Using BEX we can check whether our implementation of the BASE64 decoder correctly inverts the encoder. This is done by writing the assertion on line 30 which BEX can automatically verify. The assertion says that the composition of the encoder with the decoder is functionally equivalent to the identity. Verifying this property requires the ability to compose programs and check their equivalence.

2.3 Symbolic extended finite automata and transducers

In this section we formally define the automata and transducer models that provide the semantics of BEX.

2.3.1 Preliminaries

We assume a recursively enumerable (r.e.) *background universe* $\mathcal{U}$ with function and relation symbols. Definitions below are given with $\mathcal{U}$ as an implicit parameter. We use λ-terms to represent anonymous functions. A Boolean λ-term $\lambda x.\varphi(x)$, where x is a variable of type σ is called a σ-predicate. Our notational conventions are consistent with the definition of symbolic transducers [VHL+12]. The universe is multi-typed with $\mathcal{U}^\tau$ denoting the sub-universe of elements of type τ. We write Σ for $\mathcal{U}^\sigma$ and Γ for $\mathcal{U}^\gamma$.

A *label theory* is given by a recursively enumerable set Ψ of formulas that is closed under Boolean operations, substitution, equality and if-then-else terms. A label theory Ψ is *decidable* when satisfiability for $\varphi \in \Psi$, $\mathit{IsSat}(\varphi)$, is decidable.

For σ-predicates φ, we assume an effective *witness* function $\mathscr{W}$ such that, if $\mathit{IsSat}(\varphi)$ then $\mathscr{W}(\varphi) \in [\![\varphi]\!]$, where $[\![\varphi]\!] \subseteq \mathcal{U}^\sigma$ is the set of all values that satisfy φ; φ is *valid*, $\mathit{IsValid}(\varphi)$, when $[\![\varphi]\!] = \mathcal{U}^\sigma$.

2.3.2 Model definitions

In this chapter we present an extension of S-FTs where transitions are allowed to consume more than one symbol, called *extended* S-FTs or *S-EFTs*.

Definition 2.1. A *Symbolic Extended Finite Transducer (S-EFT)* with *input type* σ and *output type* γ is a tuple $A = (Q, q^0, R)$,

- Q is a finite set of *states;*
- $q^0 \in Q$ is the *initial state*;
- R is a finite set of *rules*, $R = \Delta \cup F$, where
  - Δ is a set of *transitions* $r = (p, \ell, \varphi, f, q)$, denoted $p \xrightarrow[\ell]{\varphi/f} q$, where

    $p \in Q$ is the *start* state of r;

    $\ell \geq 1$ is the *look-ahead* of r;

    φ, the *guard* of r, is a σ^ℓ-predicate;

    f, the *output* of r, is a λ-terms from σ^ℓ to sequences of γ;

    $q \in Q$ is the *continuation* state of r;
  - F is a set of *finalizers* $r = (p, \ell, \varphi, f)$, denoted $p \xrightarrow[\ell]{\varphi/f} \bullet$, with components as above and where ℓ may be 0.

The *look-ahead* of A is the maximum of all look-aheads of rules in R. An S-EFT where all the rules have output ε is a *Symbolic Extended Finite Automaton* (*S-EFA*).

A finalizer is a rule without a continuation state. A finalizer with look-ahead ℓ is used when the end of the input sequence has been reached with *exactly* ℓ input elements remaining. A finalizer is a generalization of a final state. In the non-symbolic setting, finalizers can be avoided by adding a new symbol to the alphabet that is only used to mark the end of the input. In the presence of arbitrary input types, this is not always possible without affecting the theory, e.g., when the input type is *Int* then that symbol would have to be outside *Int*.

In the remainder of the section let $A = (Q, q^0, R)$, $R = \Delta \cup F$, be a fixed S-EFT with input type σ and output type γ. The semantics of rules in R is as follows:

$$[\![p \xrightarrow[\ell]{\varphi/f} q]\!] \stackrel{\text{def}}{=} \{ p \xrightarrow[\ell]{[a_0,\ldots,a_{\ell-1}]/[\![f]\!](a_0,\ldots,a_{\ell-1})} q \mid (a_0,\ldots,a_{\ell-1}) \in [\![\varphi]\!] \}.$$

The notation $p \xrightarrow[\ell]{[a_0,\ldots,a_{\ell-1}]/[\![f]\!](a_0,\ldots,a_{\ell-1})} q$ is a shorthand for a tuple $(p, \ell, (a_0,\ldots,a_{\ell-1}), [\![f]\!](a_0,\ldots,a_{\ell-1}), q)$. Intuitively, a rule with look-ahead ℓ reads ℓ adjacent input symbols $s = [a_0,\ldots,a_{\ell-1}]$ and produces a sequence of output symbols that is a function of the consumed input $f(s)$.

Let $[\![R]\!] \stackrel{\text{def}}{=} \bigcup_{r\in R} [\![r]\!]$. We write $s_1 \cdot s_2$ for the concatenation of sequences s_1 and s_2.

Definition 2.2. For $u \in \Sigma^*, v \in \Gamma^*, q \in Q, q' \in Q \cup \{\bullet\}$, define $q \overset{u/v}{\twoheadrightarrow}_A q'$ as follows: there exists $n \geq 0$ and $\{p_i \xrightarrow{u_i/v_i} p_{i+1} \mid i \leq n\} \subseteq [\![R]\!]$ such that

$$u = u_0 \cdot u_1 \cdots u_n, \quad v = v_0 \cdot v_1 \cdots v_n, \quad q = p_0, \quad q' = p_{n+1}.$$

Let also $q \overset{\varepsilon/\varepsilon}{\twoheadrightarrow}_A q$ for all $q \in Q$.

Definition 2.3. The *transduction relation* of A is defined as $\mathscr{T}_A(u) \stackrel{\text{def}}{=} \{v \mid q^0 \overset{u/v}{\twoheadrightarrow} \bullet\}$.

The following example illustrates typical (realistic) S-EFTs over a label theory of linear modular arithmetic. We use the following abbreviated notation for rules, by omitting explicit λ's. We write

$$p \xrightarrow[\ell]{\varphi(\bar{x})/[f_1(\bar{x}),\ldots,f_k(\bar{x})]} q \quad \text{for} \quad p \xrightarrow[\ell]{\lambda\bar{x}.\varphi(\bar{x})/\lambda\bar{x}.[f_1(\bar{x}),\ldots,f_k(\bar{x})]} q,$$

where φ and f_i are terms whose free variables are among $\bar{x} = (x_0,\ldots,x_{\ell-1})$.

Example 2.4. *This example illustrates the S-EFTs Base64encode and Base64decode corresponding to the* BEX *programs 'base64encode' and 'base64decode' presented in Figure 2.3.*

Base64encode is an S-EFT with one state and four rules:

$$p \xrightarrow[3]{\mathit{true}/[\ulcorner b_2^7(x_0)\urcorner,\, \ulcorner(b_0^1(x_0)\ll 4)|b_4^7(x_1)\urcorner,\, \ulcorner(b_0^3(x_1)\ll 2)|b_6^7(x_2)\urcorner,\, \ulcorner b_0^5(x_2)\urcorner]} p$$

$$p \xrightarrow[0]{\mathit{true}/[]} \bullet \qquad p \xrightarrow[1]{\mathit{true}/[\ulcorner b_2^7(x_0)\urcorner,\, \ulcorner b_0^1(x_0)\ll 4\urcorner,\ \text{'='},\ \text{'='}]} \bullet$$

$$p \xrightarrow[2]{\mathit{true}/[\ulcorner b_2^7(x_0)\urcorner,\, \ulcorner(b_0^1(x_0)\ll 4)|b_4^7(x_1)\urcorner,\, \ulcorner b_0^3(x_1)\ll 2\urcorner,\ \text{'='}]} \bullet$$

where $b_n^m(x)$ extracts bits m through n from x, e.g., $b_2^3(13) = 3$, $x|y$ is bitwise OR of x and y, $x\ll k$ is x shifted left by k bits, and $\ulcorner x\urcorner$ is the mapping

$$\ulcorner x\urcorner \overset{\mathit{def}}{=} (x\le 25\,?\,x{+}65 : (x\le 51\,?\,x{+}71 : (x\le 61\,?\,x{-}4 : (x{=}62\,?\ \text{'+'} : \text{'/'}))))$$

of values between 0 *and* 63 *into a standardized sequence of safe ASCII character codes ('en' in Figure 2.3). The last two finalizers correspond to the cases when the length of the input sequence is not a multiple of three. Observe that the length of the output sequence is always a multiple of four. The character* '=' *(61 in ASCII) is used as a padding character and it is not a* Base64 *digit. i.e.,* '=' *is not in the range of $\ulcorner x\urcorner$.*

Base64decode is an S-EFT that decodes a Base64 *encoded sequence back into the original byte sequence. Base64decode has also one state and four rules:*

$$q \xrightarrow[4]{\bigwedge_{i=0}^{3}\beta_{64}(x_i)/[(\llcorner x_0\lrcorner\ll 2)|b_4^5(\llcorner x_1\lrcorner),\ (b_0^3(\llcorner x_1\lrcorner)\ll 4)|b_2^5(\llcorner x_2\lrcorner),\ (b_0^1(\llcorner x_2\lrcorner)\ll 6)|\llcorner x_3\lrcorner]} q$$

$$q \xrightarrow[0]{\mathit{true}/[]} \bullet \qquad q \xrightarrow[4]{\beta_{64}(x_0)\wedge\beta'_{64}(x_1)\wedge x_2{=}\ \text{'='}\wedge x_3{=}\ \text{'='}/[(\llcorner x_0\lrcorner\ll 2)|b_4^5(\llcorner x_1\lrcorner)]} \bullet$$

$$q \xrightarrow[4]{\beta_{64}(x_0)\wedge\beta_{64}(x_1)\wedge\beta''_{64}(x_2)\wedge x_3{=}\ \text{'='}/[(\llcorner x_0\lrcorner\ll 2)|b_4^5(\llcorner x_1\lrcorner),\ (b_0^3(\llcorner x_1\lrcorner)\ll 4)|b_2^5(\llcorner x_2\lrcorner)]} \bullet$$

The function $\llcorner y\lrcorner$ is the inverse of $\ulcorner x\urcorner$, i.e., $\llcorner\ulcorner x\urcorner\lrcorner = x$, for $0 \le x \le 63$. The predicate $\beta_{64}(y)$ is true iff y is a valid Base64 *digit, i.e., $y = \ulcorner x\urcorner$ for some x, $0 \le x \le 63$. The predicates $\beta'_{64}(y)$ and $\beta''_{64}(y)$ are restricted versions of $\beta_{64}(y)$. Unlike Base64encode, Base64decode does not accept all input sequences of bytes, and sequences that do not correspond to any encoding are rejected.*[4]

The following subclass of S-EFTs captures transductions that behave as partial functions from Σ^* to Γ^*.

Definition 2.5. A function $\mathbf{f} : X \to 2^Y$ is *single-valued* if $|\mathbf{f}(x)| \le 1$ for all $x \in X$. An S-EFT A is single-valued if $\mathscr{T}_A$ is single-valued.

A sufficient condition for single-valuedness is determinism. We define $\varphi \curlywedge \psi$, where φ is a σ^m-predicate and ψ a σ^n-predicate, as the $\sigma^{\max(m,n)}$-predicate

[4]For more information see `http://www.rise4fun.com/Bek/tutorial/base64`.

$\lambda(x_1,\ldots,x_{\max(m,n)}).\varphi(x_1,\ldots,x_m) \wedge \psi(x_1,\ldots,x_n)$. We define *equivalence of f and g with respect to φ, $f \equiv_\varphi g$*, as: $\mathit{IsValid}(\lambda\bar{x}.(\varphi(\bar{x}) \Rightarrow f(\bar{x}) = g(\bar{x})))$.

Definition 2.6. A is *deterministic* if for all $p \xrightarrow[\ell]{\varphi/f} q, p \xrightarrow[\ell']{\varphi'/f'} q' \in R$:

(a) Assume $q, q' \in Q$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $q = q'$, $\ell = \ell'$ and $f \equiv_{\varphi \curlywedge \varphi'} f'$.

(b) Assume $q = q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ and $\ell = \ell'$ then $f \equiv_{\varphi \curlywedge \varphi'} f'$.

(c) Assume $q \in Q$ and $q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $\ell > \ell'$.

Intuitively, determinism means that no two rules may overlap. It follows from the definitions that if A is deterministic then A is single-valued. Both S-EFTs in Example 2.4 are deterministic.

The *domain* of a function $\mathbf{f} : X \to 2^Y$ is $\mathscr{D}(\mathbf{f}) \stackrel{\text{def}}{=} \{x \in X \mid \mathbf{f}(x) \neq \emptyset\}$ and for an S-EFT A, $\mathscr{D}(A) \stackrel{\text{def}}{=} \mathscr{D}(\mathscr{T}_A)$. When A is single-valued and $u \in \mathscr{D}(A)$, we treat A as a partial function from Σ^* to Γ^* and write $A(u)$ for the value v such that $\mathscr{T}_A(u) = \{v\}$. For example, *Base64encode*(`"Foo"`) = `"Rm9v"` and *Base64decode*(`"QmFy"`) = `"Bar"`.

2.3.3 From BEX to S-EFTs

We describe how BEX programs are compiled into S-EFTs. The syntaxes of S-EFTs and BEX are very similar and BEX provides a simple way to represent the rules of an S-EFT.

In BEX the state component is maintained by an integer variable q, which is initialized to a value v_0.

$$\textbf{program}\ \text{name} := [q = v_0]\ \textbf{repeat}\ \{\mathit{body}\}$$

The program base64encode in Figure 2.2 has only one state and this component does not appear. Rules are of the two different forms. The first form is

$$\mathit{guard}, q = v_1 \Rightarrow \mathit{update}[q = v_2]$$

where *guard* is a regular expression that accepts only strings with some finite length l and does not contain the symbol \$ denoting the end of string, and *update* is a sequence of bit-vector-arithmetic expressions over the free variables $\{\#0, \ldots, \#\{l-1\}\}$. This rule is translated into an S-EFT rule

$$v_1 \xrightarrow[l]{\lambda x_1,\ldots,x_l.(x_1\cdots x_l)\in L(\mathit{guard})/\lambda\#0,\ldots,\#\{l-1\}.\mathit{update}} v_2$$

where $L(\mathit{guard})$ is the set of strings accepted by the regular expression *guard*. The programs in Figure 2.2 only have one state and the state component is therefore implicit.

The second form is

$$guard\$, q = v_1 \Rightarrow update$$

where *guard* is a regular expression that accepts only strings with some finite length l and does not contain the symbol $\$$, and *update* is a sequence of bit-vector-arithmetic expressions over the free variables $\{\#0, \ldots, \#\{l-1\}\}$. This rule is translated into an S-EFT rule

$$v_1 \xrightarrow[l]{\lambda x_1,\ldots,x_l.(x_1\cdots x_l)\in L(guard)/\lambda\#0,\ldots,\#\{l-1\}.update} \bullet$$

2.3.4 Cartesian S-EFAs and S-EFTs

We introduce a subclass of S-EFTs that plays an important role in this chapter. A binary relation R over X is *Cartesian over* X if R is the Cartesian product $R_1 \times R_2$ of some $R_1, R_2 \subseteq X$. The definition is lifted to n-ary relations and σ^n-predicates for $n \geq 2$ in the obvious way. In order to decide if a satisfiable σ^n-predicate φ is Cartesian over σ, let $(a_0, \ldots, a_{n-1}) = \mathscr{W}(\varphi)$ and perform the following validity check:

$$IsCartesian(\varphi) \overset{\text{def}}{=} \forall \bar{x}\, (\varphi(\bar{x}) \Leftrightarrow \bigwedge_{i<n} \varphi(a_0, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_{n-1})).$$

In other words, a σ^n-predicate φ is Cartesian over σ if φ can be rewritten equivalently as a conjunction of n independent σ-predicates.

Definition 2.7. An S-EFT (S-EFA) is *Cartesian* if all its guards are Cartesian.

Both S-EFTs in Example 2.4 are Cartesian. *Base64encode* is trivially so, while the guards of all rules of *Base64decode* are conjunctions of independent unary predicates. In contrast, a predicate such as $\lambda(x_0, x_1).x_0 = x_1$ is not Cartesian.

Note that $IsCartesian(\varphi)$ is decidable by using the decision procedure of the label theory. Namely, decide unsatisfiability of $\neg IsCartesian(\varphi)$.

Cartesian S-EFAs capture exactly the class of S-FA definable languages. An S-FA is an S-EFA with final states and in which all transitions have lookahead 1.

Theorem 2.8 (Cartesian S-EFA = S-FA). *Cartesian S-EFAs and S-FAs are equivalent in expressiveness.*

Proof. The $\Leftarrow$ direction is immediate. We prove the $\Rightarrow$ direction. Given a Cartesian S-EFA $A = (Q, q_0, (\Delta, F))$ we construct an equivalent S-FA A'. Without loss of generality we assume that every rule r in Δ has look-ahead 2 and every finalizer has look-ahead 0. For every rule $r = p \xrightarrow[2]{\varphi(x_1)\wedge\psi(x_2)} q$, the S-FA A' has a 3 states q, p, q_r and two rules

$p \xrightarrow{\varphi(x_1)} r$, $r \xrightarrow{\psi(x_2)} q$. Finally, if A has a finalizer $q \xrightarrow[0]{true} \bullet$, the state q will be final in A'. □

As a consequence Cartesian S-EFAs enjoy all the properties of S-FAs (regular languages) such as boolean closures and decidability of equivalence.

2.3.5 Monadic S-EFAs and S-EFTs

We say that a (quantifier free) formula is in *monadic normal form* or *MNF* if it is a Boolean combination of unary formulas, where a *unary formula* is a formula with at most one free variable. A formula is *monadic* if it has an equivalent MNF. A natural problem that arises is deciding whether a formula is monadic and, if so, constructing its MNF. For example, the formula $x < y$ over integers does not have an MNF while the formula $x < y \bmod 2$ has an MNF $(x < 0 \wedge y \bmod 2 = 0) \vee (x < 1 \wedge y \bmod 2 = 1)$ that is also a DNF. Another MNF of $x < y \bmod 2$ is $x < 0 \vee (x < 1 \wedge y \bmod 2 = 1)$, which is also a DNF but with semantically overlapping disjuncts.

Definition 2.9. An S-EFT (S-EFA) is *monadic* if all its guards are monadic.

Veanes et al. [VBNB14] showed that if the label theory is decidable and a formula is monadic then its MNF can be constructed effectively.

Theorem 2.10. *Monadic S-EFTs and Cartesian S-EFTs are effectively equivalent. Moreover, this holds also for the deterministic case.*

Proof. The $\Leftarrow$ direction is immediate because Cartesian S-EFTs are a special case of monadic S-EFTs. For the direction $\Rightarrow$, we first apply the procedure *mondec* from Veanes et al. [VBNB14] to each guard ϕ of the monadic S-EFT to obtain an equivalent MNF of the guard, which we then rewrite into an equivalent DNF $\bigvee_{i<n} \phi_i$. Finally, we can replace each rule $p \xrightarrow[\ell]{\phi/f} q$ by the rules $p \xrightarrow[\ell]{\phi_i/f} q$, for $i < n$, where all ϕ_i are Cartesian. Determinism is clearly preserved, because all the new rules have identical outputs so the conditions (a) and (b) of Definition 2.6 are trivially fulfilled. □

2.4 Properties of symbolic extended finite automata

In this section we prove some basic properties of Symbolic Extended Finite Automata and show how they drastically differ from S-FAs and have properties similar to those of context free grammars rather than regular languages.

First, we show how checking the emptiness of the intersection of two S-EFA definable languages is an undecidable problem.

Theorem 2.11 (Domain Intersection)**.** *Given two S-EFAs A and B with look-ahead 2 over quantifier-free successor arithmetic and tuples, checking whether there exists an input accepted by both A and B is undecidable.*

Proof. Recall that a Minsky machine has two registers r_1 and r_2 that can hold natural numbers and a program that is a finite sequence of instructions. Each instruction is one of the following: INC_i (increment r_i and continue with the next instruction); DEC_i (decrement r_i if $r_i > 0$ and continue with the next instruction); $JZ_i(j)$ (if $r_i = 0$ then jump to the j'th instruction else continue with the next instruction). The machine halts when the end of the program is reached. Let M be a Minsky machine with program P. Let $\sigma = \mathbb{N}^3$ represent the type of the snapshot or configuration $(\textit{program counter}, r_1, r_2)$ of M.

Suppose $\pi_j : \sigma \to \mathbb{N}$ projects the j'th element of a k-tuple where $0 \leq j < k$. Construct S-EFAs A and B over σ as follows. Let φ^{ini} be the σ-predicate $\lambda x.(x = (0,0,0))$ stating that the program counter and both registers are 0. Let φ^{fin} be the final σ-predicate $\lambda x.(\pi_0(x) = |P| \wedge \pi_1(x) \neq 0)$.

Let φ^{step} be the σ^2-predicate $\lambda(x,x').\bigvee_{i<|P|} \varphi_i^{\text{step}}$ where φ_i^{step} is the formula for the i'th instruction. If the i'th instruction is INC_1 then φ_i^{step} is

$$\pi_0(x) = i \wedge \pi_0(x') = i+1 \wedge \pi_1(x') = \pi_1(x) + 1 \wedge \pi_2(x') = \pi_2(x).$$

If the i'th instruction is $JZ_1(j)$ then φ_i^{step} is

$$\pi_0(x) = i \wedge \pi_0(x') = Ite(\pi_1(x) = 0, j, i+1) \wedge \pi_1(x') = \pi_1(x) \wedge \pi_2(x') = \pi_2(x).$$

Similarly for the other cases. Thus, φ^{step} encodes the valid step relation of M from current configuration x to the next configuration x'. Let

$$\begin{aligned} A &= (\{p_0\}, p_0, \{p_0 \xrightarrow[2]{\varphi^{\text{step}}} p_0, \quad p_0 \xrightarrow[0]{true} \bullet\}) \text{ and} \\ B &= (\{q_0, q_1\}, q_0, \{q_0 \xrightarrow[1]{\varphi^{\text{ini}}} q_1, \quad q_1 \xrightarrow[2]{\varphi^{\text{step}}} q_1, \quad q_1 \xrightarrow[1]{\varphi^{\text{fin}}} \bullet\}). \end{aligned}$$

So $\alpha \in \mathscr{D}(A) \cap \mathscr{D}(B)$ iff α is a valid computation of M, i.e., $\alpha[0]$ is the initial configuration, $\alpha[i+1]$ is a valid successor configuration of $\alpha[i]$ (this follows from A for all odd $i < |\alpha|$ and from B for all even $i < |\alpha|$), and $\alpha[|\alpha| - 1]$ is a halting configuration.

It follows that $\mathscr{D}(A) \cap \mathscr{D}(B) \neq \emptyset$ iff M halts on input $(0,0)$ with a non-zero output in r_1. The latter is an undecidable problem as an instance of Rice's theorem. □

Theorem 2.12 (Emptiness)**.** *Given an S-EFA A it is decidable to determine whether it accepts any input.*

Given A, we first remove all the transitions with unsatisfiable guards. Let's call the new S-EFA A'. If A' has a path from the initial state to $\bullet$, then A is not empty.

Theorem 2.13 (Boolean Properties)**.** *S-EFAs are closed under union but not closed under intersection and complement.*

Proof. Given two S-EFAs $A_1 = (Q_1, q_0^1, R_1)$ and $A_2 = (Q_2, q_0^2, R_2)$ over a sort σ we construct an S-EFA B over σ such that $\mathscr{D}(C) = \mathscr{D}(A) \cup \mathscr{D}(B)$. C will have states $Q = Q_1 \cup Q_2 \cup \{q_0\}$ and initial state q_0. The transition relation R of C is then defined as follows:

$$R = R1 \cup R2 \cup \{q_0 \xrightarrow[k]{\varphi} q \mid q_0^1 \xrightarrow[k]{\varphi} q \in R_1 \vee q_0^2 \xrightarrow[k]{\varphi} q \in R_2\}.$$

As a consequence of Theorems 2.11, and 2.12 we have that S-EFA are not closed under intersection; if they were using Theorem 2.12 one could decide intersection emptiness. Since S-EFAs are closed under union, S-EFAs cannot be closed under complement; if they were we could use Boolean operations to show closure under intersection. $\square$

While checking the emptiness of an S-EFA is a decidable problem, it is not possible to decide whether an S-EFA accepts every possible input. It follows that equivalence is also undecidable.

Theorem 2.14 (Universality and Equivalence)**.** *Given an S-EFA A over σ it is undecidable to check whether A accepts all the sequences in σ^*, and given two S-EFAs A and B it is undecidable to check whether A and B accept the same language.*

Proof. Let M be a Minsky machine with program P. Let $\sigma = \mathbb{N}^3$ represent the type of the snapshot or configuration $(\textit{program counter}, r_1, r_2)$ of M. Let φ^{ini}, φ^{fin} and φ^{step} be as in Theorem 2.11.

We construct an S-EFA A_M that does not accept all the strings in σ^* iff M halts on input $(0, 0)$ with a non-zero output in r_1. The latter is an undecidable problem as an instance of Rice's theorem.

Let

$$\begin{array}{rcl}
A & = & (\{p_0, p_1\}, p_0, \{p_0 \xrightarrow[1]{true} p_0, \quad p_0 \xrightarrow[2]{\neg\varphi^{\text{step}}} p_1, \quad p_1 \xrightarrow[1]{true} p_1, \quad p_1 \xrightarrow[0]{true} \bullet\}),\\
B & = & (\{q_0, q_1\}, q_0, \{q_0 \xrightarrow[1]{\neg\varphi^{\text{ini}}} q_1, \quad q_1 \xrightarrow[1]{true} q_1, \quad q_1 \xrightarrow[0]{true} \bullet\}),\\
C & = & (\{r_0\}, r_0, \{r_0 \xrightarrow[1]{true} r_0, \quad r_0 \xrightarrow[1]{\neg\varphi^{\text{fin}}} \bullet\}),\\
D & = & (\{s_0\}, s_0, \{s_0 \xrightarrow[0]{true} \bullet\}).
\end{array}$$

A accepts all the M configuration sequences in which at least one step is wrong, B all those that start with the wrong initial state, C all those that end in the wrong configuration, and D the empty sequence. We define $A_M = A \cup B \cup C \cup D$ using Theorem 2.13. A_M does not accept all the inputs in σ^* iff M halts on input $(0,0)$ with a non-zero output in r_1 (i.e. such sequence of configuration wouldn't be accepted by A_M). Since one can trivially construct an S-EFA *All* that accepts all strings we have that being able to check the equivalence of *All* and A_M would solve an undecidable problem. The undecidability of equivalence follows. □

We finally show that longer a look-ahead adds expressiveness.

Theorem 2.15. *For every k there exists an S-EFA with look-ahead $k+1$ that cannot be represented by any S-EFA with look-ahead k.*

Proof. Consider the S-EFA A over the theory of integers with one initial state q_0 and one final state q_1. The S-EFA A has only one transition between q_0 and q_1 of look-ahead $k+1$ with the following predicate $\psi(x_1, \ldots, x_{k+1}) = x_1 = x_2 = \ldots = x_{k+1}$. There does not exist an S-EFA B with look-ahead k equivalent to A. Let's assume B exists by way of contradiction. Since A only accepts strings of length $k+1$, B can only have finitely many paths from its initial state to any final state. Let's assume that every path has length 2 and it has guards of the form $\varphi_1(x_1 \ldots x_l)$ and $\varphi_2(x_{l+1} \ldots x_{k+1})$. We now must have that $\psi(x_1, \ldots, x_{k+1}) \equiv \bigvee \varphi_1(x_1 \ldots x_l) \wedge \varphi_2(x_{l+1} \ldots x_{k+1})$. However the predicate ψ does not admit such a representation as the variables and x_l and x_{l+1} do not belong to any binary relation and cannot be compared for equality. □

2.5 Equivalence of symbolic extended finite transducers

While the general equivalence problem of $\mathscr{T}_A = \mathscr{T}_B$ is already undecidable for very restricted classes of finite state transducers [Gri68], the problem is decidable for S-FTs in the single-valued case. More generally, one-equality of transductions (defined next) is decidable for S-FTs (over decidable label theories). In this section we show that equivalence of S-EFTs is in general undecidable, but decidable for Cartesian S-EFTs.

Definition 2.16. Functions $\mathbf{f}, \mathbf{g} : X \to 2^Y$ are *one-equal*, $\mathbf{f} \overset{1}{=} \mathbf{g}$, if for all $x \in X$, if $x \in \mathscr{D}(\mathbf{f}) \cap \mathscr{D}(\mathbf{g})$ then $|\mathbf{f}(x) \cup \mathbf{g}(x)| = 1$. Let

$$\mathbf{f} \uplus \mathbf{g}(x) \overset{\text{def}}{=} \begin{cases} \mathbf{f}(x) \cup \mathbf{g}(x), & \text{if } x \in \mathscr{D}(\mathbf{f}) \cap \mathscr{D}(\mathbf{g}); \\ \emptyset, & \text{otherwise.} \end{cases}$$

Proposition 2.17. $\mathbf{f} \overset{1}{=} \mathbf{g}$ *iff* $\mathbf{f} \uplus \mathbf{g}$ *is single-valued.*

Note that $\mathbf{f} \overset{1}{=} \mathbf{f}$ iff $\mathbf{f}$ is single-valued. Thus, one-equality is a more refined notion than single-valuedness, because an effective construction of $A \uplus B$ such that $\mathscr{T}_{A \uplus B} = \mathscr{T}_A \uplus \mathscr{T}_B$ may not always be feasible or even possible for some classes of transducers.

Definition 2.18. Functions $\mathbf{f}, \mathbf{g} : X \to 2^Y$ are *domain-equivalent* if $\mathscr{D}(\mathbf{f}) = \mathscr{D}(\mathbf{g})$.

Definitions 2.16 and 2.18 are lifted to transducers. For domain-equivalent single-valued transducers A and B, $A \overset{1}{=} B$ implies equivalence of A and B ($\mathscr{T}_A = \mathscr{T}_B$).

2.5.1 Equivalence of S-EFTs is undecidable

We now show that one-equality of S-EFTs over decidable label theories is undecidable in general.

Theorem 2.19 (One-equality)**.** *One-equality of S-EFTs with look-ahead 2 over quantifier-free successor arithmetic and tuples is undecidable.*

Proof. We give a reduction from the Domain Intersection problem of Theorem 2.11. Let A_1 and A_2 be S-EFAs with look-ahead 2 over quantifier-free successor arithmetic and tuples. We construct S-EFTs A_i', for $i \in \{1, 2\}$, as follows:

$$A_i' \;=\; (Q_{A_i}, q^0_{A_i}, \Delta_{A_i} \cup \{p \xrightarrow[k]{\varphi/[i]} \bullet \mid p \xrightarrow[k]{\varphi} \bullet \in F_{A_i}\}).$$

So $\mathscr{T}_{A_i'}(t) = \{[i]\}$ if $t \in \mathscr{D}(A_i)$ and $\mathscr{T}_{A_i'}(t) = \emptyset$ otherwise. Let $\mathbf{f} = \mathscr{T}_{A_1'} \uplus \mathscr{T}_{A_2'}$. So

- $|\mathbf{f}(t)| = 0$ iff $t \notin \mathscr{D}(A_1) \cup \mathscr{D}(A_2)$;
- $|\mathbf{f}(t)| = 1$ iff $t \in \mathscr{D}(A_1) \cup \mathscr{D}(A_2)$ and $t \notin \mathscr{D}(A_1) \cap \mathscr{D}(A_2)$;
- $|\mathbf{f}(t)| = 2$ iff $t \in \mathscr{D}(A_1) \cap \mathscr{D}(A_2)$.

It follows that $A_1' \overset{1}{=} A_2'$ iff (by Proposition 2.17) $\mathbf{f}$ is single-valued iff $\mathscr{D}(A_1) \cap \mathscr{D}(A_2) = \emptyset$. Now use Theorem 2.11. $\square$

Equivalence of symbolic finite transducers with look-back In this section we briefly describe a model that is tightly related to S-EFTs and for which equivalence is also undecidable. Symbolic finite transducers with look-back k (k-SLTs) [BB13] have a sliding window of size k that allows, in addition to the current input character, references of up to $k-1$ previous characters (using predicates of arity k). All the states of an SLTs are final and are associated with a constant output. Botinčan et al. [BB13] incorrectly claimed that equivalence of SLTs is decidable. We can prove using the same technique shown in the proof of Theorem 2.19 that equivalence is also undecidable for SLTs. We do not formally define SLTs here, but we briefly explain why the proof of Theorem 2.19 extends to this model. We let $\sigma = \mathbb{N}^3$ be the input sort and represent configurations of a Minsky machine in the same way we discussed in the proof of Theorem 2.11. Unlike S-EFTs, SLTs do not consume k symbols at a time and can therefore read the same input character multiple times using look-back. We can construct an SLT A that when reading each character in the input uses the predicate φ^{step} defined in Theorem 2.19 to check whether the *current* configuration of the Minsky machine follows from the previous one. Finally, the state following the accepting configuration outputs the constant a (every other state outputs ε). The SLT A outputs ε on every run that is not an accepting one for the Minsky machine and a on the accepting run (if it exists). We can now construct a simple SLT B with look-back 1 with one transition defined on the predicate *true* that always outputs ε. The two SLTs A and B are one-equal iff the Minsky machine does not halt (i.e. the first SLT never outputs a). The incorrect proof of decidability presented by Botinčan et al. [BB13] relies on intersecting the transitions of the two SLTs and then checking emptiness. Unfortunately, this does not work because the activation of a transition may depend on what transition was triggered in the previous step and simple cross-product construction does not take this into account.

2.5.2 Equivalence of Cartesian S-EFTs is Decidable

This is the main decidability result of the chapter and it extends the corresponding result for S-FTs presented by Veanes et al. [VHL+12, Theorem 1]. We use the following definitions. A transition, $p \xrightarrow[\ell]{\varphi/f} q$ where $\ell > 1$, φ is Cartesian and $\mathscr{W}(\varphi) = (a_1, \ldots, a_\ell)$, is represented, given $\varphi_i = \lambda x.\varphi(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_\ell)$, by the following path of *split* transitions,

$$p \xrightarrow[1]{\varphi_1/f} p_1 \xrightarrow[1]{\varphi_2/\bot} p_2 \cdots p_{\ell-1} \xrightarrow[1]{\varphi_\ell/\bot} q$$

where p_i for $1 \leq i < \ell$ are new *temporary* states. Let Δ^{s}_A denote such *split* view of Δ_A. Here we assume that all finalizers have look-ahead zero, since we do not assume S-EFTs here to be deterministic.

Example 2.20. *It is trivial to transform any S-EFT into an equivalent (possibly nondeterministic) form where all finalizers have zero look-ahead. Consider the S-EFT Base64encode in Example 2.4. In the last two finalizers, replace $\bullet$ with a new state p_1 and add the new finalizer $p_1 \xrightarrow[0]{true/\varepsilon} \bullet$.*

Definition 2.21. Let A and B be Cartesian S-EFTs with same input and output types and zero-look-ahead finalizers. The *product* of A and B is the following *product S-EFT* $A\times B$. The *initial state* $q^0_{A\times B}$ of $A\times B$ is (q^0_A, q^0_B). The states and transitions of $A\times B$ are obtained as the least fixed point of

$$\left.\begin{array}{c} (p,q) \in Q_{A\times B} \\ p \xrightarrow[1]{\varphi/f} p' \in \Delta^s_A \\ q \xrightarrow[1]{\psi/g} q' \in \Delta^s_B \end{array}\right\} \overset{IsSat(\varphi\wedge\psi)}{\Longrightarrow} (p',q') \in Q_{A\times B}, \quad (p,q) \xrightarrow[1]{\varphi\wedge\psi/(f,g)} (p',q') \in \Delta_{A\times B}.$$

Let $F_{A\times B}$ be the set of all rules $(p,q) \xrightarrow[0]{true/(v,w)} \bullet$ such that $p \xrightarrow[0]{true/v} \bullet \in F_A$, $q \xrightarrow[0]{true/w} \bullet \in F_B$, and $(p,q) \in Q_{A\times B}$. Finally, remove from $Q_{A\times B}$ (and $\Delta_{A\times B}$) all dead ends (non-initial states from which $\bullet$ is not reachable).[5]

We lift the definition of transductions to product S-EFTs. A pair-state $(p,q) \in Q_{A\times B}$ is *aligned* if all transitions from (p,q) have outputs (f,g) such that $f \neq \bot$ and $g \neq \bot$. The relation $\xrightarrow{/}\!\!\!\!\rightarrow_{A\times B}$ is defined analogously to S-EFTs.

Lemma 2.22 (Product). *For all aligned $(p,q) \in Q_{A\times B}$, $u \in \Sigma^*$, $v,w \in \Gamma^*$:*

$$(p,q) \xrightarrow{u/(v,w)}\!\!\!\!\rightarrow_{A\times B} \bullet \quad \Leftrightarrow \quad p \xrightarrow{u/v}\!\!\!\!\rightarrow_A \bullet \wedge q \xrightarrow{u/w}\!\!\!\!\rightarrow_B \bullet$$

Proof. This follows by induction on the input string u. The base case $u = \varepsilon$ is trivial as the product machine $A\times B$ outputs $(\varepsilon,\varepsilon)$ and A and B both output ε, or all outputs are undefined. The inductive case is $u = a_1 \cdots a_l u'$ where $l \geq 1$ and by induction hypothesis, for all p', q', u', v', w',

$$(p',q') \xrightarrow{u'/(v',w')}\!\!\!\!\rightarrow_{A\times B} \bullet \quad \Leftrightarrow \quad p' \xrightarrow{u'/v'}\!\!\!\!\rightarrow_A \bullet \wedge q' \xrightarrow{u'/w'}\!\!\!\!\rightarrow_B \bullet$$

If they exists, let $p \xrightarrow[l]{\varphi/f} p_1$ and $q \xrightarrow[l]{\psi/g} q_1$ be the A and B transitions out of p and q that are defined on $a_1 \ldots a_l$. The outputs of A and B on $a_1 \cdots a_l$ are $f(a_1,\ldots,a_l)$ and $g(a_1,\ldots,a_l)$ respectively. The corresponding transition in $A\times B$ is $(p,q) \xrightarrow[l]{\varphi\wedge\psi/(f,g)} (p_1,q_1)$. Since $a_1 \ldots a_l$ is a model of both φ and ψ, it is also a model of $\varphi \wedge \psi$. Therefore, the output of $A\times B$ on $a_1 \cdots a_l$ is $(f(a_1,\ldots,a_l), g(a_1,\ldots,a_l))$. Since by I.H. the outputs on the suffix u' also match this concludes the proof. □

[5] In our constructions we always remove unnecessary states to improve the efficiency of our algorithms.

We define also, for all $u \in \Sigma^*$, $\mathscr{T}_{A\times B}(u) \stackrel{\text{def}}{=} \{(v,w) \mid q^0_{A\times B} \xrightarrow{u/(v,w)} \bullet\}$ and $\mathscr{D}(A\times B) \stackrel{\text{def}}{=} \mathscr{D}(\mathscr{T}_{A\times B})$. Lemma 2.22 implies that $\mathscr{D}(A\times B) = \mathscr{D}(A) \cap \mathscr{D}(B)$ and $A \stackrel{1}{\neq} B$ iff there exists u and $v \neq w$ such that $(v,w) \in \mathscr{T}_{A\times B}(u)$.

Next we prove an *alignment* lemma that allows us either to effectively eliminate all non-aligned pair-states from $A\times B$ without affecting $\mathscr{T}_{A\times B}$ or to establish that $A \stackrel{1}{\neq} B$. A product S-EFT is *aligned* if all pair-states in it are aligned.

Lemma 2.23 (Alignment). *If $A \stackrel{1}{=} B$ then there exists an aligned product S-EFT that is equivalent to $A\times B$. Moreover, there is an effective procedure that either constructs it or else proves that $A \stackrel{1}{\neq} B$, if the label theory is decidable.*

Proof. The product $A\times B$ is incrementally transformed by eliminating non-aligned pair-states from it. Each iteration preserves equivalence. Initialize the search *frontier* to be $\{q^0_{A\times B}\}$. Pick (and remove) a state (p,q) from the frontier and consider all transitions starting from it. We also keep a set of visited states initialized to $\emptyset$. The main two cases are the following:

1. For all the transitions from (p,q) where both the A-output f and the B-output g have equal look-ahead (say $\ell = 2$),
$$(p,q) \xrightarrow[1]{\varphi/(f,g)} (p_1,q_1) \xrightarrow[1]{\psi/(\bot,\bot)} (p_2,q_2),$$
replace the path with the following combined transition with look-ahead 2
$$(p,q) \xrightarrow[2]{\lambda(x_0,x_1).\varphi(x_0)\wedge\psi(x_1)/(f,g)} (p_2,q_2)$$
and add (p_2,q_2) to the frontier unless (p_2,q_2) has already been visited. Note that $(p_2,q_2) \in Q_A \times Q_B$ and thus (p_2,q_2) is aligned.

2. Assume there are transitions where the A-output f is a σ^k/γ-sequence and the B-output g is a σ^ℓ/γ-sequence ($k \neq \ell$, say $k = 2$ and $\ell = 1$):
$$(p,q) \xrightarrow{\varphi/(f,g)} (p_1,q_1) \xrightarrow{\psi/(\bot,g_1)} (p_2,q_2).$$
So p_1 is temporary while q_1 is not.

   Decide if f can be split into two independent σ/γ-sequences f_1 and f_2 such that for all $a_1 \in [\![\varphi]\!]$ and $a_2 \in [\![\psi]\!]$, $[\![f]\!](a_1,a_2) = [\![f_1]\!](a_1) \cdot [\![f_2]\!](a_2)$. To do so, choose h_1 and h_2 such that $f = \lambda(x,y).h_1(x,y) \cdot h_2(x,y)$ (note that the total number of such choices is $|f| + 1$ where $|f|$ is the length of the output sequence), let $f_1 =$

$\lambda x.h_1(x, \mathscr{W}(\psi))$, $f_2 = \lambda x.h_2(\mathscr{W}(\varphi), x)$ and check validity of the *split predicate*

$$\forall x\, y\, ((\varphi(x) \wedge \psi(y)) \Rightarrow f(x,y) = f_1(x) \cdot f_2(y)).$$

If there exists a valid split predicate then pick such f_1 and f_2, and replace the above path with

$$(p,q) \xrightarrow{\varphi/(f_1,g)} (p_1', q_1') \xrightarrow{\psi/(f_2,g_1)} (p_2, q_2)$$

where (p_1', q_1') is a new *aligned* pair-state added to the frontier.

Suppose that splitting fails. We show that $A \overset{1}{\neq} B$, by contradiction. Assume $A \overset{1}{=} B$.

Since splitting fails, the following *dependency* predicates are satisfiable:

$$\begin{aligned} D1 &= \lambda(x, x', y).(\varphi(x) \wedge \varphi(x') \wedge \psi(y) \wedge f(x,y) \neq f(x',y)), \\ D2 &= \lambda(x, y, y').(\varphi(x) \wedge \psi(y) \wedge \psi(y') \wedge f(x,y) \neq f(x,y')). \end{aligned}$$

Let $(a_1, a_1', a_2) = \mathscr{W}(D1)$ and $(e_1, e_2, e_2') = \mathscr{W}(D2)$. We proceed by case analysis over $|f|$. We know that $|f| \geq 1$, or else splitting is trivial.

(a) Assume first that $|f| = 1$. Let

$$[b] = [\![f]\!](a_1, a_2),\ [b'] = [\![f]\!](a_1', a_2),\ [d] = [\![f]\!](e_1, e_2),\ [d'] = [\![f]\!](e_1, e_2').$$

Thus $b \neq b'$ and $d \neq d'$. Since (p,q) is aligned, and (p_1, q_1) is reachable and alive (by construction of $A \times B$, $\bullet$ is reachable from (p_1, q_1)), there exists $\alpha, \beta \in \Sigma^*$, $u_1, u_2, v_1, v_2, v_3, v_4 \in \Gamma^*$, such that, by *IsSat*(*D1*),

$$\left.\begin{array}{l} p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \overset{[a_1,a_2]/[b]}{\twoheadrightarrow} p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\ q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[a_1]/[\![g]\!](a_1)} q_1 \overset{[a_2]\cdot\beta/v_2}{\twoheadrightarrow}_B \bullet \end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [b] \cdot u_2 = \\ v_1 \cdot [\![g]\!](a_1) \cdot v_2 \end{array}$$

$$\left.\begin{array}{l} p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \overset{[a_1',a_2]/[b']}{\twoheadrightarrow} p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\ q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[a_1']/[\![g]\!](a_1')} q_1 \overset{[a_2]\cdot\beta/v_2}{\twoheadrightarrow}_B \bullet \end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [b'] \cdot u_2 = \\ v_1 \cdot [\![g]\!](a_1') \cdot v_2 \end{array}$$

By $b \neq b'$, $|v_1| \leq |u_1| < |v_1 \cdot [\![g]\!](a_1)| = |v_1| + |g|$. Also, by *IsSat*(*D2*),

$$\left.\begin{array}{l} p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \overset{[e_1,e_2]/[d]}{\twoheadrightarrow} p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\ q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[e_1]/[\![g]\!](e_1)} q_1 \overset{[e_2]\cdot\beta/v_3}{\twoheadrightarrow}_B \bullet \end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [d] \cdot u_2 = \\ v_1 \cdot [\![g]\!](e_1) \cdot v_3 \end{array}$$

$$\left.\begin{array}{l} p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \overset{[e_1,e_2']/[d']}{\twoheadrightarrow} p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\ q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[e_1]/[\![g]\!](e_1)} q_1 \overset{[e_2']\cdot\beta/v_4}{\twoheadrightarrow}_B \bullet \end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [d'] \cdot u_2 = \\ v_1 \cdot [\![g]\!](e_1) \cdot v_4 \end{array}$$

By $d \neq d'$, $|v_1 \cdot [\![g]\!](e_1)| = |v_1| + |g| \leq |u_1|$. But $|u_1| < |v_1| + |g|$, which is a contradiction.

(b) Assume that $f = \lambda(x, y).[f_1(x, y), f_2(x, y)]$ (the case for $|f| > 2$ is similar). Since f cannot be split, either $f_1(x, y)$ depends on y or $f_2(x, y)$ depends on x.

i. Suppose $f_1(x, y)$ does not depend on y. Then $f_2(x, y)$ must depend on *both* x and y or else f can be split. We can then choose values $a_1, a_1', e_1 \in [\![\varphi]\!]$ and $a_2, e_2, e_2' \in [\![\psi]\!]$ such that $[\![f_2]\!](a_1, a_2) \neq [\![f_2]\!](a_1', a_2)$ and $[\![f_2]\!](e_1, e_2) \neq [\![f_2]\!](e_1, e_2')$. A contradiction is reached similarly to the case of $|f| = 1$.

ii. The case when $f_2(x, y)$ does not depend on x is symmetrical to (i).

iii. Suppose $f_1(x, y)$ depends on y and $f_2(x, y)$ depends on x. Choose $e_1, a_1, a_1' \in [\![\varphi]\!]$ and $e_2, e_2', a_2 \in [\![\psi]\!]$ such that $[\![f_1]\!](e_1, e_2) \neq [\![f_1]\!](e_1, e_2')$ and $[\![f_2]\!](a_1, a_2) \neq [\![f_2]\!](a_1', a_2)$. Let

$$b_1 = [\![f_1]\!](e_1, e_2),\ b_1' = [\![f_1]\!](e_1, e_2'),\ b_2 = [\![f_2]\!](a_1, a_2),\ b_2' = [\![f_2]\!](a_1', a_2)$$

Since (p, q) is input-synchronized, and (p_1, q_1) is reachable and alive, there exists $\alpha, \beta \in \Sigma^*$, $u_1, u_2, v_1, v_2, v_3, v_4 \in \Gamma^*$, such that:

$$\left.\begin{array}{l}
p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \xrightarrow{[a_1,a_2]/[\_,b_2]}\!\!\!\!\twoheadrightarrow p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\
q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[a_1]/[\![g]\!](a_1)} q_1 \overset{[a_2]\cdot\beta/v_2}{\twoheadrightarrow}_B \bullet
\end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [\_, b_2] \cdot u_2 = \\ v_1 \cdot [\![g]\!](a_1) \cdot v_2 \end{array}$$

$$\left.\begin{array}{l}
p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \xrightarrow{[a_1',a_2]/[\_,b_2']}\!\!\!\!\twoheadrightarrow p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\
q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[a_1']/[\![g]\!](a_1')} q_1 \overset{[a_2]\cdot\beta/v_2}{\twoheadrightarrow}_B \bullet
\end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [\_, b_2'] \cdot u_2 = \\ v_1 \cdot [\![g]\!](a_1') \cdot v_2 \end{array}$$

Since $b_2 \neq b_2'$ it must be that $|u_1| + 1 < |v_1 \cdot [\![g]\!](a_1)| = |v_1| + |g|$ Also,

$$\left.\begin{array}{l}
p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \xrightarrow{[e_1,e_2]/[b_1,\_]}\!\!\!\!\twoheadrightarrow p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\
q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[e_1]/[\![g]\!](e_1)} q_1 \overset{[e_2]\cdot\beta/v_3}{\twoheadrightarrow}_B \bullet
\end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [b_1, \_] \cdot u_2 = \\ v_1 \cdot [\![g]\!](e_1) \cdot v_3 \end{array}$$

$$\left.\begin{array}{l}
p_0 \overset{\alpha/u_1}{\twoheadrightarrow} p \xrightarrow{[e_1,e_2']/[b_1',\_]}\!\!\!\!\twoheadrightarrow p_2 \overset{\beta/u_2}{\twoheadrightarrow}_A \bullet \\
q_0 \overset{\alpha/v_1}{\twoheadrightarrow} q \xrightarrow{[e_1]/[\![g]\!](e_1)} q_1 \overset{[e_2']\cdot\beta/v_4}{\twoheadrightarrow}_B \bullet
\end{array}\right\} \Longrightarrow \begin{array}{l} u_1 \cdot [b_1', \_] \cdot u_2 = \\ v_1 \cdot [\![g]\!](e_1) \cdot v_4 \end{array}$$

Thus, since $b_1 \neq b_1'$, we have $|v_1 \cdot [\![g]\!](e_1)| = |v_1| + |g| \leq |u_1|$. But $|u_1| < |v_1| + |g|$.

The remaining cases are similar and effectively eliminate all non-aligned pair-states from $A \times B$ or else establish that $A \overset{1}{\not\equiv} B$. $\square$

Assume $A \times B$ is aligned and let $\lceil A \times B \rceil$ be the following *product S-FT* (product S-EFT all of whose transitions have look-ahead 1) over the input type σ^*. For each $p \xrightarrow[\ell]{\lambda\bar{x}.\varphi(x_0,x_1,\ldots,x_{\ell-1})/(f,g)} q$ in $\Delta_{A\times B}$ let y be a variable of sort σ^* and let φ_1 be the σ^*-predicate

$$\lambda y.\varphi(y[0],y[1],\ldots,y[\ell-1]) \wedge tail^{\ell}(y) = \varepsilon \bigwedge_{i<\ell} tail^{i}(y) \neq \varepsilon$$

where $y[i]$ is the term that accesses the i'th head of y and $tail^i(y)$ is the term that accesses the i'th tail of y. Lift f to the σ^*/γ-sequence $f_1 = \lambda y.f(y[0],y[1],\ldots,y[\ell-1])$ and lift g similarly to g_1. Add the rule $p \xrightarrow[1]{\varphi_1/(f_1,g_1)} q$ as a rule of $\lceil A \times B \rceil$. Thus, the domain type of $\mathscr{T}_{\lceil A\times B\rceil}$ is $(\Sigma^*)^*$ while the range type is $2^{\Gamma^*\times\Gamma^*}$. For $u = [u_0,u_1,\ldots,u_n] \in (\Sigma^*)^*$, let $\lfloor u \rfloor \overset{\text{def}}{=} u_0 \cdot u_1 \cdots u_n$ in Σ^*.

Lemma 2.24 (Grouping). *Assume $A \times B$ is aligned. For all $u \in \Sigma^*$ and $v,w \in \Gamma^*$: $(v,w) \in \mathscr{T}_{A\times B}(u)$ iff $\exists z(u = \lfloor z \rfloor \wedge (v,w) \in \mathscr{T}_{\lceil A\times B\rceil}(z))$.*

Proof. The type lifting does not affect the semantics of the label-theory specific transformations. □

Note that, $[[a_1,a_2],[a_3]]$ and $[[a_1],[a_2,a_3]]$ may be distinct inputs of the lifted product, while both correspond to the same flattened input $[a_1,a_2,a_3]$ of the original product. Intuitively, the internal subsequences correspond to input alignment boundaries of the two S-EFTs A and B.

So, in particular, grouping preserves the property: there exists an input u and outputs $v \neq w$ such that $(v,w) \in \mathscr{T}_{A\times B}(u)$. We use the following lemma that is extracted from the main result from Veanes et al. [VHL+12, Proof of Theorem 1].

Lemma 2.25 (S-FT One-Equality [VHL+12]). *Let C be a product S-FT over a decidable label theory. The problem of deciding if there exist u and $v \neq w$ such that $(v,w) \in \mathscr{T}_C(u)$ is decidable.*

We can now prove the main decidability result of this chapter.

Theorem 2.26 (Cartesian S-EFT one-equality). *One-equality of Cartesian S-EFTs over decidable label theories is decidable.*

Proof. Let A and B be Cartesian S-EFTs. Construct $A \times B$. By the Product lemma 2.22, $\mathscr{D}(A\times B) = \mathscr{D}(A) \cap \mathscr{D}(B)$ and $A \overset{1}{\neq} B$ iff there exist u and $v \neq w$ such that $(v,w) \in \mathscr{T}_{A\times B}(u)$. By using the Alignment lemma 2.23, construct aligned product S-FT C such that $\mathscr{T}_C = \mathscr{T}_{A\times B}$ or else determine that $A \overset{1}{\neq} B$. Now lift C to $\lceil C \rceil$, and by using the Grouping lemma 2.24, $A \overset{1}{\neq} B$ iff there exist u and $v \neq w$ such that $(v,w) \in \mathscr{T}_{\lceil C\rceil}(u)$.

Finally, observe that adding the sequence operations for accessing the head and the tail of sequences in the lifting construction do not affect, by themselves, decidability of the label theory. We can therefore apply Lemma 2.25. □

Since a Monadic S-EFT can be effectively transformed into an equivalent Cartesian S-EFT (Theorem 2.26) we get the following result.

Corollary 2.27 (Monadic S-EFT One-Equality)**.** *One-equality of monadic S-EFTs over decidable label theories is decidable.*

2.6 Composition of symbolic extended finite transducers

In this section we show few preliminary results on the problem of composing S-EFTs. We first show that S-EFTs and Cartesian S-EFTs are not closed under composition. Moreover, even if the composition of two S-EFTs is definable by another S-EFT, it is undecidable to compute such an S-EFT. Last, we give an incomplete algorithm for composing S-EFTs.

2.6.1 S-EFTs are not closed under composition

Given a function$\mathbf{f} : X \to 2^Y$ and a set of inputs $\mathbf{x} \subseteq X$, the lifting of function f to a set of inputs $\mathbf{x}$ is defined as $\mathbf{f}(\mathbf{x}) \stackrel{\text{def}}{=} \bigcup_{x \in \mathbf{x}} \mathbf{f}(x)$. Given two lifted functions $\mathbf{f} : X \to 2^Y$ and $\mathbf{g} : Y \to 2^Z$, we define their composition $\mathbf{f} \circ \mathbf{g}(x) \stackrel{\text{def}}{=} \mathbf{g}(\mathbf{f}(x))$. This definition follows the convention from Fülöp et al. [FV98], i.e., $\circ$ applies first $\mathbf{f}$, then $\mathbf{g}$, contrary to how $\circ$ is used for standard function composition. The intuition is that $\mathbf{f}$ corresponds to the relation $R_{\mathbf{f}} : X \times Y$, $R_{\mathbf{f}} \stackrel{\text{def}}{=} \{(x,y) \mid y \in \mathbf{f}(x)\}$, so that $\mathbf{f} \circ \mathbf{g}$ corresponds to the binary relation composition $R_{\mathbf{f}} \circ R_{\mathbf{g}} \stackrel{\text{def}}{=} \{(x,z) \mid \exists y (R_{\mathbf{f}}(x,y) \wedge R_{\mathbf{g}}(y,z))\}$.

Definition 2.28. A class of transducers C is closed under composition iff for every $\mathscr{T}_1$ and $\mathscr{T}_2$ that are C-definable $\mathscr{T}_1 \circ \mathscr{T}_2$ is also C-definable.

Theorem 2.29. *The composition of two Cartesian S-EFTs is not S-EFT-definable.*

Proof. We show two Cartesian S-EFTs whose composition cannot be expressed by any S-EFT. Let A be following S-EFT over $Int \to Int$

$$A = (\{q\}, q, \{q \xrightarrow[2]{true/[x_1,x_0]} q, q \xrightarrow[0]{true/[]} \bullet\})$$

and B be following S-EFT over $Int \rightarrow Int$

$$B = (\{q_0, q_1\}, q_0, \{q_0 \xrightarrow[1]{true/[x_0]} q_1, q_1 \xrightarrow[2]{true/[x_1,x_0]} q_1, q_1 \xrightarrow[1]{true/[x_0]} \bullet\}).$$

The two transformations behave as in the following examples:

$$\mathscr{T}_A([a_0, a_1, a_2, a_3, a_4, a_5, a_6, \ldots]) = [a_1, a_0, a_3, a_2, a_5, a_4, a_7, \ldots],$$

$$\mathscr{T}_B([b_0, b_1, b_2, b_3, b_4, b_5, \ldots]) = [b_0, b_2, b_1, b_4, b_3, b_6, b_5, \ldots].$$

When we compose $\mathscr{T}_A$ and $\mathscr{T}_B$ we get the following transformation:

$$\mathscr{T}_{A\circ B}([a_0, a_1, a_2, a_3, a_4, a_5, a_6, \ldots]) = [a_1, a_3, a_0, a_5, a_2, a_7, a_4, a_9, a_6, a_8, \ldots].$$

Intuitively, looking at $\mathscr{T}_{A\circ B}$ we can see that no finite look-ahead seems to suffice for this function. Formally, for each a_i such that $i \geq 0$, $\mathscr{T}_{A\circ B}$ is the following function:

- if $i = 1$, a_i is output at position 0;
- if i is even and greater than 1, a_i is output at position $i - 2$;
- if i is equal to $k - 2$ where k is the length of the input, a_i is output at position $k - 1$;
- if i is odd and different from $k - 2$, a_i is output at position $i + 2$.

It is easy to see that the above transformation cannot be computed by any S-EFT. Let's assume by contradiction that there exists an S-EFT that computes $\mathscr{T}_{A\circ B}$. We consider the S-EFT C with minimal look-ahead (let's say n) that computes $\mathscr{T}_{A\circ B}$.

We now show that on an input of length greater than $n + 2$, C will misbehave. The first transition of C that will apply to the input will have a look-ahead of size $l \leq n$. We now have three possibilities (the case $n = k - 2$ does not apply due to the length of the input):

$l = 1$*:* before outputting a_0 (at position 2) we need to output a_1 and a_3 which we have not read yet; this is a contradiction;

l is odd: position $l + 1$ is receiving a_{l-1} therefore C must output also the elements at position l; position l should receive a_{l+2} which is not reachable with a look-ahead of just l; this is a contradiction;

l is even and greater than 1*:* since $l > 1$, position l is receiving a_{l-2}. This means C is also outputting position $l - 1$; position $l - 1$ should receive a_{l+1} which is not reachable with a look-ahead of just l; this is a contradiction.

Thus, we have shown that n cannot be the minimal look-ahead which contradicts our initial hypothesis. Therefore $\mathscr{T}_{A\circ B}$ is not S-EFT-definable. $\square$

Corollary 2.30. *S-EFT are not closed under composition.*

We now show that in general the composition of two S-EFTs cannot be effectively constructed.

Theorem 2.31 (Composition is not constructible)**.** *Given two S-EFTs with look-ahead 2 over quantifier free successor arithmetic and tuples, A and B, such that composition $f = \mathscr{T}_{A\circ B}$ is S-EFT definable, we cannot effectively construct an S-EFT that defines the transformation f.*

Proof. Given a Minsky machine M, we construct two S-EFTs A and B such that their composition $A \circ B$ is definable by an S-EFT C such that:

- if M halts on input $(0,0)$ with a non-zero output in r_1, C is defined exactly on the run of M, and
- otherwise C is the empty transducer that is undefined on any input.

The proof is analogous to that of Theorem 2.11 and we use the composition of S-EFTs to simulate the intersection of two S-EFAs. Consider the predicates defined in the proof of Theorem 2.11. Let

$$\begin{aligned} A &= (\{p_0\}, p_0, \{p_0 \xrightarrow[2]{\varphi^{\text{step}}/[x_0,x_1]} p_0, \quad p_0 \xrightarrow[0]{true/\bullet} \}), \\ B &= (\{q_0,q_1\}, q_0, \{q_0 \xrightarrow[1]{\varphi^{\text{ini}}/[x_0]} q_1, \quad q_1 \xrightarrow[2]{\varphi^{\text{step}}/[x_0,x_1]} q_1, \quad q_1 \xrightarrow[1]{\varphi^{\text{fin}}/[x_0]} \bullet\}). \end{aligned}$$

This construction ensures that $\alpha \in \mathscr{D}(A \circ B)$ iff α is a valid run of M, i.e., $\alpha[0]$ is the initial configuration, $\alpha[i+1]$ is a valid successor configuration of $\alpha[i]$ (this follows from A for all odd $i < |\alpha|$ and from B for all even $i < |\alpha|$), and $\alpha[|\alpha|-1]$ is a halting configuration. Since M is deterministic and we fix the initial configuration, we have that $\mathscr{D}(A \circ B) = \{\alpha\}$ iff there exists α, such that M halts on α or $\mathscr{D}(A \circ B) = \emptyset$ otherwise. In the first case we will have $\mathscr{T}_{A\circ B}(\alpha) = \alpha$ and undefined on any input different from α. In the second case $\mathscr{T}_{A\circ B}$ is always undefined. In both cases $\mathscr{T}_{A\circ B}$ is S-EFT definable. Let's call C the S-EFT that implements $\mathscr{T}_{A\circ B}$. Since emptiness of S-EFT is a decidable problem, we can decide if M halts on input $(0,0)$ with a non-zero output in r_1. Since, the latter is an undecidable problem we have a contradiction and therefore the composition of two S-EFTs cannot be computed. $\square$

Composition of symbolic finite transducers with look-back In this section we discuss how SLTs compare to S-EFTs regarding composition. Recall that symbolic finite transducers with look-back k (k-SLTs) [BB13] have a sliding window of size k that allows, in addition to the current input character, references of up to $k-1$ previous characters (using predicates of arity k). All the states of an SLT are final and are associated with a constant output. Botinčan et al. [BB13] incorrectly claimed that SLTs are closed under composition. We briefly explain SLTs are not closed under composition using two SLTs over the sort $\sigma = \mathbb{N}$. Consider an SLT A that echoes the first element of the input, then deletes all the subsequent elements that are smaller or equal than 5, and finally outputs the first element that is greater than 5. For example on the input sequence $[1,2,4,2,5,6]$, the SLT A outputs the sequence $[1,6]$. We observe that on any input sequence of the form $a_1 \dots a_n$ such that for every $1 < i \leq n$, $a_i \leq 5$, and $a_n > 5$, the SLT A outputs the sequence $a_1 a_n$. Next consider the SLT B that given a sequence $a_1 a_2$ outputs the sequence $a_2 a_1$ (this can be implemented by an SLT with look-back 2). On any input sequence of the form $a_1 \dots a_n$ such that for every $1 < i \leq n$, $a_i \leq 5$, and $a_n > 5$, the function resulting by composing A with B should output the sequence $a_n a_1$. But this function can't be implemented using finite look-back. In particular, in order to output the symbol a_1 to the right of a_n, the symbol a_1 must be read by a transition that also reads the symbol a_n. Since n can be arbitrarily large, no finite look-back k would suffice.

2.6.2 A practical algorithm for composing S-EFTs

In this section we present a sound algorithm for composing S-EFTs that is not guaranteed to work in all cases, but works for many practical purposes [DV13b]. Given two S-EFTs, the algorithm first transforms them into Symbolic Transducers with registers [VHL+12] (S-Ts), and by using the fact that S-Ts are closed under composition, it computes their composition. The next step is a register elimination algorithm that tries to build an S-EFT that is equivalent to the composed S-T. This second step is sound but incomplete, and this is due to the fact that S-EFTs are not closed under composition. Recall that the closure fails already for restricted classes of S-EFTs (Corollary 2.30).

2.6.2.1 Symbolic transducers with registers

Registers provide a practical generalization of S-FTs. S-FTs with registers are called S-Ts, since their state space (reachable by registers) may no longer be finite. An S-T uses a register as a symbolic representation of states in addition to explicit (control) states.

The rules of an S-T are guarded commands with a symbolic input and output component that may use the register. By using Cartesian product types, multiple registers are represented with a single (compound) register. Equivalence of S-Ts is undecidable but S-Ts are closed under composition [VHL+12].

Definition 2.32. A *Symbolic Transducer* or *S-T* over $\sigma \to \gamma$ and register type τ is a tuple $A = (Q, q^0, \rho^0, R)$, where

- Q is a finite set of *states*;
- $q^0 \in Q$ is the *initial state*;
- $\rho^0 \in \mathcal{U}^\tau$ is the *initial register value*;
- R is a finite set of *rules* $R = \Delta \cup F$;
- Δ is a set of *transitions* $r = (p, \varphi, o, u, q)$, also denoted $p \xrightarrow{\varphi/o;u} q$, where
  - $p \in Q$ is the *start* state of r;
  - φ, the *guard* of r, is a $(\sigma \times \tau)$-predicate;
  - o, the *output* of r, is a finite sequence of $(\sigma \times \tau)/\gamma$-terms;
  - u, the *update* of r, is a $(\sigma \times \tau)/\tau$-term;
  - $q \in Q$ is the *end* state of r;
- F is a set of *final rules* $r = (p, \varphi, o)$, also denoted $p \xrightarrow{\varphi/o} \bullet$, where
  - $p \in Q$ is the *start* state of r;
  - φ, the *guard* of r, is a τ-predicate;
  - o, the *output* of r, is a finite sequence of τ/γ-terms.

All S-T rules in R have look-ahead 1 and all final rules have look-ahead 0. Longer look-aheads are not needed because registers can be used to record history, in particular they may be used to record previous input characters. A canonical way to do so is to let τ be σ^* that records previously seen characters, where initially $\rho^0 = []$, indicating that no input characters have been seen yet.

An S-EFT transition

$$p \xrightarrow[3]{\lambda(x_0,x_1,x_2).\varphi(x_0,x_1,x_2)/\lambda(x_0,x_1,x_2).o(x_0,x_1,x_2)} q$$

can be encoded as the following set of S-T rules where p_1 and p_2 are new states

$$p \xrightarrow{(\lambda(x,y).true)/\varepsilon;\lambda(x,y).cons(x,[])} p_1 \quad p_1 \xrightarrow{(\lambda(x,y).true)/\varepsilon;\lambda(x,y).cons(x,y)} p_2$$
$$p_2 \xrightarrow{(\lambda(x,y).\varphi(y[1],y[0],x))/\lambda(x,y).o(y[1],y[0],x);\lambda(x,y).[]} q$$

Final rules are encoded similarly. The only difference is that q above is $\bullet$ and the register update is not used in the third rule. An S-T rule $(p, \varphi, o, u, q) \in R$ denotes the following set of concrete transitions:

$$[\![(p, \varphi, o, u, q)]\!] \stackrel{\text{def}}{=} \{(p, s) \xrightarrow{a/[\![o]\!](a,s)} (q, [\![u]\!](a, s)) \mid (a, s) \in [\![\varphi]\!]\}.$$

A final S-T rule $(p, \varphi, o) \in F$ denotes the following set of concrete transitions:

$$[\![(p, \varphi, o)]\!] \stackrel{\text{def}}{=} \{(p, s) \xrightarrow{\varepsilon/[\![o]\!](s)} \bullet \mid s \in [\![\varphi]\!]\}.$$

The reachability relation $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\!\to_A q$ for $\boldsymbol{a} \in \Sigma^*, \boldsymbol{b} \in \Gamma^*, p \in (Q \times \mathcal{U}^\tau), q \in (Q \times \mathcal{U}^\tau) \cup \{\bullet\}$ is defined analogously to S-EFTs and $\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \mid (q^0, \rho^0) \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\!\to \bullet\}$.

The following example illustrates a simplified case when an S-EFT is turned into an S-T by saving a single character in a register, thus $\tau = \sigma$ in this case. The resulting S-T is then composed with itself. We abbreviate an S-EFT rule $p \xrightarrow[k]{\lambda\bar{x}.\varphi(\bar{x})/\lambda\bar{x}.o(\bar{x})} q$, where $|\bar{x}| = k$, by $p \xrightarrow[k]{\varphi(\bar{x})/o(\bar{x})} q$, and an S-T rule $p \xrightarrow{\lambda(x,y).\varphi(x,y)/\lambda(x,y).o(x,y);\lambda(x,y).u(x,y)} q$ by $p \xrightarrow{\varphi(x,y)/o(x,y);u(x,y)} q$.

Example 2.33. *Let A be an S-EFT with the single state q and the rules*

$$q \xrightarrow[2]{true/[x_1,x_0,x_0]} q, \quad q \xrightarrow[0]{true/[]} \bullet$$

For example, A transforms the input $[0, 1, 2, 3]$ into the output $[1, 0, 0, 3, 2, 2]$. The corresponding S-T of A, A^{st}, has the following transitions

$$q \xrightarrow{true/[];\, x} p, \quad p \xrightarrow{true/[x,y,y];\, 0} q, \quad q \xrightarrow{true/[]} \bullet$$

where y refers to the register, x refers to the current input, p is a new state, and the initial register value is assumed to be 0. The first transition outputs nothing, and saves the current character in the register. The second transition outputs the current character followed by outputting the register twice in a row, and resets the register back to its initial value. Let us consider the self-composition $A^{\mathrm{st}} \circ A^{\mathrm{st}}$. The register of $A^{\mathrm{st}} \circ A^{\mathrm{st}}$ has the type $\sigma \times \sigma$ whose first component y_0 is the register of first instance of A and whose second component y_1 is the register of the second instance of A. The composed transitions are

$$q_0 \xrightarrow{true/[];(x,y_1)} q_1, \quad q_1 \xrightarrow{true/[y_0,x,x];(0,y_0)} q_2,$$
$$q_2 \xrightarrow{true/[];(x,y_1)} q_3, \quad q_3 \xrightarrow{true/[x,y_1,y_1,y_0,y_0,y_0];(0,0)} q_0, \quad q_0 \xrightarrow{true/[]} \bullet$$

where q_0 is the initial state and initially register $y = (0,0)$, i.e., $y_0 = y_1 = 0$. Only q_0 is a final state (has a finalizer with the empty output).

2.6.2.2 A register elimination algorithm

In this section we describe an algorithm for transforming a class of S-Ts into S-EFTs. The core idea that underlies the register elimination algorithm is a symbolic generalization of the state elimination algorithm for converting an NFA into a regular expression (see e.g. [Yu97, Section 3.3]), which uses the notion of extended automata whose transitions are labelled by regular expressions. We present the algorithm and then an example of how it works. Here the labels of the S-T are predicates over sequences of elements of fixed look-ahead. Essentially the intermediate data structure of the algorithm is an "Extended S-T". We often abbreviate a transition $p \xrightarrow{\lambda(x,y).\varphi(x,y)/\lambda(x,y).o(x,y);\lambda(x,y).u(x,y)} q$ by $p \xrightarrow{\varphi(x,y)/o(x,y);u(x,y)} q$.

Input: S-T $A^{\sigma/\gamma;\tau}$.

Output: $\perp$ or an S-EFT over $\sigma \rightarrow \gamma$ that is equivalent to A.

1: Lift A to the input type σ^*. Replace each transition $p \xrightarrow{\varphi(x,y)/o(x,y);u(x,y)} q$ with the following transition where [] is the empty list of type σ^*:

$$p \xrightarrow[(1)]{x \neq [] \wedge \varphi(head(x),y)/o(head(x),y);(head(x),y)} q.$$

Intuitively, x must be a non-empty list, i.e., input [] is not allowed. In the rules we indicate, for clarity, that x is a list of length at least k by annotating the transition with subscript (k). Apply similar transformation to final rules.

2: Repeat the steps 2.a-2.c while there exists a state that does not have a self loop (a self loop is a transition whose start and end states are equal).

2.a: Choose a state p that is not the state of any self loop and is not the initial state.

2.b: For all transitions $p_1 \xrightarrow[(k)]{\varphi_1/o_1;u_1} p \xrightarrow[(\ell)]{\varphi_2/o_2:u_2} p_2$ in R:

- **let** $\varphi = \lambda(x,y).\varphi_1(x,y) \wedge \varphi_2(tail^k(x), u_1(x,y))$,
- **let** $o = \lambda(x,y).o_1(x,y) \cdot o_2(tail^k(x), u_1(x,y))$,
- **let** $u = \lambda(x,y).u_2(tail^k(x), u_1(x,y))$,
- **if** $IsSat(\varphi)$ **then** add $p_1 \xrightarrow[(k+\ell)]{\varphi/o;u} p_2$ as a new rule.

2.c: Delete the state p.

3: If all guards and outputs do not depend on the register, remove the register from all the rules in the S-T and return the resulting S-EFT. Otherwise return $\perp$.

After the first step, the original S-T accepts an input $[a_0, a_1, a_2]$ and produces output v iff the transformed S-T accepts $[cons(a_0, \_), cons(a_1, \_), cons(a_2, \_)]$ and produces output v, where the tails _ are unconstrained and irrelevant. Step 2 further groups the inputs characters, e.g., to $[cons(a_0, cons(a_1, \_)), cons(a_2, \_)]$, etc, while maintaining this input/output property with respect to the original S-T. Finally, in step 3, turning the S-T into an S-EFT, leads to elimination of the register as well as lowering of the character sort back to σ, and replacing each occurrence of $head(tail^k(x))$ with corresponding individual tuple element variable x_k. Soundness of the algorithm follows.

The algorithm omits several implementation aspects that have considerable effect on performance. One important choice is the order in which states are removed. In our implementation the states with lowest total number of incoming and outgoing rules are eliminated first. It is also important to perform the choices in an order that avoids unreachable state spaces. For example, the elimination of a state p in step 2 may imply that φ is unsatisfiable and consequently that p_2 is unreachable if the transition from p is the only transition leading to p_2. In this case, if p is reachable from the initial state, choosing p_2 before p in step 2 would be wasteful.

For the class of S-Ts in which no register value is passed *through* a loop the algorithm always succeeds. Intuitively this capture the cases in which there are no symbolic dependencies between separate loop iterations. In other words, the register is only used through a fixed number of states, and reset after that. The following example illustrates a case for which the register elimination succeeds.

Example 2.34. *Consider the S-T $A^{st} \circ A^{st}$ from Example 2.33. We follow the steps of the algorithm and show how the composed S-T can be transformed into an equivalent S-EFT.*
Step 1: *We lift the input type to σ^* so that a lifted character is a list (sequence of type σ^*). We let $|x| > k$ abbreviate the formula $\bigwedge_{i=0}^{k} tail^k(x) \neq []$. When $|x| > i$, we write x_i for the i'th element $head(tail^i(x))$ of x. Recall that y_0 is the first component of the register and y_1 is the second component. The lifted transitions of $A^{st} \circ A^{st}$ are:*

$$q_0 \xrightarrow[(1)]{x\neq[]/[];(x_0,y_1)} q_1, \quad q_1 \xrightarrow[(1)]{x\neq[]/[y_0,x_0,x_0];(0,x_0)} q_2,$$
$$q_2 \xrightarrow[(1)]{x\neq[]/[];(x_0,y_1)} q_3, \quad q_3 \xrightarrow[(1)]{x\neq[]/[x_0,y_1,y_1,y_0,y_0,y_0];(0,0)} q_0, \quad q_0 \xrightarrow{true/[]} \bullet$$

Repeat Step 2: *Choose $p = q_1$. Eliminate q_1 by merging the first two transitions. Here $k = \ell = 1$. Observe that x_0 in the second rule becomes $x_{0+k} = x_1$ and y_0 refers to the first* *sub-register update of the first transition, that is x_0. The merged transition is:*

$$q_0 \xrightarrow[(2)]{|x|>1/[x_0,x_1,x_1];(0,x_1)} q_2.$$

Next, choose $p = q_2$. Eliminate q_2 similarly by replacing the new transition to q_2 and the original transition from q_2 by:

$$q_0 \xrightarrow[(3)]{|x|>2/[x_0,x_1,x_1];(x_2,x_1)} q_3.$$

Finally, choose $p = q_3$. Eliminate q_3 similarly by replacing the new transition to q_3 and the original transition from q_3 by:

$$q_0 \xrightarrow[(4)]{|x|>3/[x_0,x_1,x_1,x_3,x_0,x_0,x_2,x_2,x_2];(0,0)} q_0.$$

Step 3: *It is now safe to remove the register because it is not being used any more in any guard or update. So the final S-EFT, say AA, has the rules*

$$q_0 \xrightarrow[4]{true/[x_0,x_1,x_1,x_3,x_0,x_0,x_2,x_2,x_2]} q_0, \quad q_0 \xrightarrow[0]{true/[]} \bullet$$

where the lifting has been undone and here each variable x_i is of type σ. For example

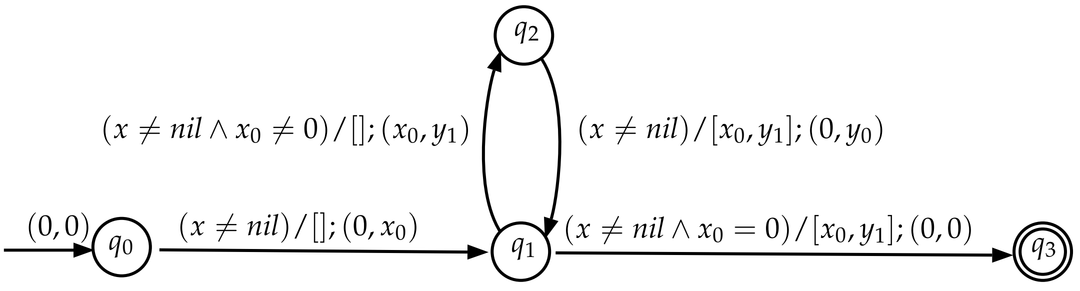
$$AA([0,1,2,3]) = [0,1,1,3,0,0,2,2,2].$$

We now discuss the cases when the algorithm fails. We first discuss the cases when it *should* fail, or else the algorithm would be unsound, and then identify two cases when it fails due to incompleteness (w.r.t. the class of S-Ts that have an equivalent S-EFT).

We know from Theorem 2.29 that already Cartesian S-EFTs are not closed under composition. For example, if we take the S-EFTs A and B from the proof of Theorem 2.29, first transform them into equivalent S-Ts and then compose the S-Ts, then the resulting S-T cannot be transformed back into an S-EFT. Although we know the algorithm will fail, it is nevertheless useful to see how this happens in the following example.

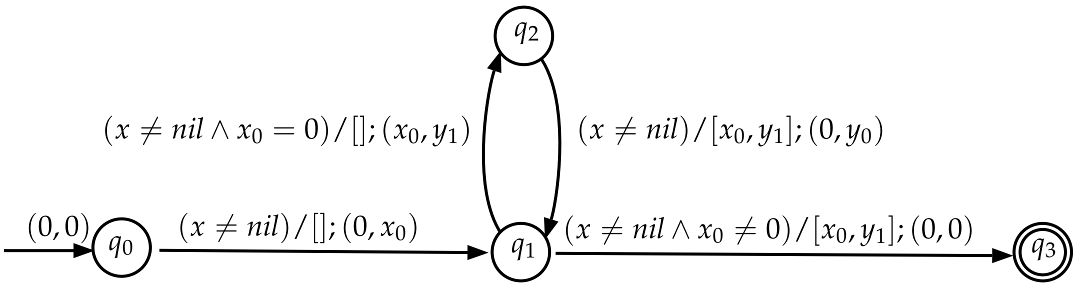
Example 2.35. *Consider S-EFTs A and B from the proof of Theorem 2.29. Modify B so that it is deterministic, by letting the guard on the self-loop on q_1 be $x_0 \neq 0$ and the guard on the finalizer from q_1 be $x_0 = 0$. The composition $B^{\text{st}} \circ A^{\text{st}}$, after lifting the input type, is then the following S-T (recall that $B^{\text{st}} \circ A^{\text{st}}(w) = A^{\text{st}}(B^{\text{st}}(w))$):*

If we apply the register elimination algorithm to this S-T, it may first eliminate the state q_2, by creating the transition $q_1 \xrightarrow[(2)]{|x|>1 \wedge x_0 \neq 0/[x_1,y_1];(0,x_0)} q_1$. After this step the algorithm stops and returns $\perp$.


q_2
$(x \neq nil \wedge x_0 \neq 0)/[];(x_0,y_1)$
$(x \neq nil)/[x_0,y_1];(0,y_0)$
$(0,0)$
q_0
$(x \neq nil)/[];(0,x_0)$
q_1
$(x \neq nil \wedge x_0 = 0)/[x_0,y_1];(0,0)$
q_3

Another reason why the algorithm fails for some inputs is due to Theorem 2.31, which states there are cases in which S-EFTs can be composed, but their composition cannot be effectively constructed. In particular composing the two S-EFTs of Theorem 2.31 requires *loop unrolling* in order to construct an equivalent S-EFT. In this case the algorithm fails, as shown by the following example.

Example 2.36. *Consider again the S-EFTs A and B from the proof of Theorem 2.29. This time modify B so that the guard on the self-loop on q_1 is $x_0 = 0$ and the guard on the finalizer from q_1 is $x_0 \neq 0$. The composition $B^{\mathrm{st}} \circ A^{\mathrm{st}}$, after lifting the input type, is then the following S-T, which is very similar to the one in Example 2.35:*


q_2
$(x \neq nil \wedge x_0 = 0)/[];(x_0,y_1)$
$(x \neq nil)/[x_0,y_1];(0,y_0)$
$(0,0)$
q_0
$(x \neq nil)/[];(0,x_0)$
q_1
$(x \neq nil \wedge x_0 \neq 0)/[x_0,y_1];(0,0)$
q_3

If we apply the register elimination algorithm to this S-T, it may, again, first eliminate the state q_2, by creating the transition $q_1 \xrightarrow[(2)]{|x|>1\wedge x_0=0/[x_1,y_1];(0,0)} q_1$. The algorithm is not able to detect that unrolling the loop once will remove the dependency on y_1 from the output (because y_1 will be fixed to 0 in all remaining iterations). The algorithm stops and returns $\perp$.

The register elimination algorithm also returns undefined when the register is used in a way that does not affect the transducer's semantics. For example, if there is an output element $r - r$ where r is an integer valued register, then the value will always be 0. But the algorithm does not perform any theory specific reasoning and will therefore not detect such cases.

2.6.2.3 A practical composition algorithm

We now have all the ingredients necessary to try to compose S-EFTs. The algorithm proceeds as follows. Given two S-EFTs A and B:

1. Compute two S-Ts A' and B' equivalent to A and B respectively;
2. Compute a S-T $C' = A' \circ B'$;
3. Run the register elimination algorithm on the S-T C' and if it terminates output the S-EFT C equivalent to $A \circ B$.

2.7 Experiments and applications

In this section we show how BEX can be used to verify real-world string coders and how S-EFTs can be used model problems in networking and functional programming.

2.7.1 Analysis of string encoders

We first discuss how BEX can prove the correctness of several real-world coders, and then present some scalability results.

2.7.1.1 Functional correctness

A string encoder E transforms input strings in a given format into output strings in a different format. A decoder D inverts such a transformation. For coders E and D to invert each-other, the following equalities should hold: $E \circ D \stackrel{1}{=} I$ and $D \circ E \stackrel{1}{=} I$ (where I is the identity transducer).

We illustrated in Example 2.4 how the BASE64 encoder and decoder can be modeled using Cartesian S-EFTs. Similarly, we can model BASE32, BASE16, and UTF8 coders. Using the equivalence and composition procedures presented in this chapter we proved that the equality presented above hold for all these coders. Table 2.1 shows the corresponding running times. The letters E, D, and I stand for encoder, decoder, and the identity transducer respectively. The first half of Table 2.1 shows the look-ahead sizes of both encoders and decoders, while the second half shows the running times for checking correctness. Composition times (typically 1-2 ms) are included in the measurements.

2.7.1.2 Running time analysis

In this section we analyze the cost of running our composition and equivalence algorithms on larger S-EFTs. Most of the compositions performed in this section will take more than 1 hour when trying to model the programs using finite state transducers.

	Look-ahead		Analysis (ms)	
	E	D	$E\circ D\overset{1}{=}I$	$D\circ E\overset{1}{=}I$
UTF8:	2	4	16	24
BASE64:	3	5	53	19
BASE32:	5	8	8	12
BASE16:	1	2	2	1

TABLE 2.1: Analysed coders with corresponding look-aheads and running times.

We consider consecutive compositions of encoders and decoders and analyze their correctness using 1-equality for S-EFTs. We define the following notation for consecutive composition of S-EFTs. Given an S-EFT P we define $P^1 \equiv P$ and $P^{i+1} \equiv P \circ P^i$. We fix E/D to be UTF8 encoder/decoder respectively, and verify analyze the running times of the following checks.

Equivalence for Enc/Dec: : $E^i \circ D^i \overset{1}{=} I$ for $1 \le i \le 9$, Figure 2.4(a);

Inequivalence for Enc/Dec: : $E^{i+1} \circ D^i \overset{1}{\neq} I$ for $1 \le i \le 9$, Figure 2.4(a);

Equivalence for Dec/Enc: : $D^i \circ E^i \overset{1}{=} I$ for $1 \le i \le 3$, Figure 2.4(b);

Inequivalence for Dec/Enc: : $D^i \circ E^{i+1} \overset{1}{\neq} I$ for $1 \le i \le 3$, Figure 2.4(b).

The top of Figure 2.4 shows the running times for the case in which we first encode and then decode. The figure plots the following measures where i varies between 1 and 9:

Composition: : cost of computing $E^{i+1} \circ D^i$ (we omit the cost of computing $E^i \circ D^i$ since it is almost equivalent);

Equivalence: : cost of checking $E^i \circ D^i \overset{1}{=} I$;

Inequivalence: : cost of checking $E^{i+1} \circ D^i \overset{1}{\neq} I$.

In this case the algorithm scales pretty well with the number of S-Ts. It is worth noticing that at every i we are analyzing the composition of $2i$ transducers in the case of equivalence and $2i + 1$ transducers in the case of inequivalence.

The bottom of Figure 2.4 shows the running times for the case in which we first decode and then encode. The plot has the same meaning as before, but in this case the running time increases at a faster pace. This happens because both the state space and the look-ahead become larger than for the case in which encode first. In the case in which we first encode, the number of states and transitions does not grow when i increases. However, when we first decode, we already reach a large number of states (3645) and transitions (6791) for $i = 3$. Moreover, while the size of the look-ahead remains 2 in the case of encode/decode, it grows exponentially with i when we first decode.

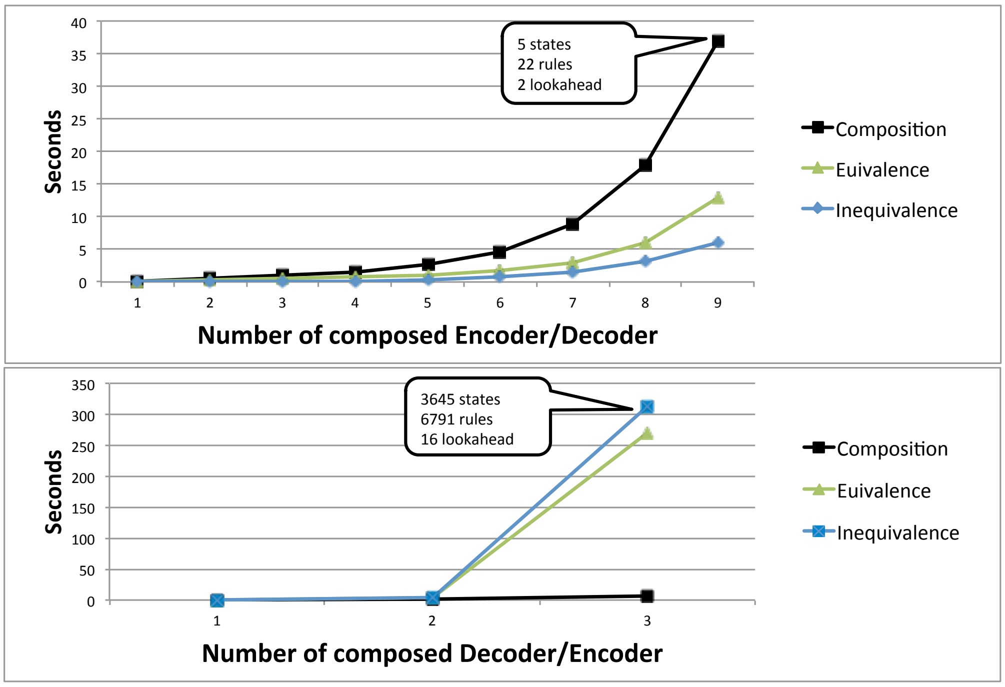


FIGURE 2.4: Running times for equivalence/inequivalence checking.

2.7.2 Deep packet inspection

Fast identification of network traffic patterns is of vital importance in network routing, firewall filtering, and intrusion detection. This task is addressed with the name "deep packet inspection" (DPI) [SEJK08]. Due to performance constraints, DPI must be performed in a single pass over the input. The simplest approach is to use DFAs and NFAs to identify patterns. These representations are either not succinct or not streamable. Extended Finite Automata (XFA) [SEJK08] make use of registers to reduce the state space while preserving determinism and therefore deterministic S-EFAs can be seen as a subclass of XFAs that are able to deal with finite look-ahead. Deterministic S-EFA can also represent the alphabet symbolically, which enables a new level of succinctness. We believe that deterministic S-EFAs can help achieve further succinctness. To support this hypothesis we observe that examples shown described by Smith et al. [SEJK08, Figure 2,3] can be represented as deterministic S-EFAs with few transitions. For example the language `^/\ncmd[^\n]{200}$`, which accepts all strings of the form '`\ncmds`' such that *s* has 200 symbols, can be succinctly modeled as a deterministic S-EFA with one transition! Moreover, the ability to compile S-EFA to Symbolic Automata with registers 2.6.2.1 makes this model appealing for efficient deterministic left-to-right DPI.

2.7.3 Verification of list manipulating programs

In our previous work [DVLM14] we showed how S-FTs can be used to verify pre and post conditions of list manipulating programs. However, S-FTs can only model programs in which each node in the output list depends on at most one node in the input list. S-EFTs can be used to mitigate this problem as they can be used to model sequential pattern matching. For example, the CAML guards

`x1::x2::xs -> (x1+x2)::(f2 xs)` and
`x1::x2::x3::xs -> (x1+x2+x3)::(f3 xs)`

can be naturally expressed as S-EFT transitions. Let's consider two functions f_2 and f_3, both of type *list int* $\rightarrow$ *list int*, that respectively contain the two guards defined above. These functions can be modeled as S-EFTs. Using the one-equality algorithm of Section 2.5 and the composition algorithm of Section 2.6.2.3 we were able to prove that $\forall l.f_3(f_2\ l) \overset{1}{=} f_2(f_3\ l)$ in less than 1 millisecond.

2.8 Related work

2.8.1 From S-FA/S-FT to S-EFA/S-EFT

The concept of automata with predicates instead of concrete symbols was first mentioned by Watson [Wat99] and was first discussed by Noord et al. [vNG01] in the context of natural language processing. Symbolic finite automata have been further studied in in the context of automata minimization [DV14].

Symbolic finite transducers (S-FTs) were originally introduced by Hooimeijer et al. [HLM+11] with a focus on security analysis of sanitizers. The formal foundations and the theoretical analysis of the underlying S-FT algorithms, in particular, an algorithm for one-equality of S-FTs, modulo a decidable background theory is studied by Veanes et al. [VHL+12]. The same work defined Symbolic Transducers (S-Ts). Full equivalence of finite state transducers is undecidable [Gri68], and already so for very restricted fragments [Iba78]. Equivalence of single-valued finite state transducers is decidable [Sch75] and the result has been extended to the finite-valued case [CK86, Web93]. Symbolic finite transducers have been extended to tree structures [DVLM14]. When reasoning about string coders, the use of symbolic alphabets is only beneficial when adding look-ahead, and S-EFTs are strictly more succinct than S-FTs.

An algorithm for checking whether a predicate is monadic (finite disjunction of Cartesian predicates) has been proposed by Veanes et al. [VBNB14]. This algorithm can be used to improve the expressiveness of BEX. The register elimination algorithm presented in Section 2.6.2.2 has also been extended to larger classes of S-Ts [VMML15]. In the future, we plan to extend BEX to support such an algorithm in order to support richer analysis.

2.8.2 Models that allow binary predicates

In recent years there has been considerable interest in automata that accept words over infinite alphabets [Seg06, KF94]. In this line of work, symbols can only be compared using equality and arbitrary predicates are not allowed, making the proposed models incomparable to those analyzed in this chapter. Here, we focus on proving negative and positive properties of S-EFAs and S-EFTs over arbitrary decidable Boolean algebras. While we do not investigate specific theories, it would be interesting to understand whether the properties we discussed hold when considering an alphabet theory that only supports equality.

Symbolic finite transducers with look-back k (k-SLTs) have a sliding window of size k that allows to refer to the $k-1$ previous characters [BB13] . We showed how k-SLTs are more expressive than S-EFTs, and contrary to the claim by Botinčan et al. [BB13], k-SLTs are *not* closed under composition, and equivalence of k-SLTs is *undecidable*.

Streaming string transducers [AC11] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders in order to maintain decidability of equivalence [AC11]. Streaming transducers are largely orthogonal to S-FTs or the extension of S-EFTs. For example, streaming transducers do not allow arithmetic, but can reverse the input, which is not possible with S-EFTs.

2.8.3 Models over finite alphabets

Extended Finite Automata (XFA) are introduced in [SEJK08] for network packet inspection. XFAs are a succinct representation of DFAs that uses registers to store and inspect values. History-based finite automata [KCTV07] are another extension of DFAs introduced in the context of network intrusion detection, that uses a single register (bit-vector) to keep track of the symbols read so far. In both models the register is used together with the input character to determine when a transition is enabled. Both these models focus on succinctness and the differ from S-EFAs in two ways: 1) they only support finite alphabets; and 2) they can relate symbols at arbitrary positions, while

S-EFAs can only relate adjacent positions. We have not investigated the application of S-EFAs to network packet inspection and network intrusion detection, but we think that S-EFAs can help achieving a further level of succinctness in these domains.

Extended Top-Down Tree Transducers [MGHK09] (ETTTs) are commonly used in natural language processing. ETTTs also allow finite look-ahead on transformation from trees to trees, but only support finite alphabets. The special case in which the input is a string (unary tree) is equivalent to S-EFTs over finite alphabets. This chapter focuses on S-EFTs over any decidable theory. We leave as future work extending S-EFTs to tree transformations.

Chapter 3

FAST: a transducer-based language for manipulating trees

"but most of all, samy is my hero"

— Samy Kamkar, *The Samy Worm*

3.1 Introduction

Trees are ubiquitous data structures and are used in a variety of applications in software engineering. For example, XML, HTML, and JSON are tree formats, compilers operate over an abstract syntax tree, and in natural language processing sentences are structured as trees. Providing better support for manipulating trees is clearly beneficial. In this chapter we focus on using variants of tree automata and tree transducers to design a language, FAST, that can analyze practical tree-manipulating programs.

3.1.1 Limitations of current models

Tree automata are used in variety of applications in software engineering, from analysis of XML programs [HP03] to type-checking [Sei94]. Tree transducers extend tree automata to model functions from trees to trees, and appear in fields such as natural language processing [MGHK09, PS12, MK08] and XML transformations [MBPS05]. While these formalisms are of immense practical use, they suffer from a major drawback: in the most common forms they can only handle finite alphabets. Moreover, in practice existing models do not scale well even for finite but large alphabets.

In order to overcome this limitation, *symbolic tree automata* (S-TAs) and *symbolic tree transducers* (S-TTs) [VB12b, FV14] support complex alphabets by allowing transitions to be labeled with formulas in a specified alphabet theory. While the concept is straightforward, traditional algorithms for deciding composition, equivalence, and other properties of finite automata and transducers *do not* immediately generalize. A notable example is the one we discussed in Chapter 2, where we showed that, while allowing finite state automata transitions to read subsequent inputs does not add expressiveness, in the symbolic case this extension makes most problems, such as checking equivalence, undecidable.

Symbolic tree automata still enjoy the closure and decidability properties of tree automata [VB12b] under the assumption that the alphabet theory forms a decidable Boolean algebra (i.e., closed under Boolean operations). In particular S-TAs are closed under Boolean operations, and enjoy decidable emptiness and equivalence.

A *symbolic tree transducer* (S-TT) traverses the input tree in a top-down fashion, processes one node at a time, and produces an output tree. This simple model can capture several natural language transformations and simple recursive programs that operate over trees. However, in most useful cases S-TTs are not closed under sequential composition [FV14]. This is a practical limitation when verifying even simple program properties.

3.1.2 Contributions

To overcome the limitations we just discussed, we introduce *symbolic tree transducers with regular look-ahead* (S-TTRs), which extend S-TTs with regular look-ahead [Eng77] — the capability of checking whether the subtrees of each processed node belong to some regular tree languages. S-TTRs support complex, potentially infinite alphabets, and we show that S-TTRs are closed under composition in most practical scenarios: two S-TTRs A and B can be composed into a single S-TTR $A \circ B$ if either A is single-valued (for every input it produces at most one output), or B is linear (it traverses each node in the tree at most once). Remarkably, the algorithm works for any decidable alphabet theory that forms an effective Boolean algebra.

We present FAST, a frontend programming language for S-TAs and S-TTRs. FAST (Functional abstraction of symbolic transducers) is a functional language that integrates symbolic tree automata and transducers with Z3 [DMB08], a state-of-the-art solver capable of supporting complex theories that range from data-types to non-linear real arithmetic.

We use FAST to model several real-world scenarios and analysis problems: we demonstrate applications to HTML sanitization, interference checking of augmented reality applications submitted to an app store, deforestation in functional language compilation, and analysis of functional programs over trees. We also sketch how FAST can capture simple CSS analysis tasks. All these problems require the use of symbolic alphabets.

This chapter is structured as follows:

- Section 3.2 introduces the language FAST using an example in the context of HTML sanitization;
- Section 3.3 presents a *theory of symbolic tree transducers with regular look-ahead* (S-TTR) and FAST, a transducer-based language founded on the theory of S-TTRs;
- Section 3.4 discusses a new algorithm for composing S-TTRs, along with a proof of correctness;
- Section 3.5 presents five concrete applications of FAST and shows how the closure under composition of S-TTR can be beneficial in practical settings;
- Section 3.6 compares FAST with previous domain specific languages for tree manipulation.

3.2 Overview of FAST

We use a simple scenario to illustrate the main features of the language FAST and the analysis enabled by the use of symbolic transducers. Here, we choose to model a basic HTML sanitizer. An HTML sanitizer is a program that traverses an input HTML document and removes or modifies nodes, attributes, and values that can cause malicious code to be executed on a server. Every HTML sanitizer works in a different way, but the general structure is as follows: 1) the input HTML is parsed into a DOM (Document Object Model) tree, 2) the DOM is modified by a sequence of sanitization functions $f_1, \ldots, f_n$, and 3) the modified DOM tree is transformed back into an HTML document[1]. In the following paragraphs we use FAST to describe some of the functions used during step 2. Each function f_i takes as input a DOM tree received from the browser's parser and transforms it into an updated DOM tree. As an example, the FAST program *sani* (Figure 3.1, line 30) traverses the input DOM and outputs a copy where

[1]Some sanitizers process the input HTML as a string, often causing the output not to be standards compliant.

```
1   // Datatype definition for HTML encoding
2   type HtmlE[tag : String]{nil(0), val(1), attr(2), node(3)}
3   // Language of well-formed HTML trees
4   lang nodeTree : HtmlE {
5           node(x1, x2, x3) given (attrTree x1) (nodeTree x2) (nodeTree x3)
6       |   nil() where (tag = "")   }
7   lang attrTree : HtmlE {
8           attr(x1, x2) given (valTree x1) (attrTree x2)
9       |   nil() where (tag = "")   }
10  lang valTree : HtmlE {
11          val(x1) where (tag ≠ "") given (valTree x1)
12      |   nil() where (tag = "")   }
13  // Sanitization functions
14  trans remScript : HtmlE → HtmlE {
15          node(x1, x2, x3) where (tag ≠ "script") to
16                                   (node [tag] x1 (remScript x2) (remScript x3))
17      |   node(x1, x2, x3) where (tag = "script") to x3
18      |   nil() to (nil [tag])   }
19  trans esc : HtmlE → HtmlE {
20          node(x1, x2, x3) to (node [tag] (esc x1) (esc x2) (esc x3))
21      |   attr(x1, x2) to (attr [tag] (esc x1) (esc x2))
22      |   val(x1) where (tag = "'" ∨ tag = """) to (val ["\"](val [tag] (esc x1)))
23      |   val(x1) where (tag ≠ "'" ∧ tag ≠ """) to (val [tag] (esc x1))
24      |   nil() to (nil [tag])   }
25  // Compose remScript and esc, restrict to well-formed trees
26  def rem_esc : HtmlE → HtmlE := (compose remScript esc)
27  def sani : HtmlE → HtmlE := (restrict rem_esc nodeTree)
28  // Language of bad outputs that contain a "script" node
29  lang badOutput : HtmlE {
30          node(x1, x2, x3) where (tag = "script")
31      |   node(x1, x2, x3) given (badOutput x2)
32      |   node(x1, x2, x3) given (badOutput x3)   }
33  // Check that no input produces a bad output
34  def bad_inputs : HtmlE := (pre-image sani badOutput)
35  assert-true (is-empty bad_inputs)
```

FIGURE 3.1: Implementation and analysis of an HTML sanitizer in FAST.

all subtrees in which the root is labeled with the string `"script"` have been removed, and all the characters `"'"` and `"""` have been escaped with a `"\"`.

The following informally describes each component of Figure 3.1. Line 2 defines the data-type *HtmlE* of "raw" trees. Each node of type *HtmlE* contains an attribute *tag* of type *string* and is built using one of the constructors *nil*, *val*, *attr*, or *node*. Each constructor has a number of children associated with it (2 for *attr*) and all such children are *HtmlE* nodes. We use the type *HtmlE* to model DOM trees. Since DOM trees are unranked (each node can have an arbitrary number of children), we will first encode them as ranked trees.

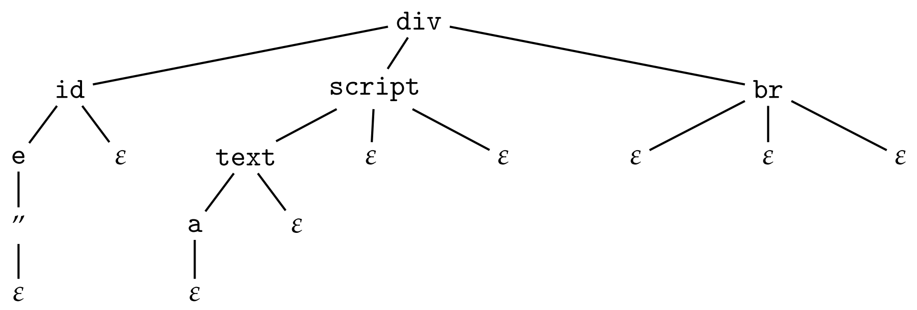


FIGURE 3.2: Encoding of `<div id='e"'><script>a</script></div><br />`.

We adopt a slight variation of a common binary encoding of unranked trees (Figure 3.2). We first informally describe the encoding and then show how it can be formalized in FAST. Each HTML node n is encoded as an $HtmlE$ element $node(x_1, x_2, x_3)$ with three children x_1, x_2, x_3 where: 1) x_1 encodes the list of attributes of n, 2) x_2 encodes the first child of n in the DOM, 3) x_3 encodes the next sibling of n, and 4) *tag* contains the node type of n (`div`, etc.). Each HTML attribute a with value s is encoded as an $HtmlE$ element $attr(x_1, x_2)$ with two children x_1, x_2 where: 1) x_1 encodes the value s (*nil* if s is the empty string), 2) x_2 encodes the list of attributes following a (*nil* if a is the last attribute), and 3) *tag* contains the name of a (`id`, etc.). Each non-empty string $w = s_1 \dots s_n$ is encoded as an $HtmlE$ element $val(x_1)$ where *tag* contains the string "s_1", and x_1 encodes the suffix $s_2 \dots s_n$. Each element *nil* has *tag* `""`, and can be seen as a termination operator for lists, strings, and trees. This encoding can be expressed in FAST (lines 4-12). For example, *nodeTree* (lines 4-6) is the language of correct HTML encodings (nodes): 1) the tree $node(x_1, x_2, x_3)$ is in the language *nodeTree* if x_1 is in the language *attrTree*, x_2 is in the language *nodeTree*, and x_3 is in the language *nodeTree*; 2) the tree *nil* is in *nodeTree* if its tag contains the empty string. The other language definitions are similar.

We now describe the sanitization functions. The transformation *remScript* (lines 14-18) takes an input tree t of type $HtmlE$ and produces an output tree of type $HtmlE$: 1) if $t = node(x_1, x_2, x_3)$ and its *tag* is different from `"script"`, *remScript* outputs a copy of t in which x_2 and x_3 are replaced by the results of invoking *remScript* on x_2 and x_3 respectively; 2) if $t = node(x_1, x_2, x_3)$ and its *tag* is equal to `"script"`, *remScript* outputs a copy of x_3, 3) if $t = nil$, *remScript* outputs a copy t. The transformation *esc* (lines 19-24) of type $HtmlE \rightarrow HtmlE$ escapes the characters `'` and `"`, and it outputs a copy of the input tree in which each node *val* with tag `"'"` or `"""` is pre-pended a node *val* with tag `"\"`. The transformations *remScript* and *esc* are then composed into a single transformation *rem_esc* (line 26). This is done using transducer composition. The square bracket syntax is used to represent the assignments to the attribute tag. One might notice that *rem_esc* also accepts input trees that are not in the language *nodeTree* and do not correspond to correct encodings. Therefore, we compute the transformation *sani*

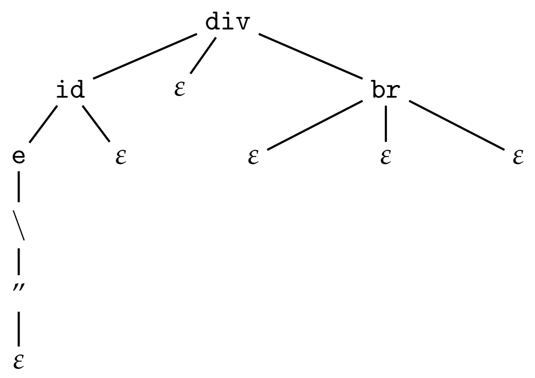


FIGURE 3.3: Result of applying *sani* to the tree in Figure 3.2.

(line 27), which is same as *rem_esc*, but restricted to only accept inputs in the language *nodeTree*. Sanitizing the tree of Figure 3.2 with the function *sani* yields the *HtmlE* tree corresponding to `<div id='e\"'></div><br />` (see Figure 3.3).

We can now use FAST to analyze the program *sani*. First, we define the language *bad_output* (lines 29-32), which accepts all the trees containing at least one node with tag `"script"`.[2] Next, using pre-image computation, we compute the language *bad_inputs* (line 34) of inputs that produce a bad output. Finally, if *bad_inputs* is the empty language, *sani* never produces bad outputs. When running this program in FAST this checking (line 35) fails, and FAST provides the following counterexample:

$$\mathit{node}\,[\texttt{"script"}]\;\mathit{nil}\;\mathit{nil}\;(\mathit{node}\,[\texttt{"script"}]\;\mathit{nil}\;\mathit{nil}\;\mathit{nil})$$

where we omit the attribute for the *nil* nodes. This is due to a bug in line 17, where the rule does not recursively invoke the transformation *remScript* on x_3. After fixing this bug the assertion becomes valid. In this example, we showed how in FAST simple sanitization functions can be first coded independently and then composed without worrying about efficiency. Finally, the resulting transformation can be analyzed using transducer based techniques.

3.3 Symbolic tree transducers and FAST

The concrete syntax of FAST is shown in Figure 3.4. We describe it in detail in the rest of the section. Nonterminals and meta-symbols are in italic. Constant expressions for strings and numbers use C# syntax [HWG03]. Additional well-formedness conditions (such as well-typed terms) are assumed to hold. FAST is designed for describing trees, tree languages, and functions from trees to trees. These are supported using *symbolic*

[2]This definition illustrates the nondeterministic semantics of FAST: a tree *t* belongs to *bad_output* if at least one of the three rules applies.

Indentifiers	ID : $(\mathtt{a}..\mathtt{z}\vert\mathtt{A}..\mathtt{Z}\vert\_)(\mathtt{a}..\mathtt{z}\vert\mathtt{A}..\mathtt{Z}\vert\_\vert\,\mathtt{.}\,\vert\mathtt{0}..\mathtt{9})^*$		
Basic types	σ : *String* | *Int* | $\mathbb{R}$ | $\mathbb{B}$ ...		
Built-in operators	op : `<` | `>` | `=` | `+` | `and` | `or` | ...		
Constructors	c : ID	Natural numbers	k : $\mathbb{N}$
Tree types	τ : ID	Language states	p : ID
Transformation states	q : ID	Attribute fields	x : ID
Subtree variables	y : ID		

Main definitions :

$$
\begin{array}{rcl}
\text{Fast} & ::= & \textbf{type}\ \tau\ \texttt{[}(x\texttt{:}\sigma)^*\texttt{]}\ \texttt{\{}(c\,\texttt{(}k\texttt{)}\,)^+\texttt{\}} \mid \textbf{tree}\ t\ \texttt{:}\ \tau\ \texttt{:=}\,\text{TR} \\
& \mid & \textbf{lang}\ p\ \texttt{:}\ \tau\ \texttt{\{}\ \text{Lrule}^+\ \texttt{\}} \mid \textbf{trans}\ q\ \texttt{:}\ \tau\ \to\ \tau\ \texttt{\{}\ \text{Trule}^+\ \texttt{\}} \\
& \mid & \textbf{def}\ p\ \texttt{:}\ \tau\ \texttt{:=}\,\text{L} \mid \textbf{def}\ q\ \texttt{:}\ \tau\ \to\ \tau\ \texttt{:=}\,\text{T} \\
& \mid & \textbf{assert-true}\ \text{A} \mid \textbf{assert-false}\ \text{A} \\
\text{Lrule} & ::= & c\,\texttt{(}y_1,\ldots,y_n\texttt{)}\ (\textbf{where}\ \text{Aexp})?\ (\textbf{given}\ (\ \texttt{(}\ p\ y\ \texttt{)}\,)^+)? \\
\text{Trule} & ::= & \text{Lrule}\ \textbf{to}\ \text{Tout} \\
\text{Tout} & ::= & y \mid \texttt{(}\ q\ y\ \texttt{)} \mid \texttt{(}\ c\,\texttt{[}\ \text{Aexp}^+\,\texttt{]}\ \text{Tout}^*\texttt{)} \\
\text{Aexp} & ::= & \text{ID} \mid \text{Const} \mid \texttt{(}\text{op Aexp}^+\texttt{)}
\end{array}
$$

Operations over languages, transductions, and trees :

$$
\begin{array}{rcl}
\text{L} & ::= & \texttt{(}\textbf{intersect}\ \text{L L}\texttt{)} \mid \texttt{(}\textbf{union}\ \text{L L}\texttt{)} \mid \texttt{(}\textbf{complement}\ \text{L}\texttt{)} \mid \\
& & \texttt{(}\textbf{difference}\ \text{L L}\texttt{)} \mid \texttt{(}\text{L}\texttt{)} \mid \texttt{(}\textbf{domain}\ \text{T}\texttt{)} \\
& \mid & \texttt{(}\textbf{pre-image}\ \text{T L}\texttt{)} \mid p \\
\text{T} & ::= & \texttt{(}\textbf{compose}\ \text{T T}\texttt{)} \mid \texttt{(}\textbf{restrict}\ \text{T L}\texttt{)} \mid \texttt{(}\textbf{restrict-out}\ \text{T L}\texttt{)} \mid q \\
\text{TR} & ::= & t \mid \texttt{(}\ c\,\texttt{[}\ \text{Aexp}^*\,\texttt{]}\ \text{TR}^*\texttt{)} \mid \texttt{(}\textbf{apply}\ \text{T TR}\texttt{)} \mid \texttt{(}\textbf{get-witness}\ \text{L}\texttt{)} \\
\text{A} & ::= & \text{L} == \text{L} \mid (\textbf{is-empty}\ \text{L}) \mid (\textbf{is-empty}\ \text{T}) \mid \text{TR} \in \text{T} \\
& \mid & \texttt{(}\textbf{type-check}\ \text{L T L}\texttt{)}
\end{array}
$$

FIGURE 3.4: Syntax of FAST.

tree automata (*S-TAs*) and *symbolic tree transducers with regular look-ahead* (*S-TTRs*). This section covers these objects and how they describe the semantics of FAST.

3.3.1 Background

All definitions are parametric with respect to a given background theory, called a *label theory*, over a fixed background structure with a recursively enumerable universe of elements. Such a theory is allowed to support arbitrary operations (such as addition, etc.), however all the results in the following only require it to be 1) closed under Boolean operations and equality, and 2) decidable (quantifier free formulas with free variables can be checked for satisfiability).

We use λ-expressions for defining anonymous functions called *λ-terms* without having to name them explicitly. In general, we use the standard first-order logic and follow the notational used by Veanes et al. [VHL+12]. We write $f(v)$ for the functional application of the λ-term f to the term v, σ for a type, and $\mathcal{U}^\sigma$ for the universe of elements of type

σ. A σ-predicate is a λ-term $\lambda x.\varphi(x)$ where x has type σ and φ is a formula for which the free variables $FV(\varphi)$ are contained in $\{x\}$. Given a σ-predicate φ, $[\![\varphi]\!]$ denotes the set of all $a \in \mathcal{U}^\sigma$ such that $\varphi(a)$. The set of σ-predicates is denoted by $\Psi(\sigma)$. Given a type σ (such as *int*), we extend the universe with σ-labeled finite trees as an algebraic datatype $\mathcal{T}^\sigma_\Sigma$ where Σ is a finite set of *tree constructors* f with *rank* $R(f) \geq 0$; f has type $\sigma \times (\mathcal{T}^\sigma_\Sigma)^{R(f)} \to \mathcal{T}^\sigma_\Sigma$.[3] Let $\Sigma(k) \stackrel{\text{def}}{=} \{f \in \Sigma \mid R(f) = k\}$. We call $\mathcal{T}^\sigma_\Sigma$ a *tree type*. We require that $\Sigma(0)$ is *non-empty* so that $\mathcal{U}^{\mathcal{T}^\sigma_\Sigma}$ is non-empty. We write $f[a](\bar{u})$ for $f(a, \bar{u})$ and abbreviate $f[a]()$ by $f[a]$.

Example 3.1. *The* FAST *program in Figure 3.1, declares* $HtmlE = \mathcal{T}^{String}_\Sigma$ *over* $\Sigma = \{nil, val, attr, node\}$, *where* $R(nil) = 0$, $R(val) = 1$, $R(attr) = 2$, *and* $R(node) = 3$. *For example* $attr[\texttt{"a"}](nil[\texttt{"b"}], nil[\texttt{"c"}])$ *is in* $\mathcal{U}^{\mathcal{T}^{String}_\Sigma}$.

We write $\bar{e}$ for a tuple (sequence) of length $k \geq 0$ and denote the i'th element of $\bar{e}$ by e_i for $1 \leq i \leq k$. We also write $(e_i)^k_{i=1}$ for $\bar{e}$. The empty tuple is $()$ and $(e_i)^1_{i=1} = e_1$. We use the following operations over k-tuples of sets. If $\bar{X}$ and $\bar{Y}$ are k-tuples of sets then $\bar{X} \uplus \bar{Y} \stackrel{\text{def}}{=} (X_i \cup Y_i)^k_{i=1}$. If $\bar{X}$ is a k-tuple of sets, $j \in \{1, \ldots, k\}$ and Y is a set then $(\bar{X} \uplus_j Y)$ is the k-tuple $(\textit{if } i{=}j \textit{ then } X_i{\cup}Y \textit{ else } X_i)^k_{i=1}$.

3.3.2 Alternating symbolic tree automata

We introduce and develop the basic theory of alternating symbolic tree automata, which adds a form of alternation to the basic definition originally presented by Veanes et al. [VB12b]. We decide to use alternating S-TAs instead of their non-alternating counterpart because they are succinct and arise naturally when composing tree transducers.

Definition 3.2. An *Alternating Symbolic Tree Automaton (Alternating S-TA) A* is a tuple $(Q, \mathcal{T}^\sigma_\Sigma, \delta)$, where Q is a finite set of *states*, $\mathcal{T}^\sigma_\Sigma$ is a *tree type*, and $\delta \subseteq \bigcup_{k \geq 0}(Q \times \Sigma(k) \times \Psi(\sigma) \times (2^Q)^k)$ is a finite set of *rules* $(q, f, \varphi, \bar{\ell})$, where q is the *source state*, f the *symbol*, φ the *guard*, and $\bar{\ell}$ the *look-ahead*.

For $q \in Q$, $\delta(q) \stackrel{\text{def}}{=} \{r \in \delta \mid \text{the source state of } r \text{ is } q\}$. In FAST $\delta(q)$ is

$$\textbf{lang}\ q : \tau\ \{c(\bar{y})\ \textbf{where}\ \varphi(\bar{x})\ \textbf{given}\ \bar{\ell}(\bar{y}) \mid \ldots\}.$$

Next, we define the semantics of an S-TA $A = (Q, \mathcal{T}^\sigma_\Sigma, \delta)$.

[3]If $R(f) = 0$ then f has type $\sigma \to \mathcal{T}^\sigma_\Sigma$.

```
type BT[i : Int]{L(0), N(2)}
lang p : BT {
        L() where (i > 0)
    |   N(x,y) given (p x) (p y)
}
lang o : BT {
        L() where (odd i)
    |   N(x,y) given (o x) (o y)
}
lang q : BT { N(x,y) given (p y) (o y) }
```

FIGURE 3.5: Tree languages in FAST.

Definition 3.3. For every state $q \in Q$ the *language of A at q*, is the set

$$\mathbf{L}_A^q \stackrel{\text{def}}{=} \{f[a](\bar{t}) \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma} \mid (q,f,\varphi,\bar{\ell}) \in \delta,\ a \in [\![\varphi]\!],\ \bigwedge_{i=1}^{R(f)} \bigwedge_{p \in \ell_i} t_i \in \mathbf{L}_A^p\}.$$

Each look-ahead set ℓ_i is treated as a *conjunction* of conditions. If ℓ_i is *empty* then there are no restrictions on the i'th subtree t_i. We extend the definition to all $\mathbf{q} \subseteq Q$:

$$\mathbf{L}_A^{\mathbf{q}} \stackrel{\text{def}}{=} \begin{cases} \bigcap_{q \in \mathbf{q}} \mathbf{L}_A^q, & \text{if } \mathbf{q} \neq \emptyset; \\ \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}, & \text{otherwise.} \end{cases}$$

Example 3.4. *Consider the* FAST *program in Figure 3.5 An equivalent S-TA A over $\mathcal{T}_{BT}^{\mathtt{Int}}$ has states $\{o,p,q\}$ and rules*

$$\begin{aligned}\{\ &(p,\mathtt{L},\lambda x.x>0,()),\ (p,\mathtt{N},\lambda x.true,(\{p\},\{p\})),\\ &(o,\mathtt{L},\lambda x.odd(x),()),\ (o,\mathtt{N},\lambda x.true,(\{o\},\{o\})),\\ &(q,\mathtt{N},\lambda x.true,(\emptyset,\{p,o\}))\ \}.\end{aligned}$$

Since the first subtree in the definition of q is unconstrained, the corresponding component in the last rule is empty. The definition for q has no case for `L`*, so there is no rule.*

In the following we say S-TA for alternating S-TA.[4]

Definition 3.5. A is *normalized* if, for all $(p,f,\varphi,\bar{\ell}) \in \delta$ and all i with $1 \leq i \leq R(f)$, ℓ_i is a singleton set.

For example, the S-TA in Example 3.4 is not normalized because of the rule with source q.

[4]When compared to the model in in the TATA book [CDG+07], the S-TAs defined above are "almost" alternating, in the sense that they only allow disjunctions of conjunctions, rather than arbitrary positive Boolean combinations. Concretely, the look-ahead of a rule r corresponds to a *conjunction* of states, while several rules from the same source state provide a *disjunction* of cases.

Normalization. The normalization procedure is a subset construction that removes the sources of alternation from an S-TA and makes sure that each lookahead component is a singleton set. Intuitively, alternating rules are merged by taking the conjunction of their predicates and the union of their activation states.

Let $A = (Q, \mathcal{T}^\sigma_\Sigma, \delta)$ be an S-TA. We compute *merged rules* $(\boldsymbol{q}, f, \varphi, \bar{\boldsymbol{\rho}})$ over *merged states* $\boldsymbol{q} \in 2^Q$, where $\bar{\boldsymbol{\rho}} \in (2^Q)^{R(f)}$. For $f \in \Sigma$, let $\boldsymbol{\delta}^f = \bigcup_{\boldsymbol{p} \subseteq Q} \boldsymbol{\delta}^f(\boldsymbol{p})$, where

$$\begin{array}{rcl} \boldsymbol{\delta}^f(\emptyset) & = & \{(\emptyset, f, \emptyset, (\emptyset)_{i=1}^{R(f)})\}, \\ \boldsymbol{\delta}^f(\boldsymbol{p} \cup \boldsymbol{q}) & = & \{r \mathbin{/\!\!\!\backslash\!\!\backslash} s \mid r \in \boldsymbol{\delta}^f(\boldsymbol{p}), s \in \boldsymbol{\delta}^f(\boldsymbol{q})\}, \\ \boldsymbol{\delta}^f(\{p\}) & = & \{(\{p\}, f, \{\varphi\}, \bar{\rho}) \mid (p, f, \varphi, \bar{\rho}) \in \delta\}, \end{array}$$

and where *merge* operation $\mathbin{/\!\!\!\backslash\!\!\backslash}$ over merged rules is defined as follows:

$$(\boldsymbol{p}, f, \boldsymbol{\varphi}, \bar{\boldsymbol{p}}) \mathbin{/\!\!\!\backslash\!\!\backslash} (\boldsymbol{q}, f, \boldsymbol{\psi}, \bar{\boldsymbol{q}}) \stackrel{\text{def}}{=} (\boldsymbol{p} \cup \boldsymbol{q}, f, \boldsymbol{\varphi} \cup \boldsymbol{\psi}, \bar{\boldsymbol{p}} \uplus \bar{\boldsymbol{q}}).$$

Definition 3.6. The *normalized form of A* is the S-TA

$$\mathcal{N}(A) \stackrel{\text{def}}{=} (2^Q, \mathcal{T}^\sigma_\Sigma, \{(\boldsymbol{p}, f, \bigwedge \boldsymbol{\varphi}, (\{\boldsymbol{q}_i\})_{i=1}^{R(f)}) \mid f \in \Sigma, (\boldsymbol{p}, f, \boldsymbol{\varphi}, \bar{\boldsymbol{q}}) \in \boldsymbol{\delta}^f\}).$$

The original rules of the normalized form are precisely the ones for which the states are singleton sets in 2^Q. In practice, merged rules are computed lazily starting from the initial state. Merged rules with unsatisfiable guards $\boldsymbol{\varphi}$ are eliminated eagerly. New concrete states are created for all the reachable merged states. Finally, the normalized S-TA is cleaned by eliminating states that accept no trees, e.g., by using elimination of useless symbols from a context-free grammar [HU79, p. 88–89].

As expected, normalization preserves the language semantics of S-TAs.

Theorem 3.7. *For all* $\mathbf{q} \subseteq Q$*,* $\mathbf{L}^{\mathbf{q}}_A = \mathbf{L}^{\mathbf{q}}_{\mathcal{N}(A)}$.

Proof. The case when $\mathbf{q} = \emptyset$ is clear because the state $\emptyset$ in $\mathcal{N}(A)$ has the same semantics as $\mathbf{L}^{\emptyset}_A$. Assume $\mathbf{q} \neq \emptyset$. We show (3.1) for all $t \in \mathcal{T}^\sigma_\Sigma$:

$$t \in \mathbf{L}^{\mathbf{q}}_A \Leftrightarrow t \in \mathbf{L}^{\mathbf{q}}_{\mathcal{N}(A)} \tag{3.1}$$

The proof is by induction over the height of t. As the base case assume $t = f[a]$ where $R(f) = 0$. Then

$$\begin{array}{rcl} f[a] \in \mathbf{L}_A^{\mathbf{q}} & \Leftrightarrow & \forall q \in \mathbf{q}(f[a] \in \mathbf{L}_A^q) \\ & \Leftrightarrow & \forall q \in \mathbf{q}(\exists \varphi((q, f, \varphi, ()) \in \delta, a \in [\![\varphi]\!])) \\ & \Leftrightarrow & \forall q \in \mathbf{q}(\exists \varphi((\{q\}, f, \{\varphi\}, ()) \in \delta^f(\{q\}), a \in [\![\varphi]\!])) \\ & \overset{\text{def of } \bigwedge\!\!\!\bigwedge}{\Leftrightarrow} & \exists \boldsymbol{\varphi}((\mathbf{q}, f, \boldsymbol{\varphi}, ()) \in \delta^f(\mathbf{q}), a \in [\![\bigwedge \boldsymbol{\varphi}]\!]) \\ & \Leftrightarrow & f[a] \in \mathbf{L}_{\mathcal{N}(A)}^{\mathbf{q}}. \end{array}$$

We prove the induction case next. For ease of presentation assume f is binary and $\mathbf{q} = \{q_1, q_2\}$. As the induction case consider $t = f[a](t_1, t_2)$.

$$\begin{array}{rcl} t \in \mathbf{L}_A^{\mathbf{q}} & \Leftrightarrow & \bigwedge_{i=1}^{2} t \in \mathbf{L}_A^{q_i} \\ & \Leftrightarrow & \bigwedge_{i=1}^{2} \exists \varphi^i, \boldsymbol{p}_1^i, \boldsymbol{p}_2^i : (q_i, f, \varphi^i, (\boldsymbol{p}_1^i, \boldsymbol{p}_2^i)) \in \delta_A, a \in [\![\varphi^i]\!], t_1 \in \mathbf{L}_A^{\boldsymbol{p}_1^i}, t_2 \in \mathbf{L}_A^{\boldsymbol{p}_2^i} \\ & \Leftrightarrow & \bigwedge_{i=1}^{2} \exists \varphi^i, \boldsymbol{p}_1^i, \boldsymbol{p}_2^i : (\{q_i\}, f, \{\varphi^i\}, (\boldsymbol{p}_1^i, \boldsymbol{p}_2^i)) \in \delta^f, a \in [\![\varphi^i]\!], t_1 \in \mathbf{L}_A^{\boldsymbol{p}_1^i}, t_2 \in \mathbf{L}_A^{\boldsymbol{p}_2^i} \\ & \overset{\text{def of } \bigwedge\!\!\!\bigwedge}{\Leftrightarrow} & \exists \varphi^1, \boldsymbol{p}_1^1, \boldsymbol{p}_2^1, \varphi^2, \boldsymbol{p}_1^2, \boldsymbol{p}_2^2 : t_1 \in \mathbf{L}_A^{\boldsymbol{p}_1^1} \cap \mathbf{L}_A^{\boldsymbol{p}_1^2}, t_2 \in \mathbf{L}_A^{\boldsymbol{p}_2^1} \cap \mathbf{L}_A^{\boldsymbol{p}_2^2} \\ & & (\mathbf{q}, f, \{\varphi^1, \varphi^2\}, (\boldsymbol{p}_1^1 \cup \boldsymbol{p}_1^2, \boldsymbol{p}_2^1 \cup \boldsymbol{p}_2^2)) \in \delta^f, a \in [\![\varphi^1 \wedge \varphi^2]\!] \\ & \Leftrightarrow & \exists \varphi, \boldsymbol{p}_1, \boldsymbol{p}_2 : t_1 \in \mathbf{L}_A^{\boldsymbol{p}_1}, t_2 \in \mathbf{L}_A^{\boldsymbol{p}_2}, (\mathbf{q}, f, \varphi, (\{\boldsymbol{p}_1\}, \{\boldsymbol{p}_2\})) \in \delta_{\mathcal{N}(A)}, a \in [\![\varphi]\!] \\ & \overset{\text{IH}}{\Leftrightarrow} & \exists \varphi, \boldsymbol{p}_1, \boldsymbol{p}_2 : t_1 \in \mathbf{L}_{\mathcal{N}(A)}^{\boldsymbol{p}_1}, t_2 \in \mathbf{L}_{\mathcal{N}(A)}^{\boldsymbol{p}_2} \\ & & (\mathbf{q}, f, \varphi, (\{\boldsymbol{p}_1\}, \{\boldsymbol{p}_2\})) \in \delta_{\mathcal{N}(A)}, a \in [\![\varphi]\!] \\ & \Leftrightarrow & t \in \mathbf{L}_{\mathcal{N}(A)}^{\mathbf{q}}. \end{array}$$

The theorem follows by the induction principle. □

Checking whether $\mathbf{L}_A^q \neq \emptyset$ can be done by first normalizing A, then removing unsatisfiable guards using the decision procedure of the theory $\Psi(\sigma)$, and finally using that emptiness for classic tree automata is decidable.

Proposition 3.8. *The non-emptiness problem of S-TAs is decidable if the label theory is decidable.*

While normalization is always possible, an S-TA may be *exponentially* more succinct than the equivalent normalized S-TA. This is true already for the classic case, i.e., when $\mathcal{U}^\sigma = \{()\}$.

Proposition 3.9. *The non-emptiness problem of alternating S-TAs without attributes is* EXPTIME-*complete.*

Proof. We prove this results using the fact that non-emptiness of alternating tree automata over finite alphabets is EXPTIME [CDG+07, Theorem 7.5.1]. An alternating tree automaton is a tuple $\mathcal{A} = (Q, \Sigma, \varphi, \Delta)$ such that Q is a finite set of states, Σ is ranked alphabet, φ is a propositional formula with variables in Q (e.g., $q_1 \vee \neg q_2$), and Δ is a transition function that maps pairs $(q, f) \in Q \times \Sigma$ of states and symbols to propositional formula with variables in $Q \times \{1, \dots, R(f)\}$ (e.g., $(q_1, 2) \wedge (q_2, 1)$). In particular, for each symbol f of arity 0 and for every state q, $\Delta(q, f)$ is either true or false. A tree t is accepted by $\mathcal{A}$ at state $q \in Q$, $t \in L_q$, if

$t = f(t_1, \dots, t_n)$ and $\Delta(q, f)[(p, i)/(t_i \in L_p)]$, where $[(p, i)/(t_i \in L_p)]$ denotes the substitution of each pair (p, i) in the propositional formula $\Delta(q, f)$ with the Boolean value $t_i \in L_p$.

For inclusion in EXPTIME, consider an S-TA $A = (Q, \mathcal{T}_\Sigma, \delta)$ and $q \in Q$. Here $\mathcal{U}^\sigma = \{()\}$, i.e. there are no attributes. Construct an alternating tree automaton $\mathcal{A} = (Q, \Sigma, \{q\}, \Delta)$ over Σ with state set Q, initial state q, and mapping Δ such that for $(q, f) \in Q \times \Sigma$,

$$\Delta(q, f) \overset{\text{def}}{=} \bigvee_{(q, f, \varphi, \bar{\ell}) \in \delta(q)} \bigwedge_{i=1}^{R(f)} \bigwedge_{p \in \ell_i} (p, i).$$

Then $L(\mathcal{A})$ is non-empty iff $\mathcal{U}^{L^q_A}$ is non-empty. For inclusion in EXPTIME we use Theorem 7.5.1 from the TATA book [CDG+07].

For EXPTIME-hardness a converse reduction is not as simple because alternating tree automata allow general (positive) Boolean combinations of $Q \times \Sigma$ in the mapping Δ. Instead, let $A_i = (Q_i, \mathcal{T}_\Sigma, \delta_i)$ be top-down tree automata with initial states $q_i \in Q_i$ for $1 \leq i \leq n$ [CDG+07]. Consider all these automata as S-TAs without attributes and with pairwise disjoint Q_i. In particular, all A_i are normalized. Expand Σ to $\Sigma' = \Sigma \cup \{f\}$ where f is a fresh symbol of rank 1. Let A be the S-TA $(\{q\} \cup \bigcup_i Q_i, \mathcal{T}_{\Sigma'}, \bigcup_i \delta_i \cup \{(q, f, \lambda x.true, (\{q_i\}_{1 \leq i \leq n}))\})$ where q is a new state. It follows from the definitions that $\mathcal{U}^{L^q_A} \neq \emptyset$ iff $\bigcap_i \mathcal{U}^{L^{q_i}_{A_i}} \neq \emptyset$. EXPTIME-hardness follows now from the intersection non-emptiness problem of tree automata [FSVY91] (already restricted to the top-down-deterministic case [Sei94]). $\square$

3.3.3 Symbolic tree transducers with regular look-ahead

Symbolic tree transducers (S-TTs) augment S-TAs with outputs. Symbolic tree transducers with regular look-ahead further augment S-TTs by allowing rules to be guarded by symbolic tree automata. Intuitively, a rule is applied to a node if and only if its children are accepted by some symbolic tree automata. We first define terms that are used below as output components of transformation rules. We assume that we have a given tree type $\mathcal{T}^{\sigma}_{\Sigma}$ for both the input trees as well as the output trees. In the case that the input tree type and the output tree type are intended to be different, we assume that $\mathcal{T}^{\sigma}_{\Sigma}$ is a combined tree type that covers both. This assumption avoids a lot of cumbersome overhead of type annotations and can be made without loss of generality. The guards and the look-aheads can be used to restrict the types as needed.

The set of *extended tree terms* is the set of tree terms of type $\mathcal{T}^{\sigma}_{\Sigma\cup\{State\}}$ where $State \notin \Sigma$ is a new fixed symbol of rank 1. A term $State[q](t)$ is always used with a *concrete value* q and $State[q]$ is also written as $\widetilde{q}$.

Definition 3.10. Given a tree type $\mathcal{T}^{\sigma}_{\Sigma}$, a finite set Q of states, and $k \geq 0$, the set $\Lambda(\mathcal{T}^{\sigma}_{\Sigma}, Q, k)$ is defined as the least set S of λ-terms called *k-rank tree transformers* that satisfies the following conditions. Let $\bar{y}$ be a k-tuple of variables of type $\mathcal{T}^{\sigma}_{\Sigma\cup\{State\}}$ and let x be a variable of type σ:

- for all $q \in Q$, and all i, $1 \leq i \leq k$, $\lambda(x,\bar{y}).\widetilde{q}(y_i) \in S$;
- for all $f \in \Sigma$, all $e : \sigma \to \sigma$ and, all $t_1, \ldots, t_{R(f)} \in S$, $\lambda(x,\bar{y}).f[e(x)](t_1(x,\bar{y}), \ldots, t_{R(f)}(x,\bar{y})) \in S$.

Definition 3.11. A *Symbolic Tree Transducer with Regular look-ahead (S-TTR) T* is a tuple $(Q, q^0, \mathcal{T}^{\sigma}_{\Sigma}, \Delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $\mathcal{T}^{\sigma}_{\Sigma}$ is the *tree type*, $\Delta \subseteq \bigcup_{k\geq 0}(Q \times \Sigma(k) \times \Psi_{\sigma} \times (2^Q)^k \times \Lambda(\mathcal{T}^{\sigma}_{\Sigma}, Q, k))$ is a finite set of *rules* $(q, f, \varphi, \bar{\ell}, t)$, where t is the *output*. For $k = 0$ we assume that $(2^Q)^k = \{()\}$, i.e., a rule for $c \in \Sigma(0)$ has the form $(q, c, \varphi, (), \lambda x.t(x))$ where $t(x)$ is a tree term.

A rule is *linear* if its output is $\lambda(x,\bar{y}).u$ where each y_i occurs at most once in u. T is linear when all rules of T are linear.

A rule $(q, f, \varphi, \bar{\ell}, t)$ is also denoted by $q \xrightarrow{f,\varphi,\bar{\ell}} t$. The *open view* of a rule $q \xrightarrow{f,\varphi,\bar{\ell}} t$ is denoted by $\widetilde{q}(f[x](\bar{y})) \xrightarrow{\varphi(x),\bar{\ell}} t(x,\bar{y})$. The open view treats the state as a function symbol that takes a tree as input; this view is similar to the syntax of FAST and is technically more convenient for *term rewriting*. The look-ahead, when omitted, is $\bar{\emptyset}$ by default. Figure 3.6 illustrates an open view of a linear rule over the tree type $\mathcal{T}^{Int}_{\Sigma_1}$ over $\Sigma_1 = \{f, g, h\}$, where $R(f) = 2$, $R(g) = 3$, and $R(h) = 0$.

Let T be an S-TTR $(Q, q^0, \mathcal{T}^{\sigma}_{\Sigma}, \Delta)$. The following construction is used to extract an S-TA from T that accepts all input trees for which T is defined. Let t be a k-rank tree

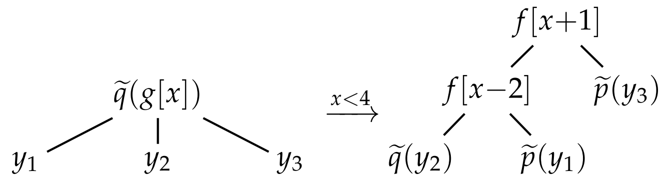


FIGURE 3.6: Depiction of an S-TT linear rule of rank 3.

transformer. For $1 \leq i \leq k$ let $St(i,t)$ denote the set of all states q such that $\widetilde{q}(y_i)$ occurs in t.

Definition 3.12. The *domain automaton of T*, $\mathbf{d}(T)$, is the S-TA $(Q, \mathcal{T}_\Sigma^\sigma, \{(q, f, \varphi, (\ell_i \cup St(i,t))_{i=1}^{R(f)}) \mid q \xrightarrow{f,\varphi,\bar{\ell}} t \in \Delta\})$.

The rules of the domain automaton also take into account the states that occur in the outputs in addition to the look-ahead states. For example, the rule in Figure 3.6 yields the domain automaton rule $(q, g, \lambda x.x < 4, (\{p\},\{q\},\{p\}))$.

We recall that given a lambda term $u = \lambda(x,\bar{y}).v$, the term $u(a,\bar{s})$ is the function application of u to $(a,\bar{s})$, where a and $\bar{s}$ substitute x and $\bar{y}$ respectively. In the following let T be the S-TTR, and for $\ell \subseteq Q$, let $\mathbf{L}_T^\ell \stackrel{\text{def}}{=} \mathbf{L}_{\mathbf{d}(T)}^\ell$.

Definition 3.13. For all $q \in Q_T$, the *transduction of T at q* is the function $\mathbf{T}_T^q$ from $\mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$ to $2^{\mathcal{U}^{\mathcal{T}_\Sigma^\sigma}}$ such that, for all $t = f[a](\bar{s}) \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$,

$$\begin{aligned}
\mathbf{T}_T^q(t) &\stackrel{\text{def}}{=} \Downarrow_T(\widetilde{q}(t)),\\
\Downarrow_T(\widetilde{q}(t)) &\stackrel{\text{def}}{=} \bigcup\{\Downarrow_T(u(a,\bar{s})) \mid (q,f,\varphi,\bar{\ell},u) \in \Delta_T, a \in [\![\varphi]\!], \bigwedge_{i=1}^{R(f)} s_i \in \mathbf{L}_T^{\ell_i}\},\\
\Downarrow_T(t) &\stackrel{\text{def}}{=} \{f[a](\bar{v}) \mid \bigwedge_{i=1}^{R(f)} v_i \in \Downarrow_T(s_i)\}.
\end{aligned}$$

The *transduction of T* is $\mathscr{T}_T \stackrel{\text{def}}{=} \mathbf{T}_T^{q^0}$. The definitions are lifted to sets using set union.

We omit T from $\mathbf{T}_T^q$ and $\Downarrow_T$ when T is clear from the context.

Example 3.14. *Recall the transformation remScript in Figure 3.1. These are the corresponding rules. We use q for the state of remScript, and ı for a state that outputs the identity transformation. The "safe" case is*

$$\widetilde{q}(node[x](y_1,y_2,y_3)) \xrightarrow{x \neq \texttt{"script"}} node[x](\widetilde{\imath}(y_1), \widetilde{q}(y_2), \widetilde{q}(y_3)),$$

the "unsafe" case is $\widetilde{q}(node[x](y_1,y_2,y_3)) \xrightarrow{x = \texttt{"script"}} \widetilde{\imath}(y_3)$, *and the "harmless" case is* $\widetilde{q}(nil[x]()) \xrightarrow{true} nil[x]()$.

```
type BT[x : Int]{L(0), N(2)}
lang oddRoot : BT {
        N(t1, t2) where (odd x)
    |   L() where (odd x)
}
def evenRoot : BT := (complement oddRoot)
trans h : BT → BT {
        N(t1, t2) given (oddRoot t1) to (N [−x] (h t1) (h t2))
    |   N(t1, t2) given (evenRoot t1) to (N [x] (h t1) (h t2))
    |   L() to (L [x])
}
```

FIGURE 3.7: Conditional flip.

In FAST, a transformation $\mathbf{T}^q$ is defined by the statement

$$\textbf{trans}\ q : \tau\ \rightarrow\ \tau\ \{\underbrace{f(\bar{y})\ \textbf{where}\ \varphi(x)\ \textbf{given}\ \ell(\bar{y})\ \textbf{to}\ t(x,\bar{y})}_{\text{a rule with source state } q \text{ and input } f[x](\bar{y})} \mid \ldots\}$$

where $\ell(\bar{y})$ denotes the look-ahead $(\{r \mid (r\ y_i) \in \ell(\bar{y})\})_{i=1}^{R(f)}$. The semantics of a FAST transformation is given by the induced S-TTR.

Example 3.15. *The S-TTR* h *in Figure 3.8 negates a node value when the value in its left child is odd, leaves it unchanged otherwise, and is then invoked recursively on the children.*

The following property of S-TTRs will be used in Section 3.4.

Definition 3.16. T is *single-valued* if $\forall(t \in \mathcal{U}^{\mathcal{T}^{\sigma}_{\Sigma}}, q \in Q_T) : |\mathbf{T}^q_T(t)| \leq 1$.

Determinism, as defined next, implies single-valuedness, and determinism is easy to decide. Intuitively, determinism means that there are no two distinct transformation rules that are enabled for the same node of any input tree. Although single-valuedness can be decided in the classic case [Ési80], decidability of single-valuedness of S-TTRs is an open problem.

Definition 3.17. T is *deterministic* when, for all $q \in Q$, $f \in \Sigma$, and rules $q \xrightarrow{f,\varphi,\bar{\ell}} t$ and $q \xrightarrow{f,\psi,\bar{r}} u$ in Δ_T, if $[\![\varphi]\!] \cap [\![\psi]\!] \neq \emptyset$ and, for all $i \in \{1,\ldots,R(f)\}$, $\mathbf{L}^{\ell_i} \cap \mathbf{L}^{r_i} \neq \emptyset$, then $t = u$.

3.3.4 The role of regular look-ahead

In this section we briefly describe what motivated our choice of considering S-TTRs in place of S-TTs. The main drawback of S-TTs is that they are not closed under composition, even for very restricted classes. As shown in the next example, when S-TTs are allowed to *delete* subtrees, the *domain* is not preserved by composition.

```
type $BBT$ $[b : \mathbb{B}]\{L(0), N(2)\}$
trans $s_1 : BBT \rightarrow BBT$ {
        $L()$ where $b$ to $(L[b])$
    |   $N(x,y)$ where $b$ to $(N\ [b]\ (s_1\ x)\ (s_1\ y))$
}
trans $s_2 : BBT \rightarrow BBT$ {
        $L()$ to $(L\ [true])$
    |   $N(x,y)$ to $(L\ [true])$
}
```

FIGURE 3.8: S-TTs for which the composition is naturally expressible by an S-TTR

Example 3.18. *Consider the* FAST *program in Figure 3.7.*

Given an input t, s_1 outputs the same tree t iff all the nodes in t have attribute true. Given an input t, s_2 always outputs $L[true]$. Both transductions are definable using S-TTs since they do not use look-ahead. Now consider the composed transduction $s = s_1 \circ s_2$ that outputs $L[true]$ iff all the nodes in t have attribute value true. This function cannot be computed by an S-TT: when reading a node $N[b](x,y)$, if the S-TT does not produce any output, it can only continue reading one of the two subtrees. This means that the S-TT cannot check whether the other subtree contains any node with attribute value $false$. However, s can be computed using an S-TTR that checks that both x and y contain only nodes with attribute true.

Example 3.15 shows that STTRs are sometimes more convenient to use than STTs. Although the transformation h can be expressed using a nondeterministic STT that guesses if the attribute of the left child is odd or even, using a deterministic STTR is a more natural solution.

3.3.5 Operations on automata and transducers

In FAST one can define new languages and new transformations in terms of previously defined ones. FAST also supports an assertion language for checking simple program properties such as **assert-true** (**is-empty** a).

- *Operations that compute new languages:*

  intersect A_1 A_2, **complement** A, *etc.:* operations over S-TAs [VB15];

  domain T*:* computes the domain of the S-TTR T using the operation from Definition 3.12;

  pre-image T A*:* computes an S-TA accepting all the inputs for which T produces an output belonging to A.

- *Operations that compute new transformations:*

  restrict *T A:* constructs a new S-TTR that behaves like T but is only defined on the inputs that belong to A;

  restrict-out *T A:* constructs a new S-TTR that behaves like T but is only defined on the inputs for which T produces an output that belongs to A;

  compose T_1 T_2*:* constructs a new S-TTR that computes the functional composition $T_1 \circ T_2$ of T_1 and T_2 (algorithm described in Section 3.4).

- *Assertions:*

  $a \in A$, $A_1 = A_2$, **is-empty** *A:* decision procedures for S-TAs; in the order, membership, language equivalence, and emptiness (Proposition 3.8 and [VB15]);

  type-check A_1 T A_2*:* true iff, for every input in A_1, T only produces outputs in A_2.

Finally, we show how the transducer operations we described are special applications of composition.

Proposition 3.19.

$$\begin{aligned}
\textbf{\textit{restrict}}\ T\ A &= \textbf{\textit{compose}}\ I_A\ T\\
\textbf{\textit{restrict-out}}\ T\ A &\equiv \textbf{\textit{compose}}\ T\ I_A\\
\textbf{\textit{pre-image}}\ T\ A &\equiv \textbf{\textit{domain}}\ (\textbf{\textit{restrict-out}}\ T\ A)\\
\textbf{\textit{type-check}}\ A_1\ T\ A_2 &\equiv \textbf{\textit{is-empty}}\ (\textbf{\textit{intersect}}\ A_1\ (\textbf{\textit{pre-image}}\ T\ (\textbf{\textit{complement}}\ A_2)))
\end{aligned}$$

where I is the identity S-TTR and I_A is the identity S-TTR that is defined only on the set of trees accepted by A.

3.4 Composition of S-TTRs

Closure under composition is a fundamental property for transducers. Composition is needed as a building block for many operations, such as pre-image computation and output restriction. Unfortunately, as shown in Example 3.18 and by Fülöp and Vogler [FV14], S-TTs are not closed under composition. Particularly, when tree rules may *delete* and/or *duplicate* input subtrees, the composition of two S-TT transductions might not be expressible as an S-TT transduction. This is already known for top-down tree transducers and can be avoided either by considering restricted fragments, or by instead adding regular look-ahead [Eng75, Bak79, Eng80]. Here, we consider the latter option. Intuitively, regular look-ahead acts as an additional child-guard that is carried

```
trans f : BT → BT {
        L() to (L [i])
      | L() to (L [5])
      | N (x,y) to (N [i] (f x) (f y))
  }
trans g : BT → BT {
        L() to (N [0] (L [i]) (L [i]))
      | N(x,y) to (N [0] (N [i] x y) (N [i] x y))
  }
```

FIGURE 3.9: Non-composable S-TTR-definable transformations.

over in the composition so that even when a subtree is deleted, the child-guard remains in the composed transducer and is not "forgotten". While deletion can be handled by S-TTRs, duplication is a much more difficult feature to support. When duplication is combined with nondeterminism, as shown in the next example, it is still not possible to compose S-TTRs. In practice this case is unusual, and it can only appear when programs produce more than one output for a given input.

Example 3.20. *Let f be the function that, given a tree of type* `BT` *(see Example 3.4) transforms it by nondeterministically replacing some leaves with the value 5. Let g be the function that given any tree t always outputs $N[0](t,t)$. The* FAST *programs corresponding to f and g are shown in Figure 3.9.*

The composed function $g(f(L[1]))$ produces the trees $N[0](L[1], L[1])$ and $N[0](L[5], L[5])$, where the two leaves contain the same value since they are "synchronized" on the same run. The function $f \circ g$ cannot be expressed by an S-TTR.

3.4.1 Composition algorithm

Algorithms for composing transducers with regular look-ahead have been studied extensively [FV88]. However, as shown by Fülöp and Vogler [FV14], extending existing transducers results to the symbolic setting is a far from trivial task. The key property that makes symbolic transducers semantically different and much more challenging than classic tree transducers, apart from the complexity of the label theory itself, is the *output computation*. In symbolic transducers the output attributes depend *symbolically* on the input attribute. Effectively, this breaks the application of some well-established techniques that no longer carry over to the symbolic setting. For example, while for top-down tree transducers the output language is always regular, this is not the case for symbolic tree transducers. This anomaly is caused by the fact that the input attribute can appear more than once in the output of a rule.

Let S and T be two S-TTRs with disjoint sets of states Q_S and Q_T respectively. We want to construct a composed S-TTR $S \circ T$ such that $\mathbf{T}_{S \circ T} = \mathbf{T}_S \circ \mathbf{T}_T$. The composition $\mathbf{T}_S \circ \mathbf{T}_T$ is defined as $(\mathbf{T}_S \circ \mathbf{T}_T)(x) = \bigcup_{y \in \mathbf{T}_S(x)} \mathbf{T}_T(y)$, following the convention Fülöp and Vogler [FV98].

For $p \in Q_S$ and $q \in Q_T$, assume that '.' is an injective pairing function that constructs a new pair state $p.q \notin Q_S \cup Q_T$. In a nutshell, we use a least fixed point construction starting with the initial state $q^0_S.q^0_T$. Given a reached (unexplored) pair state $p.q$ and symbol $f \in \Sigma$, the rules from $p.q$ and f are constructed by considering all possible constrained rewrite reductions of the form

$$(true, (\emptyset)_{i=1}^{R(f)}, \widetilde{q}(\widetilde{p}(f[x](\bar{y})))) \xrightarrow[S]{} (\_, \_, \widetilde{q}(\_)) \xrightarrow[T]{*} (\varphi, \bar{\ell}, t)$$

where t is irreducible. There are finitely many such reductions. Each such reduction is done modulo attribute and look-ahead constraints and returns a rule $p.q \xrightarrow{f, \varphi, \bar{\ell}} t$.

Example 3.21. *Suppose* $\widetilde{p}(f[x](y_1, y_2)) \xrightarrow[S]{x>0} \widetilde{p}(y_2)$. *Assume also that* $q \in Q_T$ *and that* $p.q$ *has been reached. Then*

$$(true, \bar{\emptyset}, \widetilde{q}(\widetilde{p}(f[x](y_1, y_2)))) \xrightarrow[S]{} (x>0, \emptyset, \widetilde{q}(\widetilde{p}(y_2)))$$

where $\widetilde{q}(\widetilde{p}(y_2))$ *is irreducible. The resulting rule (in open form) is* $\widetilde{p.q}(f[x](y_1, y_2)) \xrightarrow{x>0} \widetilde{p.q}(y_2)$.

The rewriting steps are done modulo attribute constraints. To this end, a *k-configuration* is a triple (γ, L, u) where γ is a formula with $FV(\gamma) \subseteq \{x\}$, L is a k-tuple of sets of pair states $p.q$ where $p \in Q_S$ and $q \in Q_T$, and u is an extended tree term. We use configurations to describe reductions of T. Formally, given two S-TTRs $S = (Q_S, q^0_S, \mathcal{T}^\sigma_\Sigma, \Delta_S)$ and $T = (Q_T, q^0_T, \mathcal{T}^\sigma_\Sigma, \Delta_T)$, the composition of S and T is defined as follows

$$\begin{aligned} S \circ T \;\stackrel{\text{def}}{=}\; & (Q_S \cup \{p.q \mid p \in Q_S, q \in Q_T\}, q^0_S.q^0_T, \mathcal{T}^\sigma_\Sigma, \\ & \Delta_S \cup \bigcup_{p \in Q_S, q \in Q_T, f \in \Sigma} \mathbf{Compose}(p, q, f)). \end{aligned}$$

For $p \in Q_S$, $q \in Q_T$ and $f \in \Sigma$, the procedure for creating all composed rules from $p.q$ and symbol f is as follows.

$\mathbf{Compose}(p, q, f) \stackrel{\text{def}}{=}$

1. ***choose*** $(p, f, \varphi, \bar{\ell}, u)$ ***from*** Δ_S;

2. ***choose*** $(\psi, \bar{P}, t)$ ***from*** $\mathbf{Reduce}(\varphi, (\emptyset)_{i=1}^{R(f)}, \widetilde{q}(u))$;

3. ***return*** $(p.q, f, \psi, \bar{\ell} \uplus \bar{P}, t)$.

The procedure **Reduce** uses a procedure $\textbf{Look}(\varphi, L, q, t)$ that, given an attribute formula φ with $FV(\varphi) \subseteq \{x\}$, a composed look-ahead L of rank k, a state $q \in Q_T$, and an extended tree term t including states from Q_S, returns all possible extended contexts and look-aheads (i.e. those containing pair states). Assume, without loss of generality, that $\mathbf{d}(T)$ is normalized. We define a function *sin*, such that $sin(\{e\}) \stackrel{\text{def}}{=} e$ for any singleton set $\{e\}$, and undefined otherwise. This function extracts the only element in a singleton set. Notice that since we operate over normalized transducers, *sin* is always defined.

$\textbf{Look}(\varphi, L, q, t) \stackrel{\text{def}}{=}$

```
1. if $t = \widetilde{p}(y_i)$ where $p \in Q_S$ then return $(\varphi, L \uplus_i \{p.q\})$;
2. if $t = g[u_0](\bar{u})$ where $g \in \Sigma$ then
   (a) choose $(q, g, \psi, \bar{\ell})$ from $\delta_{\mathbf{d}(T)}$ where $IsSat(\varphi \wedge \psi(u_0))$;
   (b) $L_0 := L$, $\varphi_0 := \varphi \wedge \psi(u_0)$;
   (c) for $(i = 1; i \leq R(g); i{+}{+})$
           choose $(\varphi_i, L_i)$ from Look$(\varphi_{i-1}, L_{i-1}, sin(\ell_i), u_i)$;
   (d) return $(\varphi_{R(g)}, L_{R(g)})$.
```

The function $\textbf{Look}(\varphi, L, q, t)$ returns a finite (possibly empty) set of pairs because there are only finitely many choices in 2(a), and in 2(c) the term u_i is strictly smaller than t. Moreover, the satisfiability check in 2(a) ensures that $\varphi_{R(g)}$ is satisfiable. The combined conditions allow cross-level dependencies between attributes, which are not expressible using classic tree transducers.

Example 3.22. *Consider the instance* $\textbf{Look}(x{>}0, \bar{\emptyset}, q, t)$ *for* $t = g[x{+}1](g[x{-}2](\widetilde{p_1}(y_2)))$ *where* $g \in \Sigma(1)$. *Suppose there is a rule* $(q, g, \lambda x.odd(x), \{q\}) \in \delta_{\mathbf{d}(T)}$ *that requires that all attributes of g are odd and assume that there is no other rule for g from q. The term t itself may arise as an output of a rule* $\widetilde{p}(f[x](y_1, y_2)) \to g[x{+}1](g[x{-}2](\widetilde{p_1}(y_2)))$ *of S. Clearly, this means that t is not a valid input of T at q because of the cross-level dependency between attributes due to x, implying that both attributes cannot be odd at the same time. Let us examine how this is handled by the* ***Look*** *procedure.*

In $\textbf{Look}(x{>}0, \bar{\emptyset}, q, t)$ *line 2(c) we have the recursive call* $\textbf{Look}(x{>}0 \wedge odd(x{+}1), \bar{\emptyset}, q, g[x{-}2](\widetilde{p_1}(y_2)))$. *Inside the recursive call we have the failing satisfiability check of* $IsSat(x{>}0 \wedge odd(x{+}1) \wedge odd(x{-}2))$ *in line 2(a). So that there exists no choice for which 2(d) is reached in the original call so the set of return values of* $\textbf{Look}(x{>}0, \bar{\emptyset}, q, t)$ *is empty.*

In the following we pretend, without loss of generality, that for each rule $\tau = (q, f, \varphi, \bar{\ell}, t)$ there is a state q_τ that uniquely identifies the rule $(q_\tau, f, \varphi, \bar{\ell}, t)$; q_τ is used to refer to the guard and the look-ahead of τ chosen in line 2(a) in the call to **Look** in 2(b) below, q_τ is not used elsewhere.

Reduce$(\gamma, L, v) \stackrel{\text{def}}{=}$

```
1. if $v = \widetilde{q}(\widetilde{p}(y_i))$ where $q \in Q_T$ and $p \in Q_S$ then return $(\gamma, L, \widetilde{p.q}(y_i))$
2. if $v = \widetilde{q}(g[u_0](\bar{u}))$ where $q \in Q_T$ and $g \in \Sigma$ then
    (a) choose $\tau = (q, g, \_, \_, t)$ from $\Delta_T$;
    (b) choose $(\gamma_1, L_1)$ from Look$(\gamma, L, q_\tau, g[u_0](\bar{u}))$;
    (c) choose $\chi$ from Reduce$(\gamma_1, L_1, t(u_0, \bar{u}))$ return $\chi$;
3. if $v = g[t_0](\bar{t})$ where $g \in \Sigma$ then
    (a) $\gamma_0 := \gamma$, $L_0 := L$;
    (b) for $(i = 1; i \le R(g); i{+}{+})$
            choose $(\gamma_i, L_i, u_i)$ from Reduce$(\gamma_{i-1}, L_{i-1}, t_i)$;
    (c) return $(\gamma_{R(g)}, L_{R(g)}, g[t_0](\bar{u}))$.
```

There is a close relationship between **Reduce** and Definition 3.13. We include the case

$$\mathbf{T}_T^q(\widetilde{p}(t)) \stackrel{\text{def}}{=} \mathbf{T}_T^q(\mathbf{T}_S^p(t)) \quad \text{for } p \in Q_S \text{ and } t \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}, \tag{3.2}$$

that allows states of S to occur in the input trees to $\mathbf{T}_T^q$ in a non-nested manner. Intuitively this means that rewrite steps of T are carried out first while rewrite steps of S are being postponed (called by name).

3.4.2 Proof of correctness

We start by proving a lemma that justifies the extension 3.2..

Lemma 3.23. *For all $t \in \Lambda(\mathcal{T}_\Sigma^\sigma, Q_S, k)$, $\mathtt{a} \in \mathcal{U}^\sigma$, and $\mathtt{u}_i \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$:*

1. $\mathbf{T}_T^q(\Downarrow_S(t(\mathtt{a}, \bar{\mathtt{u}}))) \subseteq \mathbf{T}_T^q(t(\mathtt{a}, \bar{\mathtt{u}}))$, *and*
2. $\mathbf{T}_T^q(\Downarrow_S(t(\mathtt{a}, \bar{\mathtt{u}}))) = \mathbf{T}_T^q(t(\mathtt{a}, \bar{\mathtt{u}}))$ *when S is single-valued or T is linear.*

Proof. We prove statements 1 and 2 by induction over t. The base case is $t = \lambda(x,\bar{y}).\widetilde{p}(y_i)$ for some $p \in Q_S$ and some i, $1 \le i \le k$. We have

$$\mathscr{T}_T^q(\Downarrow_S(\widetilde{p}(\mathtt{u_i}))) = \mathscr{T}_T^q(\mathscr{T}_S^p(\mathtt{u_i})) = \mathscr{T}_T^q(\widetilde{p}(\mathtt{u_i}))$$

where the last equality holds by using equation (3.2). The induction case is as follows. Let $t = \lambda(x,\bar{y}).f[t_0(x)](t_i(x,\bar{y})_{i=1}^{R(f)})$. Suppose $R(f) = 1$, the proof of the general case is analogous.

$$\begin{aligned}
&\mathscr{T}_T^q(\Downarrow_S(f[t_0(\mathtt{a})](t_1(\mathtt{a},\bar{\mathtt{u}}))))\\
\overset{\text{Def}\Downarrow_S}{=}\;& \mathscr{T}_T^q\{f[t_0(\mathtt{a})](\mathtt{v}) \mid \mathtt{v} \in \Downarrow_S(t_1(\mathtt{a},\bar{\mathtt{u}}))\}\\
\overset{\text{Def}\,\mathscr{T}_T^q}{=}\;& \{w(t_0(\mathtt{a}),(\mathtt{w}_i)_{i=1}^m) \mid (\exists\,\varphi,\bar{\ell},\bar{q})\; t_0(\mathtt{a}) \in [\![\varphi]\!] \wedge\\
& q \xrightarrow{f,\varphi,\bar{\ell}} \lambda(x,y).w(x,(\widetilde{q}_i(y))_{i=1}^m) \in \Delta_T (\exists\,\mathtt{v})\,\mathtt{v} \in \Downarrow_S(t_1(\mathtt{a},\bar{\mathtt{u}})) \wedge \bigwedge_{i=1}^m \mathtt{w}_i \in \mathscr{T}_T^{q_i}(\mathtt{v})\}\\
\overset{(\star)}{\subseteq}\;& \{w(t_0(\mathtt{a}),(\mathtt{w}_i)_{i=1}^m)) \mid (\exists\,\varphi,\bar{\ell},\bar{q})\; t_0(\mathtt{a}) \in [\![\varphi]\!] \wedge\\
& q \xrightarrow{f,\varphi,\bar{\ell}} \lambda(x,y).w(x,(\widetilde{q}_i(y))_{i=1}^m) \in \Delta_T \wedge \bigwedge_{i=1}^m \mathtt{w}_i \in \mathscr{T}_T^{q_i}(\Downarrow_S(t_1(\mathtt{a},\bar{\mathtt{u}})))\}\\
\overset{\text{IH}}{\subseteq}\;& \{w(t_0(\mathtt{a}),(\mathtt{w}_i)_{i=1}^m)) \mid (\exists\,\varphi,\bar{\ell},\bar{q})\; t_0(\mathtt{a}) \in [\![\varphi]\!] \wedge\\
& q \xrightarrow{f,\varphi,\bar{\ell}} \lambda(x,y).w(x,(\widetilde{q}_i(y))_{i=1}^m) \in \Delta_T \wedge \bigwedge_{i=1}^m \mathtt{w}_i \in \mathscr{T}_T^{q_i}(t_1(\mathtt{a},\bar{\mathtt{u}}))\}\\
\overset{\text{Def}\,\mathscr{T}_T^q}{=}\;& \mathscr{T}_T^q(f[t_0(\mathtt{a})](t_1(\mathtt{a},\bar{\mathtt{u}}))).
\end{aligned}$$

The step $(\star)$ becomes '=' when either $|\Downarrow_S(t_1(\mathtt{a},\bar{\mathtt{u}}))| \le 1$ or when $m \le 1$. The first case holds if S is single-valued. The second case holds if T is linear in which case also the induction step becomes '='. Both statements of the lemma follow by using the induction principle. □

Example 3.24. *The example shows a case when*

$$\mathbf{T}_T^q(\Downarrow_S(t(\mathtt{a},\bar{\mathtt{u}}))) \neq \mathbf{T}_T^q(t(\mathtt{a},\bar{\mathtt{u}})).$$

Suppose $p \xrightarrow[S]{c,\top} \blacktriangle$, $p \xrightarrow[S]{c,\top} \triangle$, *and* $q \xrightarrow[T]{g,\top} \lambda xy.f[x](\widetilde{q}(y),\widetilde{q}(y))$. *Let* $\mathtt{f} = f[0]$, $\mathtt{c} = c[0]$, $\mathtt{g} = g[0]$. *Then*

$$\begin{aligned}
\widetilde{q}(\mathtt{g}(\widetilde{p}(\mathtt{c}))) &\underset{T}{\rightarrow} \mathtt{f}(\widetilde{q}(\widetilde{p}(\mathtt{c})),\widetilde{q}(\widetilde{p}(\mathtt{c})))\\
&\overset{*}{\underset{S}{\rightarrow}} \{\mathtt{f}(\widetilde{q}(\blacktriangle),\widetilde{q}(\blacktriangle)),\mathtt{f}(\widetilde{q}(\triangle),\widetilde{q}(\triangle))\} \cup \{\mathtt{f}(\widetilde{q}(\blacktriangle),\widetilde{q}(\triangle)),\mathtt{f}(\widetilde{q}(\triangle),\widetilde{q}(\blacktriangle))\}
\end{aligned}$$

but

$$\begin{array}{rcl}\widetilde{q}(\mathtt{g}(\widetilde{p}(\mathtt{c}))) & \xrightarrow[S]{} & \{\widetilde{q}(\mathtt{g}(\blacktriangle)),\widetilde{q}(\mathtt{g}(\triangle))\} \\ & \overset{*}{\xrightarrow[T]{}} & \{\mathtt{f}(\widetilde{q}(\blacktriangle),\widetilde{q}(\blacktriangle)),\mathtt{f}(\widetilde{q}(\triangle),\widetilde{q}(\triangle))\}\end{array}$$

where, for example, $\mathtt{f}(\widetilde{q}(\blacktriangle),\widetilde{q}(\triangle))$ *is not possible.*

The assumptions on S and T given in Lemma 3.23 are the same as in the non-symbolic setting. However the proof of Lemma 3.23 does not directly follow from existing results because the concrete alphabet $\Sigma \times \mathcal{U}^{\sigma}$ can be infinite. Theorem 3.25 generalizes to symbolic alphabets the composition result proven in by Engelfriet [Eng77, Theorem 2.11]. Theorem 3.25 uses Lemma 3.23. It implies that, in general, $\mathscr{T}_{S\circ T}$ is an overapproximation of $\mathscr{T}_S \circ \mathscr{T}_T$ and that $\mathscr{T}_{S\circ T}$ captures $\mathscr{T}_S \circ \mathscr{T}_T$ precisely either when S behaves as a partial function or when T does not duplicate its tree arguments.

Theorem 3.25. *For all $p \in Q_S$, $q \in Q_T$ and $t \in \mathcal{U}^{T^{\sigma}_{\Sigma}}$, $\mathscr{T}^{p.q}_{S\circ T}(t) \supseteq \mathscr{T}^{q}_{T}(\mathscr{T}^{p}_{S}(t))$, and if S is single-valued or if T is linear then $\mathscr{T}^{p.q}_{S\circ T}(t) \subseteq \mathscr{T}^{q}_{T}(\mathscr{T}^{p}_{S}(t))$.*

Proof. We start by introducing auxiliary definitions and by proving additional properties that help us to formalize our arguments precisely. For $p \in Q_S$ and $q \in Q_T$, given that $\mathbf{L}^{p.q}$ is the language accepted at the pair state $p.q$, we have the following relationship that is used below

$$\begin{array}{rcl}\mathbf{L}^{p.q} & \overset{\text{def}}{=} & \{t \mid \mathscr{T}^{q}_{T}(\mathscr{T}^{p}_{S}(t)) \neq \emptyset\} \\ & = & \{t \mid \exists u(u \in \mathscr{T}^{p}_{S}(t) \wedge \mathscr{T}^{q}_{T}(u) \neq \emptyset)\} \\ & = & \{t \mid \exists u(u \in \mathscr{T}^{p}_{S}(t) \wedge u \in \mathbf{L}^{q}_{T})\} \\ & = & \{t \mid \mathscr{T}^{p}_{S}(t) \cap \mathbf{L}^{q}_{T} \neq \emptyset\}.\end{array}$$

The *symbolic (or procedural) semantics* of $\mathbf{Look}(\varphi, \bar{P}, q, t)$ is the set of all pairs returned in line 1 and line 2(d) after some nondeterministic choices made in line 2(a) and the elements of recursive calls made in line 2(c). For a set P of pair states, and for a k tuple $\bar{P}$,

$$\begin{array}{rcl}\mathbf{L}^{P} & \overset{\text{def}}{=} & \bigcap_{p.q \in P} \mathbf{L}^{p.q}, \\ \mathbf{L}^{\bar{P}} & \overset{\text{def}}{=} & \{\bar{u} \mid \bigwedge_{i=1}^{k} u_i \in \mathbf{L}^{P_i}\}.\end{array}$$

The *concrete semantics* of $\mathbf{Look}(\varphi, \bar{P}, q, t)$ is defined as follows. We assume that t implicitly stands for $\lambda(x, \bar{y}).t(x, \bar{y})$ and φ stands for $\lambda x.\varphi(x)$.

$$\begin{aligned}&[\![\mathbf{Look}(\varphi, \bar{P}, q, t)]\!] \stackrel{\text{def}}{=} \\ &\{(a, \bar{u}) | a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}}, \Downarrow_S(t(a, \bar{u})) \cap \mathbf{L}^q_T \neq \emptyset\}\end{aligned} \tag{3.3}$$

The concrete semantics of a single pair $(\varphi, \bar{P})$ is

$$[\![(\varphi, \bar{P})]\!] \stackrel{\text{def}}{=} \{(a, \bar{u}) \mid a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}}\}.$$

We now prove (3.4). It is the link between the symbolic and the concrete semantics of **Look** and Definition 3.3.

$$\bigcup\{[\![\chi]\!] | \mathbf{Look}(\varphi, \bar{P}, q, t) \textit{ \textbf{returns} } \chi\} = [\![\mathbf{Look}(\varphi, \bar{P}, q, t)]\!] \tag{3.4}$$

We prove (3.4) by induction over t. The base case is when $t = \widetilde{p}(y_i)$ for some $p \in Q_S$ and y_i for some $i \in \{1, \ldots, k\}$:

$$\begin{aligned}&\bigcup\{[\![\chi]\!] | \mathbf{Look}(\varphi, \bar{P}, q, \widetilde{p}(y_i)) \textit{ \textbf{returns} } \chi\} \\ &= [\![(\varphi, \bar{P} \uplus_i p.q)]\!] \\ &= \{(a, \bar{u}) \mid a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}}, u_i \in \mathbf{L}^{p.q}\} \\ &= \{(a, \bar{u}) \mid a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}}, \mathscr{T}^p_S(u_i) \cap \mathbf{L}^q_T \neq \emptyset\} \\ &= \{(a, \bar{u}) \mid a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}}, \Downarrow_S(\widetilde{p}(u_i)) \cap \mathbf{L}^q_T \neq \emptyset\} \\ &= [\![\mathbf{Look}(\varphi, \bar{P}, q, \widetilde{p}(u_i))]\!].\end{aligned}$$

The induction case is when $t = f[t_0](\bar{t})$. Assume $R(f) = 2$. IH is that (3.4) holds for t_1 and t_2. Assume, without loss of generality, that $\mathbf{d}(T)$ is normalized. We have for all $a \in \mathcal{U}^\sigma$ and $\bar{u} \in (\mathcal{U}^{\mathcal{T}^\sigma_\Sigma})^k$,

$$\begin{aligned}&(a, \bar{u}) \in \bigcup\{[\![\chi]\!] \mid \mathbf{Look}(\varphi, \bar{P}, q, f[t_0](\bar{t})) \textit{ \textbf{returns} } \chi\} \\ \overset{(\text{Def }\mathbf{Look})}{\Leftrightarrow}\quad & (\exists \psi, q_1, q_2)\,(q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)}, \mathit{IsSat}(\varphi \wedge \psi(t_0)), \\ & (\exists \varphi', \bar{P}', \varphi'', \bar{P}'')\,\mathbf{Look}(\varphi \wedge \psi(t_0), \bar{P}, q_1, t_1) \textit{ \textbf{returns} } (\varphi', \bar{P}'), \\ & \mathbf{Look}(\varphi', \bar{P}', q_2, t_2) \textit{ \textbf{returns} } (\varphi'', \bar{P}''), (a, \bar{u}) \in [\![(\varphi'', \bar{P}'')]\!] \\ \overset{(\text{IH})}{\Leftrightarrow}\quad & (\exists \psi, q_1, q_2)\,(q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)}, \mathit{IsSat}(\varphi \wedge \psi(t_0)), \\ & (\exists \varphi', \bar{P}')\,\mathbf{Look}(\varphi \wedge \psi(t_0), \bar{P}, q_1, t_1) \textit{ \textbf{returns} } (\varphi', \bar{P}'), \\ & (a, \bar{u}) \in [\![\mathbf{Look}(\varphi', \bar{P}', q_2, t_2)]\!] \\ \overset{(\text{Eq (3.3)})}{\Leftrightarrow}\quad & (\exists \psi, q_1, q_2)\,(q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)}, \mathit{IsSat}(\varphi \wedge \psi(t_0)),\end{aligned}$$

$$
\begin{array}{cl}
& (\exists\varphi',\bar{P}')\ \mathbf{Look}(\varphi\wedge\psi(t_0),\bar{P},q_1,t_1)\ \textbf{\textit{returns}}\ (\varphi',\bar{P}'),\\
& a\in[\![\varphi']\!],\bar{u}\in\mathbf{L}^{\bar{P}'},\Downarrow_S(t_2(a,\bar{u}))\cap\mathbf{L}_T^{q_2}\neq\emptyset\\
\overset{\text{(IH)}}{\Leftrightarrow} & (\exists\psi,q_1,q_2)\,(q,f,\psi,(\{q_1\},\{q_2\}))\in\delta_{\mathbf{d}(T)},IsSat(\varphi\wedge\psi(t_0)),\\
& (a,\bar{u})\in[\![\mathbf{Look}(\varphi\wedge\psi(t_0),\bar{P},q_1,t_1)]\!],\Downarrow_S(t_2(a,\bar{u}))\cap\mathbf{L}_T^{q_2}\neq\emptyset\\
\overset{\text{(Eq (3.3))}}{\Leftrightarrow} & (\exists\psi,q_1,q_2)\,(q,f,\psi,(\{q_1\},\{q_2\}))\in\delta_{\mathbf{d}(T)},IsSat(\varphi\wedge\psi(t_0)),\\
& a\in[\![\varphi]\!]\cap[\![\psi(t_0)]\!],\bar{u}\in\mathbf{L}^{\bar{P}},\Downarrow_S(t_1(a,\bar{u}))\cap\mathbf{L}_T^{q_1}\neq\emptyset,\Downarrow_S(t_2(a,\bar{u}))\cap\mathbf{L}_T^{q_2}\neq\emptyset\\
\overset{\text{(Def 3.3)}}{\Leftrightarrow} & a\in[\![\varphi]\!],\bar{u}\in\mathbf{L}^{\bar{P}},\Downarrow_S(f[t_0(a)](t_1(a,\bar{u}),t_2(a,\bar{u})))\cap\mathbf{L}_T^{q}\neq\emptyset\\
\Leftrightarrow & a\in[\![\varphi]\!],\bar{u}\in\mathbf{L}^{\bar{P}},\Downarrow_S(t(a,\bar{u}))\cap\mathbf{L}_T^{q}\neq\emptyset\\
\overset{\text{(Eq (3.3))}}{\Leftrightarrow} & (a,\bar{u})\in[\![\mathbf{Look}(\varphi,\bar{P},q,t)]\!].
\end{array}
$$

Equation (3.4) follows by the induction principle. Observe that, so far, no assumptions on S or T were needed.

A triple $(\varphi,\bar{P},t)$ of valid arguments of **Reduce** denotes the function $\boldsymbol{\partial}_{(\varphi,\bar{P},t)}$ such that, for all $\mathtt{a}\in\mathcal{U}^{\sigma}$ and $\mathtt{u}_i\in\mathcal{U}^{\mathcal{T}^{\sigma}_{\Sigma}}$,

$$
\boldsymbol{\partial}_{(\varphi,\bar{P},t)}(\mathtt{a},\bar{\mathtt{u}})\overset{\text{def}}{=}\begin{cases}\Downarrow_T(t(\mathtt{a},\bar{\mathtt{u}})), & \text{if } (\mathtt{a},\bar{\mathtt{u}})\in[\![(\varphi,\bar{P})]\!];\\ \emptyset, & \text{otherwise.}\end{cases}\tag{3.5}
$$

Next, we prove (3.6) under the assumption that S is single-valued or T is linear. For all $\mathtt{a}\in\mathcal{U}^{\sigma}$, $\mathtt{u}_i\in\mathcal{U}^{\mathcal{T}^{\sigma}_{\Sigma}}$ and $\mathtt{v}\in\mathcal{U}^{\mathcal{T}^{\sigma}_{\Sigma}}$,

$$
\exists\alpha,\mathtt{v}\in\boldsymbol{\partial}_{\alpha}(\mathtt{a},\bar{\mathtt{u}})\wedge\mathbf{Reduce}(\varphi,\bar{P},t)\ \textbf{\textit{returns}}\ \alpha\Leftrightarrow\mathtt{v}\in\boldsymbol{\partial}_{(\varphi,\bar{P},t)}(\mathtt{a},\bar{\mathtt{u}}).\tag{3.6}
$$

The proof is by induction over t wrt the following term order: $u\prec t$ if either u is a proper subterm of t or if the largest *State*-subterm has strictly smaller height in u than in t.

The base case is $t=\widetilde{q}(\widetilde{p}(y_i))$ where $q\in Q_T$, $p\in Q_S$, and (3.6) follows because $\mathbf{Reduce}(\varphi,\bar{P},\widetilde{q}(\widetilde{p}(y_i)))$ returns $(\varphi,\bar{P},\widetilde{p.q}(y_i))$ and $\lambda y.\widetilde{p.q}(y)$ denotes, by definition, the composition $\lambda y.\widetilde{q}(\widetilde{p}(y))$.

We use the extended case (3.7) of Definition 3.13 that allows states of S to occur in $\bar{\mathtt{t}}$. This extension is justified by Lemma 3.23. For $q\in Q_T$:

$$
\begin{aligned}\Downarrow_T(\widetilde{q}(f[\mathtt{a}](\bar{\mathtt{t}})))\overset{\text{def}}{=}\bigcup\{\Downarrow_T(u(\mathtt{a},\bar{\mathtt{t}}))\mid{}&(q,f,\varphi,\bar{\ell},u)\in\Delta_T,\mathtt{a}\in[\![\varphi]\!],\\&\bigwedge_{i=1}^{R(f)}\Downarrow_S(\mathtt{t}_i)\cap\mathbf{L}_T^{\ell_i}\neq\emptyset\}.\end{aligned}\tag{3.7}
$$

Observe that when $\mathtt{t}_i$ does not contain any states of S then $\Downarrow_S(\mathtt{t}_i) = \{\mathtt{t}_i\}$ and thus the condition $\Downarrow_S(\mathtt{t}_i) \cap \mathbf{L}_T^{\ell_i} \neq \emptyset$ simplifies to the condition $\mathtt{t}_i \in \mathbf{L}_T^{\ell_i}$ used in the original version of Definition 3.13.

There are two induction cases. The first induction case is $t = \widetilde{q}(f[t_0](\bar{t}))$ where $q \in Q_T$ and $f \in \Sigma$. Let $t' = f[t_0](\bar{t})$. For all $\mathtt{a} \in \mathcal{U}^\sigma$, $\mathtt{u}_i \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$ and $\mathtt{v} \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$,

$$
\begin{aligned}
& (\exists\alpha)\ \mathtt{v} \in \boldsymbol{\partial}_\alpha(\mathtt{a},\bar{\mathtt{u}}), \mathbf{Reduce}(\varphi,\bar{P},\widetilde{q}(t'))\ \textbf{\textit{returns}}\ \alpha \\
\overset{\text{Def }\mathbf{Reduce}}{\Leftrightarrow}\ & (\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T, (\exists\psi,\bar{R})\ \mathbf{Look}(\varphi,\bar{P},q_\tau,t')\ \textbf{\textit{returns}}\ (\psi,\bar{R}), \\
& (\exists\beta)\ \mathbf{Reduce}(\psi,\bar{R},u(t_0,\bar{t}))\ \textbf{\textit{returns}}\ \beta, \mathtt{v} \in \boldsymbol{\partial}_\beta(\mathtt{a},\bar{\mathtt{u}}) \\
\overset{\text{IH}}{\Leftrightarrow}\ & (\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T, (\exists\psi,\bar{R})\ \mathbf{Look}(\varphi,\bar{P},q_\tau,t')\ \textbf{\textit{returns}}\ (\psi,\bar{R}), \\
& \mathtt{v} \in \boldsymbol{\partial}_{(\psi,\bar{R},u(t_0,\bar{t}))}(\mathtt{a},\bar{\mathtt{u}}) \\
\overset{\text{Eq (3.5)}}{\Leftrightarrow}\ & (\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T, (\exists\psi,\bar{R})\ \mathbf{Look}(\varphi,\bar{P},q_\tau,t')\ \textbf{\textit{returns}}\ (\psi,\bar{R}), \\
& \mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}}))), (\mathtt{a},\bar{\mathtt{u}}) \in [\![(\psi,\bar{R})]\!] \\
\overset{\text{Eq (3.4)}}{\Leftrightarrow}\ & (\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T, (\mathtt{a},\bar{\mathtt{u}}) \in [\![\mathbf{Look}(\varphi,\bar{P},q_\tau,t')]\!], \\
& \mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}}))) \\
\overset{\text{Eq (3.3)}}{\Leftrightarrow}\ & (\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T, \mathtt{a} \in [\![\varphi]\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}}, \Downarrow_S(t'(\mathtt{a},\bar{\mathtt{u}})), \cap\mathbf{L}_T^{q_\tau} \neq \emptyset, \\
& \mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}}))) \\
\overset{\text{Def }q_\tau}{\Leftrightarrow}\ & \mathtt{a} \in [\![\varphi]\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}}, (\exists u,\gamma,\bar{\ell})\ q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T, t_0(\mathtt{a}) \in [\![\gamma]\!], \\
& \bigwedge_{i=1}^{R(f)} \Downarrow_S(t_i(\mathtt{a},\bar{\mathtt{u}})) \cap \mathbf{L}_T^{\ell_i} \neq \emptyset, \mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}}))) \\
\overset{\text{Eq (3.7)}}{\Leftrightarrow}\ & \mathtt{a} \in [\![\varphi]\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}},\ \mathtt{v} \in \Downarrow_T(t(\mathtt{a},\bar{\mathtt{u}})) \\
\overset{\text{Def }\partial}{\Leftrightarrow}\ & \mathtt{v} \in \boldsymbol{\partial}_{(\varphi,\bar{P},t)}(\mathtt{a},\bar{\mathtt{u}}).
\end{aligned}
$$

The second induction case is $t = f[t_0](\bar{t})$. Assume $R(f) = 2$. Generalization to arbitrary ranks is straightforward by repeating IH steps below $R(f)$ times. For all $\mathtt{a} \in \mathcal{U}^\sigma$, $\mathtt{u}_i \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$, and $\mathtt{v} \in \mathcal{U}^{\mathcal{T}_\Sigma^\sigma}$,

$$
\begin{aligned}
& (\exists\alpha)\ \mathtt{v} \in \boldsymbol{\partial}_\alpha(\mathtt{a},\bar{\mathtt{u}}), \mathbf{Reduce}(\varphi,\bar{P},f[t_0](t_1,t_2))\ \textbf{\textit{returns}}\ \alpha \\
\overset{\text{Def }\mathbf{Reduce}}{\Leftrightarrow}\ & (\exists\ \varphi',\bar{P}',v_1,\varphi'',\bar{P}'',v_2), \mathbf{Reduce}(\varphi,\bar{P},t_1)\ \textbf{\textit{returns}}\ (\varphi',\bar{P}',v_1), \\
& \mathbf{Reduce}(\varphi',\bar{P}',t_2)\ \textbf{\textit{returns}}\ (\varphi'',\bar{P}'',v_2), \mathtt{v} \in \boldsymbol{\partial}_{(\varphi'',\bar{P}'',f[t_0](v_1,v_2))}(\mathtt{a},\bar{\mathtt{u}}) \\
\overset{\text{Def }\partial}{\Leftrightarrow}\ & (\exists\ \varphi',\bar{P}',w_1,\varphi'',\bar{P}'',w_2), \mathbf{Reduce}(\varphi,\bar{P},t_1)\ \textbf{\textit{returns}}\ (\varphi',\bar{P}',w_1), \\
& \mathbf{Reduce}(\varphi',\bar{P}',t_2)\ \textbf{\textit{returns}}\ (\varphi'',\bar{P}'',w_2), \mathtt{v} \in \Downarrow_T(f[t_0(\mathtt{a})](w_1(\mathtt{a},\bar{\mathtt{u}}),w_2(\mathtt{a},\bar{\mathtt{u}}))), \\
& \mathtt{a} \in [\![\varphi'']\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}''} \\
\overset{\text{Def }\Downarrow_T}{\Leftrightarrow}\ & (\exists\ \varphi',\bar{P}',w_1,\varphi'',\bar{P}'',w_2), \mathbf{Reduce}(\varphi,\bar{P},t_1)\ \textbf{\textit{returns}}\ (\varphi',\bar{P}',w_1)
\end{aligned}
$$

$$\begin{array}{ll}
& \mathbf{Reduce}(\varphi', \bar{P}', t_2)\ \boldsymbol{returns}\ (\varphi'', \bar{P}'', w_2), \\
& (\exists\, \mathtt{v}_1, \mathtt{v}_2), \mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2), \mathtt{v}_1 \in \Downarrow_T(w_1(\mathtt{a}, \bar{\mathtt{u}})), \mathtt{v}_2 \in \Downarrow_T(w_2(\mathtt{a}, \bar{\mathtt{u}})) \\
& \mathtt{a} \in [\![\varphi'']\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}''} \\
\overset{\text{IH}}{\Leftrightarrow} & (\exists\, \varphi', \bar{P}', w_1), \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', w_1) \\
& (\exists\, \mathtt{v}_1, \mathtt{v}_2), \mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2), \mathtt{v}_1 \in \Downarrow_T(w_1(\mathtt{a}, \bar{\mathtt{u}})), \mathtt{v}_2 \in \boldsymbol{\partial}_{(\varphi', \bar{P}', t_2)}(\mathtt{a}, \bar{\mathtt{u}}) \\
\overset{\text{Def}\,\partial}{\Leftrightarrow} & (\exists\, \varphi', \bar{P}', w_1), \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', w_1) \\
& (\exists\, \mathtt{v}_1, \mathtt{v}_2), \mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2), \mathtt{v}_1 \in \Downarrow_T(w_1(\mathtt{a}, \bar{\mathtt{u}})) \\
& \mathtt{a} \in [\![\varphi']\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}'},\ \mathtt{v}_2 \in \Downarrow_T(t_2(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{\text{IH}}{\Leftrightarrow} & (\exists\, \mathtt{v}_1, \mathtt{v}_2), \mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2), \mathtt{v}_1 \in \boldsymbol{\partial}_{(\varphi, \bar{P}, t_1)}(\mathtt{a}, \bar{\mathtt{u}}), \mathtt{v}_2 \in \Downarrow_T(t_2(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{\text{Def}\,\partial}{\Leftrightarrow} & (\exists\, \mathtt{v}_1, \mathtt{v}_2), \mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2), \mathtt{a} \in [\![\varphi]\!], \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}}, \\
& \mathtt{v}_1 \in \Downarrow_T(t_1(\mathtt{a}, \bar{\mathtt{u}})), \mathtt{v}_2 \in \Downarrow_T(t_2(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{\text{Def}\Downarrow_T}{\Leftrightarrow} & \mathtt{a} \in [\![\varphi]\!], \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}}, \mathtt{v} \in \Downarrow_T(f[t_0(\mathtt{a})](t_1(\mathtt{a}, \bar{\mathtt{u}}), t_2(\mathtt{a}, \bar{\mathtt{u}}))) \\
\overset{\text{Def}\,\partial}{\Leftrightarrow} & \mathtt{v} \in \boldsymbol{\partial}_{(\varphi, \bar{P}, f[t_0](t_1, t_2))}.
\end{array}$$

Equation (3.6) follows by the induction principle.

Finally, we prove $\mathscr{T}^{p.q}_{S\circ T} = \mathscr{T}^{p}_{S} \circ \mathscr{T}^{q}_{T}$. Let $p \in Q_S$, $q \in Q_T$ and $f[\mathtt{a}](\bar{\mathtt{u}}), \mathtt{w} \in \mathcal{U}^{\mathcal{T}^{\sigma}_{\Sigma}}$ be fixed.

$$\begin{array}{ll}
& \mathtt{w} \in \mathscr{T}^{p.q}_{S\circ T}(f[\mathtt{a}](\bar{\mathtt{u}})) \\
\overset{\text{Def}\,\mathbf{Compose}}{\Leftrightarrow} & (\exists\, \varphi, \bar{\ell}, t), (p, f, \varphi, \bar{\ell}, t) \in \Delta_S, (\exists\, \alpha), \mathbf{Reduce}(\varphi, \bar{\emptyset}, \widetilde{q}(t))\ \boldsymbol{returns}\ \alpha, \\
& \mathtt{w} \in \boldsymbol{\partial}_{\alpha}(\mathtt{a}, \bar{\mathtt{u}}),\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S} \\
\overset{\text{Eq.(3.6)}}{\Leftrightarrow} & (\exists\, \varphi, \bar{\ell}, t), (p, f, \varphi, \bar{\ell}, t) \in \Delta_S, \mathtt{w} \in \boldsymbol{\partial}_{(\varphi, \bar{\emptyset}, \widetilde{q}(t))}(\mathtt{a}, \bar{\mathtt{u}}), \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S} \\
\overset{\text{Def}\,\partial}{\Leftrightarrow} & (\exists\, \varphi, \bar{\ell}, t), (p, f, \varphi, \bar{\ell}, t) \in \Delta_S, \mathtt{a} \in [\![\varphi]\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\emptyset}},\ \mathtt{w} \in \Downarrow_T(\widetilde{q}(t(\mathtt{a}, \bar{\mathtt{u}}))),\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S} \\
\overset{\text{Def}\,\mathscr{T}^{q}_{T}}{\Leftrightarrow} & (\exists\, \varphi, \bar{\ell}, t), (p, f, \varphi, \bar{\ell}, t) \in \Delta_S, \mathtt{a} \in [\![\varphi]\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S},\ \mathtt{w} \in \mathscr{T}^{q}_{T}(t(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{(\star)}{\Leftrightarrow} & (\exists\, \varphi, \bar{\ell}, t), (p, f, \varphi, \bar{\ell}, t) \in \Delta_S, \mathtt{a} \in [\![\varphi]\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S},\ \mathtt{w} \in \mathscr{T}^{q}_{T}(\Downarrow_S(t(\mathtt{a}, \bar{\mathtt{u}}))) \\
\overset{\text{Def}\Downarrow_S}{\Leftrightarrow} & \mathtt{w} \in \mathscr{T}^{q}_{T}(\Downarrow_S(\widetilde{p}(f[\mathtt{a}](\bar{\mathtt{u}})))) \\
\overset{\text{Def}\,\mathscr{T}^{p}_{S}}{\Leftrightarrow} & \mathtt{w} \in \mathscr{T}^{q}_{T}(\mathscr{T}^{p}_{S}(f[\mathtt{a}](\bar{\mathtt{u}}))).
\end{array}$$

Step ($\star$) uses Lemma 3.23.2. It holds only when S is single-valued or T is linear. Otherwise, only '$\Leftarrow$' holds. $\square$

3.5 Evaluation

FAST can be used in multiple different applications. We first consider HTML input sanitization for security. Then we show how augmented reality (AR) applications can be checked for conflicts. Next, we show how FAST can perform *deforestation* and verification for functional programs. Finally, we sketch how CSS analysis can be captured in FAST.

3.5.1 HTML sanitization

A central concern for secure web applications is untrusted user inputs. These may lead to a cross-site scripting (XSS) attack, which, in its simplest form, is echoing untrusted input verbatim back to the browser. Consider bulletin boards that want to allow partial markup such as `<b>` and `<i>` tags or HTML email messages, where the email provider wants rich email content with formatting and images but wants to prevent active content such as JavaScript from propagating through. In these cases, a technique called *sanitization* is used to allow rich markup, while removing active (executable) content. However, proper sanitization is far from trivial: unfortunately, for both of these scenarios above, there have been high-profile vulnerabilities stemming from careless sanitization of specially crafted HTML input leading to the creation of the infamous Samy worm for MySpace (`http://namb.la/popular/`) and the Yamanner worm for the Yahoo Mail system. In fact, MySpace has repeatedly failed to properly sanitize their HTML inputs, leading to the Month of MySpace Bugs initiative (`http://momby.livejournal.com/586.html`).

This caused the emergence of a range of libraries attempting to do HTML sanitization, including PHP Input Filter, HTML_Safe, kses, htmLawed, Safe HTML Checker, HTML Purifier. Among these, the last one, HTML Purifier (`http://htmlpurifier.org`) is believed to be most robust, so we choose it as a comparison point for our experiments. Note that HTML Purifier is a tree-based rewriter written in PHP, which uses the HTMLTidy library to parse the input.

We show how FAST is expressive enough to model HTML sanitizers, and we argue that writing such programs is easier with FAST than with current tools. Our version of an HTML sanitizer written in FAST and automatically translated by the FAST compiler into C# is partially described in Section 3.2. Although we can't argue for the correctness of our compilation into C#, we are able to show, using type-checking, that some types of XSS attacks are prevented. Moreover, sanitizers are much simpler to write in Fast thanks to composition. In all the libraries mentioned above HTML sanitization is implemented as a monolithic function in order to achieve reasonable performance. In the case of FAST each sanitization routine can be written as a single function and all the routines can be composed preserving the property of traversing the input HTML only once.

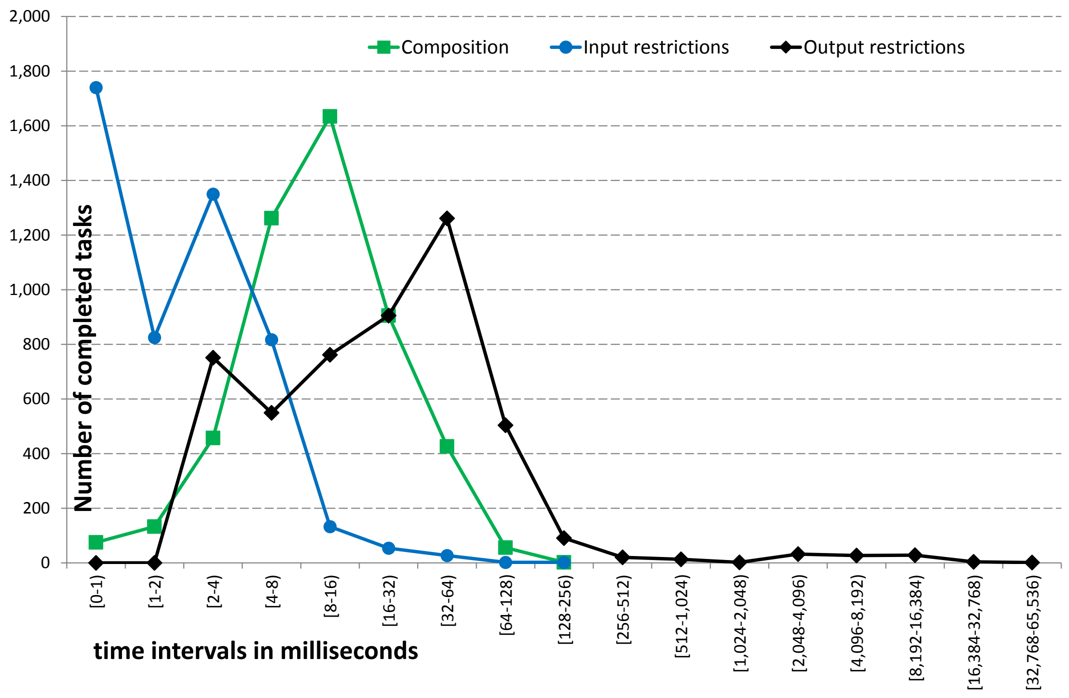


FIGURE 3.10: Augmented reality: running times for operations on transducers.

Evaluation. To compare different sanitization strategies in terms of performance, we chose 10 web sites and picked an HTML page from each of them, ranging from 20 KB (Bing) to 409 KB in size (Facebook). For speed, the FAST-based sanitizer is comparable to HTML Purify. In terms of maintainability, FAST wins on two counts. First, unlike sanitizers written in PHP, FAST programs can be analyzed statically. Second, our sanitizer is only 200 lines of FAST code instead of 10000 lines of PHP. While these are different languages, we argue that our approach is more maintainable because FAST captures the high level semantics of HTML sanitization, as well as being fewer lines of code to understand. We manually spot-checked the outputs to determine that both produce reasonable sanitizations.

3.5.2 Conflicting augmented reality applications

In *augmented reality* the view of the physical world is enriched with computer-generated information. For example, applications on the Layar AR platform provide up-to-date information such as data about crime incidents near the user's location, information about historical places and landmarks, real estate, and other points of interest.

We call a *tagger* an AR application that labels elements of a given set with a piece of information based on the properties of such elements. As an example, consider a tagger that assigns to every city a set of tags representing the monuments in such a city. A large class of shipping mobile phone AR applications are taggers, including Layar, Nokia City Lens, Nokia Job Lens, and Junaio. We assume that the physical world is represented as a list of elements, and each element is associated with a list of tags (i.e. a tree). This representation is a common one and is simple to reason about. Users should be warned if not prevented from installing applications that conflict with others they have already installed. We say that two taggers *conflict* if they both tag the same node of some input tree. In order to detect conflicts we perform the following four-step check for each pair of taggers $\langle p_1, p_2 \rangle$:

Composition: we compute p, composition of p_1 and p_2.

Input restriction: we compute p', a restriction of p that is only defined on trees where each node contains no tags.

Output restriction: we compute p'', a restriction of p' that only outputs trees in which some node contains two tags.

Check: we check if p'' is the empty transducer: if it is not the case, p_1 and p_2 conflict on every input accepted by p''.

Evaluation. Figure 3.10 shows the timing results for conflict analysis. The x-axis represents time intervals in *ms*. The y-axis shows how many cases run in a time belonging to an interval. For example about 1,600 compositions were completed between 8 and 16 ms. To collect this data, we randomly generated 100 taggers in FAST and checked whether they conflicted with each other. Each element in the input of a tagger contained a name of type string, two attributes of type real, and an attribute of type int. In our encoding the left-child of each element was the list of tags, while the right child was the next element. Each tagger we generated conforms to the following properties: 1) it is non-empty; 2) it tags on average 3 nodes; and 3) it tags each node at most once.

The sizes of our taggers varied from 1 to 95 states. The language we used for the input restriction has 3 states, the one for the output 5 states. We analyzed 4,950 possible conflicts and 222 will be actual conflicts (i.e. FAST provided an example tree on which the two taggers tagged the same node). The three plots show the time distribution for the steps of a) composition, b) input restriction, and c) output restriction respectively.

All the compositions are computed in less than 250 ms, and the average time is 15 ms. All the input restrictions are computed in less than 150 ms. The average time is 3.5 ms. All the output restrictions are computed in less than 33,000 ms. The average time is 175 ms. The output restriction takes longer to compute in some cases, due to the following two factors: 1) the input sizes are always bigger: the size of the composed transducers after the input restriction (p' in the list before) vary from 5 to 300 states and 10

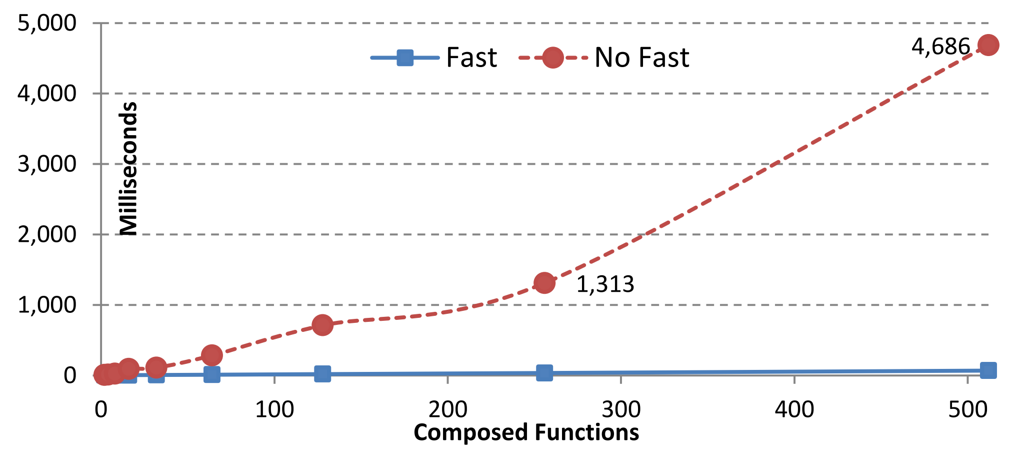


FIGURE 3.11: Deforestation advantage for a list of 4,096 integers.

to 4,000 rules. This causes the restricted output to have up to 5,000 states and 100,000 rules; and 2) since the conditions in the example are randomly generated, some of them may be complex causing the satisfiability modulo theory (SMT) solver to slow down the computation. The 33,000 ms example contains non-linear (cubic) constraints over reals. The average time of 193 ms per pairwise conflict check is quite acceptable: indeed, adding a new app to a store already containing 10,000 apps will incur an average checking overhead of about 35 minutes.

3.5.3 Deforestation

Next we explore the idea of *deforestation*. First introduced by Wadler in 1988 [Wad88], deforestation aims at eliminating intermediate computation trees when evaluating functional programs. For example, to compute the sum of the squares of the integers between 1 and n, the following small program might be used: `sum (map square (upto 1 n))`. Intermediate lists created as a result of evaluation are a source of inefficiency. However, it has been observed that transducer composition can be used to eliminate intermediate results. This can be done as long as individual functions are representable as transducers. The technique proposed by Wadler only considers transformations over finite alphabets. Here we consider transformations that use infinite alphabets and can therefore be represented only in FAST. We analyzed the performance gain obtained by deforestation in FAST on these transformations.

Evaluation. We considered the function *map_caesar* from Figure 3.12 that replaces each value x of a integer list with $(x+5)\%26$. We composed the function *map_caesar* with itself several times to see how the performance changed when using FAST. Let's call map^n the composition of *map_caesar* with itself n times. We run the experiments on

```
type IList[i : Int]{nil(0), cons(1)}
trans map_caesar : IList → IList {
        nil() to (nil[0])
    |   cons(y) to (cons [(x + 5)%26] (map_caesar y))
}
trans filter_ev : IList → IList {
        nil() to (nil[0])
    |   cons(y) where (i%2 = 0) to (cons [i] (filter_ev y))
    |   cons(y) where ¬(i%2 = 0) to (filter_ev y)
}
lang not_emp_list : IList { cons(x) }
def comp : IList → IList := (compose map_caesar filter_ev)
def comp2 : IList → IList := (compose comp comp)
def restr : IList → IList := (restrict-out comp2 not_emp_list)
assert-true (is-empty restr)
```

FIGURE 3.12: Analysis of functional programs in FAST.

lists containing randomly generated elements and we consider up to 512 composed functions. Figure 3.11 shows the running time of FAST with and without deforestation for a list of 4,096 integers used as the input. The running time of the version that uses transducer composition is almost unchanged, even for 512 compositions, while the running time of the naïvely composed functions degrades linearly in the number of composed functions. This is due to the fact that the composed version results into a single function that processes the input list in a single left-to-right pass, while the naïve composition causes the input list to be read multiple times.

3.5.4 Analysis of functional programs

FAST can also be used to perform static analysis of simple functional programs over lists and trees. Consider again the functions from Figure 3.12. As we described in the previous experiment the function *map_caesar* replaces each value x of a integer list with $(x + 5)$ *mod* 26. The function *filter_ev* removes all the odd elements from a list.

One might wonder what happens when such functions are composed. Consider the case in which we execute the map followed by the filter, followed by the map, and again by the filter. This transformation is equivalent to deleting all the elements in the list! This property can be statically checked in FAST. We first compute $comp2$ as the composition described above. As show in Figure 3.12, the language of non-empty lists can be expressed using the construct *not_emp_list*. We can then use the output restriction to restrict $comp2$ to only output non-empty lists. The final assertion shows that $comp2$ never outputs a non-empty list. In this example the whole analysis can be done in less than 10 ms.

3.5.5 CSS analysis

Cascading style-sheets (CSS) is a language for stylizing and formatting HTML documents. A CSS program is a sequence of CSS rules, where each rule contains a selector and an assignment. The selector decides which nodes are affected by the rule and the assignment is responsible for updating the selected nodes. The following is a typical CSS rule: `div p { word-spacing:30px; }`. In this case `div p` is the selector while `word-spacing:30px` is the assignment. This rule sets the attribute `word-spacing` to `30px` for every `p` node inside a `div` node. We call $C(H)$ be the updated HTML resulting from applying a CSS program C to an HTML document H. CSS programs have been analyzed using tree logic [GLQ12]. For example one can check whether given a CSS program C, there does not exist an HTML document H such that $C(H)$ contains a node n for which the attributes `color` and `background-color` both have value `black`. This property ensures that black text is never written on a black background, causing the text not to be readable. Ideally one would want to check that `color` and `background-color` never have the same value, but, since tree logic explicitly models the alphabet, the corresponding formula would be too large. By modeling CSS programs as symbolic tree transducers we can overcome this limitation. This analysis relies on the alphabet being symbolic, and we plan on extending FAST with primitives for simplifying CSS modelling.

3.6 Related work

3.6.1 Tree transducers

Tree transducers have long been studied and surveys and books are available on the topic [FV98, CDG$^+$07, Rao92]. The first models were top-down and bottom-up tree transducers [Eng75, Bak79], later extended to top-down transducers with regular look-ahead in order to achieve closure under composition [Eng77, FV88, Eng80]. Extended top-down tree transducers [MGHK09] (XTOP) were introduced in the context of program inversion and allow rules to read more than one node at a time, as long as such nodes are adjacent. When adding look-ahead, such a model is equivalent to top-down tree transducers with regular look-ahead. More complex models, such as *macro tree transducers* [EV85], have been introduced to improve expressiveness at the cost of higher complexity. Due to their high complexities we do not consider extending these models to handle symbolic alphabets.

3.6.2 Symbolic transducers

Symbolic finite transducers (S-FTs) over lists, together with a front-end language BEK, were originally introduced by Hooimeijer et al. [HLM$^+$11] with a focus on security analysis of string sanitizers. Symbolic tree transducers were originally introduced by Veanes and Bjørner [VB12b], where it was incorrectly claimed that S-TTs are closed under composition by referring to a generalization of a proof of the non-symbolic case from Fülöp and Vogler [FV98] that is only stated for total deterministic finite tree transducers. Fülöp and Vogler discovered this error and proved other properties of S-TTs [FV14] . Here we provide the first formal treatment of S-TTRs and we prove their closure under composition. We have not investigated the problem of checking equivalence of single-valued S-TTRs.

3.6.3 DSLs for tree manipulation

Domain specific languages for tree transformation have been studied in many different contexts. TTT [PS12] and Tiburon [MK08] are transducer-based languages used in natural language processing. TTT allows complex forms of pattern matching, but does not enable any form of analysis. Tiburon supports probabilistic transitions and several weighted tree transducer algorithms. Although they support weighted transitions, both the languages are limited to finite input and output alphabets. ASF+SDF [vdBHKO02] is a term-rewriting language for manipulating parsing trees. ASF+SDF is simple and efficient, but does not support any analysis. In the context of XML processing, numerous languages have been proposed for querying (XPath, XQuery [Wal07]), stream processing (STX [Bec03]), and manipulating (XSLT, XDuce [HP03]) XML trees. While being very expressive, these languages support very limited forms of analysis. Emptiness has been shown decidable for restricted fragments of XPath [BDM$^+$06]. XDuce [HP03] can be used to define basic XML transformations and supports a tree automata based type-checking that is limited to finite alphabets. We plan to extend FAST to better handle XML processing and to identify a fragment of XPath expressible in FAST. However, to the best of our knowledge, FAST is the first language for tree manipulation that supports infinite input and output alphabets while preserving decidable analysis. Table 3.1 summarizes the relations between FAST and the other domain-specific languages for tree transformations. The column σ indicates whether the language supports finite (ff) or infinite (∞) alphabets.

Language	σ	**Analysis**	**Domain**
FAST	∞	composition, typechecking, pre-image, language equivalence, determinization, complement, intersection	Tree-manipulating programs
Tiburon	ff	composition; type-checking; training; weights; language equivalence, determinization, complement, intersection	NLP
TTT	ff	-	NLP
ASF+SDF	∞	-	Parsing
XPath	∞	emptiness for a fragment	XML query (only selection)
XDuce	∞	type-checking for navigational part (finite alphabet)	XML query
XQuery, XSLT, STX	∞	-	XML transformations

TABLE 3.1: Summary of domain specific languages for manipulating trees.

3.6.4 Applications

The connection between tree transducers and deforestation was first studied by Wadler [Wad88], and then further investigated by Kühnemann et al. [KV01]. In this setting deforestation is done via *Macro Tree Transducers* (MTT) [EV85]. While being more expressive than Top Down Transducers with regular look-ahead, MTTs only support finite alphabets and their composition algorithm has very high complexity. We are not aware of an actual implementation of the techniques presented by Kühnemann et al. [KV01]. Many models of tree transducers have been introduced to analyze and execute XML transformations. Most notably, K-pebble transducers [MSV00] enjoy decidable type-checking and can capture fairly complex XSLT and XML transformations. Macro forest transducer [PS04] extend MTT to operate over unranked trees and therefore naturally capture XML transformations. Recently this model has been used to efficiently execute XQuery transformations [HMNI14]. The models we just discussed only operate over finite alphabets. Many models of automata and transducers have been applied to verifying functional programs. The equivalence problem has been shown to be decidable for some fragments of ML using Class Memory Automata [CHMO15]. This model allows values over infinite alphabets to be compared using equality, but does not support predicates arbitrary label theories. This restriction is common in the so-called data languages and makes other models operating in this setting orthogonal to symbolic automata and transducers. Higher-Order Multi-Parameter Tree Transducers (HMTT) [KTU10] are used for type-checking higher-order functional programs.

HMTTs enable sound but incomplete analysis of programs which take multiple trees as input, but only support finite alphabets. Extending our theory to multiple input trees and higher-order functions is an open research direction.

3.6.5 Open problems

Several complexity-related questions for S-TAs and S-TTRs are open and depend on the complexity of the label theory, but some lower bounds can be established using known results for finite tree automata and transducers. For example, an S-TA may be exponentially more succinct than the equivalent normalized S-TA because one can directly express the intersection non-emptiness problem of a set of normalized S-TAs as the emptiness of a single un-normalized S-TA. In the non-symbolic case, the non-emptiness problem of tree automata is P-CO, while the intersection non-emptiness problem is EXPTIME-CO [CDG$^+$07, Thm 1.7.5]. Recently, new techniques based on anti-chains have been proposed to check universality and inclusion for nondeterministic tree automata [BHH$^+$08]. Whether such techniques translate to our setting is an open research direction. Concrete open problems are decidability of: *single-valuedness of S-TTRs*, *equivalence of single-valued S-TTRs*, and *finite-valuedness of S-TTRs*. These problems are decidable when the alphabets are finite, but some proofs are quite challenging [Sei94].

Part II

Executable models for hierarchical data

Chapter 4

Symbolic visibly pushdown automata

"Like all magnificent things, it's very simple."
— Natalie Babbit, *Tuck Everlasting*

4.1 Introduction

The goal of designing streaming algorithms that can process an XML document in a single pass has resulted in streaming models for checking membership in a regular tree language and for querying [NSV04, MSV00, MGHK09]. In fact, XML documents are not the only form of hierarchical data that needs to be processed and queried efficiently: other examples are traces of structured programs, JSON files, and CSS files. In this chapter we introduce an executable single-pass model for expressing properties of hierarchical documents over complex domains.

4.1.1 Existing models

Visibly pushdown automata (VPAs) can describe languages and properties of hierarchical documents expressed as nested words. Nested words are a formalism that can model data with both linear and hierarchical structure, such as XML documents and program traces. It can be shown that VPAs are closed under Boolean operations and enjoy decidable equivalence. VPAs have been proven to be beneficial for many computational tasks, from streaming XML processing [DGN$^+$13, GN11, MZDZ13] to verification of recursive programs [CA07, DTR12]. Like many other classic models, VPAs only operate over finite alphabets, and this can be a limitation in applications such as XML processing and program trace analysis.

Symbolic Finite Automata (S-FAs) [DV14, Vea13], which we discussed in Chapter 2, are finite state automata in which the alphabet is given by a decidable Boolean algebra that may have an infinite domain, and transitions are labeled with predicates over such an algebra. S-FAs therefore accept languages of strings over a potentially infinite domain. Although strictly more expressive than finite automata, symbolic finite automata are closed under Boolean operations and admit decidable equivalence. Unfortunately S-FAs only operate over words and cannot describe properties of nested words. S-FAs are therefore not a good model to define properties over hierarchical data.

4.1.2 Contributions

We introduce Symbolic Visibly Pushdown Automata (S-VPA) as an executable model for nested words over infinite alphabets. In S-VPAs transitions are labeled with predicates over the input alphabet, analogous to symbolic finite automata that operate over strings. A key novelty of S-VPAs is the use of binary predicates to model relations between open and close tags in a nested word. Even though S-VPAs completely subsume VPAs, we show how S-VPAs still enjoy the decidable procedures and closure properties of VPAs. This result is quite surprising given that previous attempts to add binary predicates to symbolic automata caused equivalence to become undecidable and closure properties to stop holding (Section 2.4).

We finally investigate potential applications of S-VPAs in the context of analysis of XML documents and monitoring of recursive programs over infinite domains. We show how S-VPAs can model XML validation policies and program properties that are not naturally expressible with previous formalisms and provide experimental results on the performance of our implementation. For example S-VPAs can naturally express the following properties: an XML document is well-matched (every close tag is the same as the corresponding open tag), every person has age greater than 5, and every person's name starts with a capital letter. Using the closure properties of S-VPAs, all these properties can be expressed as a single deterministic S-VPA that can be efficiently executed.

This chapter is structured as follows:

- Section 4.2 illustrates how symbolic visibly pushdown automata differ from existing models;

Model	Bool. Closure and Decidable Equiv.	Determinizability	Hierarchical Inputs	Infinite Alphabets	Binary Predicates
FA	✓	✓	✗	✗	—
S-FA	✓	✓	✗	✓	—
S-EFA	✗	✗	✗	✓	Adjacent Positions
DA, RA	some variants	✗	trees	✓	Only Equality
VPA	✓	✓	✓	✗	—
S-VPA	✓	✓	✓	✓	Calls/Returns

TABLE 4.1: Properties of different automata models.

- Section 4.3 defines the new model of symbolic visibly pushdown automata;
- Section 4.4 presents new algorithms for intersecting, complementing, and determinizing S-VPAs, and for checking emptiness of S-VPAs;
- Section 4.5 discusses SVPAlib, a prototype implementation of S-VPAs, and its evaluation using XML processing and program monitoring as case-studies.

4.2 Motivating example

In dynamic analysis, program properties are monitored at runtime. Automata theory has been used for specifying monitors. Let x be a global variable of a program P. We can use Finite-state Automata (FA) to describe "correct" values of x during the execution of P. For example, if x has type *bool*, an FA can specify that x is *true* throughout the execution of P.

Infinite domains. In the previous example, x has type *bool*. In practice, one would want to express properties about variables of any type. If x is of type *int* and has infinitely many possible values, FAs do not suffice any more. For example, no FA can express the property φ_{ev} stating that x remains even throughout the whole execution of P. One solution to this problem is to use predicate abstraction and create an alphabet of two symbols $even(x)$ and $\neg even(x)$. However, this solution causes the input alphabet to be different from the original one ($\{even(x), \neg even(x)\}$ instead of the set of integers), and requires choosing a priori which abstraction to use.

Symbolic Finite Automata (S-FA) [DV14, Vea13] solve this problem by allowing transitions to be labeled with predicates over a decidable theory, and enjoy all the closure and decidability properties of finite state automata. The S-FA A_{ev} for the property φ_{ev} has one state looping on an edge labeled with the predicate $even(x)$ expressible in Presburger arithmetic. Unlike predicate abstraction, S-FAs do not change the underlying alphabet and allow predicates to be combined. For example, let A_{pos} be the

S-FA accepting all the sequences of positive integers. When intersecting A_{pos} and A_{ev} the transition predicates will be combined, and we will obtain an S-FA accepting all the sequences containing only integers that are both even and positive. An important restriction is that the underlying theory of the predicates needs to be decidable. For example, the property φ_{pr}, which states that x is a prime number at some point in P, cannot be expressed by an S-FA.

S-FAs allow only unary predicates and cannot relate values at different positions. Symbolic Extended Finite Automata (S-EFA) (Section 2.3) allow binary predicates for comparing adjacent positions, but this extension causes the model to lose closure and decidability properties (Section 2.4). Other models for comparing values over infinite alphabets at different positions are Data Automata (DA) [BDM$^+$11] and Register Automata (RA) [KT08], where one can check that all the symbols in an input sequence are equal for example. This property is not expressible by an S-FA or an S-EFA, but Data Automata can only use equality and cannot specify properties such as $even(x)$.

Procedure calls. Let x be of type *bool* and assume that the program P contains a procedure q. The following property $\varphi_=$ can be specified by neither an FA nor an S-FA: every time q is called, the value of x at the call is the same as the value of x when q returns. The problem is that none of the previous models are able to "remember" which call corresponds to which return. Visibly Pushdown Automata (VPA) [AM09] solve this problem by storing the value of x on a stack at a call and then retrieve it at the corresponding return. Unlike pushdown automata, this model still enjoys closure under Boolean operations and decidable equivalence. This is achieved by making calls and returns visible in the input, and allowing the stack to push only at calls and to pop only at returns.

Procedure calls and infinite domains. Let x be of type *int* and let's assume that the program P contains a procedure q. No VPA can express the property $\psi_<$ requiring that, whenever q is called, the value of x at the call is smaller than the value of x at the corresponding return. Expressing this kind of property in a decidable automata model is the topic of this chapter.

We introduce Symbolic Visibly Pushdown Automata (S-VPA), which combine the features of S-FAs and VPAs by allowing transitions to be labeled with predicates over any decidable theory and values to be stored on a stack at calls and retrieved at the corresponding returns. The property $\psi_<$ can then be expressed by an S-VPA $A_<$ as follows: at a procedure call of q, $A_<$ will store the value c of x on the stack. When reading the value r of x at a procedure return of q, the value c of x at the corresponding call will be on top of the stack. Using the predicate $c < r$, the transition assures that the property $\psi_<$ is met. S-VPAs still enjoy closure under Boolean operations, determinizability, and decidable equivalence; the key to decidability is that binary predicates can only be used to compare values at matching calls and returns (unlike S-EFAs).

4.3 Model definition

Nested words. Data with both linear and hierarchical structure can be encoded using nested words [AM09]. Given a set Σ of symbols, the *tagged alphabet* $\hat{\Sigma}$ consists of the symbols a, $\langle a$, and $a\rangle$, for each $a \in \Sigma$. A *nested word* over Σ is a finite sequence over $\hat{\Sigma}$. For a nested word $a_1 \cdots a_k$, a position j, for $1 \leq j \leq k$, is said to be a *call* position if the symbol a_j is of the form $\langle a$, a *return* position if the symbol a_j is of the form $a\rangle$, and an *internal* position otherwise. The tags induce a matching relation between call and return positions. Nested words can naturally encode strings and ordered trees.

Symbolic visibly pushdown automata. Symbolic Visibly Pushdown Automata (S-VPA) are an executable model for nested words over infinite alphabets. In S-VPAs transitions are labeled with predicates over the input alphabet, analogous to symbolic automata for strings over infinite alphabets. A key novelty of S-VPAs is the use of binary predicates to model relations between open and close tags in a nested word.

We use $\mathbb{P}_x(\Psi)$ and $\mathbb{P}_{x,y}(\Psi)$ to denote the set of unary and binary predicates in Ψ respectively. We assume that every unary predicate in $\mathbb{P}_x(\Psi)$ contains x as the only free variable (similarly $\mathbb{P}_{x,y}(\Psi)$ with x and y).

Definition 4.1 (S-VPA)**.** A (nondeterministic) symbolic visibly pushdown automaton over an alphabet Σ is a tuple $A = (Q, Q_0, P, \delta_i, \delta_c, \delta_r, \delta_b, F)$, where

- Q is a finite set of states;
- $Q_0 \subseteq Q$ is a set of initial states;
- P is a finite set of stack symbols;
- $\delta_i \subseteq Q \times \mathbb{P}_x \times Q$ is a finite set of internal transitions;
- $\delta_c \subseteq Q \times \mathbb{P}_x \times Q \times P$ is a finite set of call transitions;
- $\delta_r \subseteq Q \times \mathbb{P}_{x,y} \times P \times Q$ is a finite set of return transitions;
- $\delta_b \subseteq Q \times \mathbb{P}_x \times Q$ is a finite set of empty-stack return transitions;

- $F \subseteq Q$ is a set of final states.

A transition $(q, \varphi, q') \in \delta_i$, where $\varphi \in \mathbb{P}_x$, when reading a symbol a such that $a \in [\![\varphi]\!]$, starting in state q, updates the state to q'. A transition $(q, \varphi, q', p) \in \delta_c$, where $\varphi \in \mathbb{P}_x$, and $p \in P$, when reading a symbol $\langle a$ such that $a \in [\![\varphi]\!]$, starting in state q, pushes the symbol p on the stack along with the symbol a, and updates the state to q'. A transition $(q, \varphi, p, q') \in \delta_r$, where $\varphi \in \mathbb{P}_{x,y}$, is triggered when reading an input $b\rangle$, starting in state q, and with $(p, a) \in P \times \Sigma$ on top of the stack such that $(a, b) \in [\![\varphi]\!]$; the transition pops the element on the top of the stack and updates the state to q'. A transition $(q, \varphi, q') \in \delta_b$, where $\varphi \in \mathbb{P}_x$, is triggered when reading a tagged input $a\rangle$ such that $a \in [\![\varphi]\!]$, starting in state q, and with the current stack being empty; the transition updates the state to q'.

A stack is a finite sequence over $P \times \Sigma$. We denote by Γ the set of all stacks. Given a nested word $w = a_1 \dots a_k$ in Σ^*, a run of M on w starting in state q is a sequence $\rho_q(w) = (q_1, \theta_1), \dots, (q_{k+1}, \theta_{k+1})$, where $q = q_1$, each $q_i \in Q$, each $\theta_i \in \Gamma$, the initial stack θ_1 is the empty sequence ε, and for every $1 \leq i \leq k$ the following holds:

Internal: if a_i is internal, there exists $(q, \varphi, q') \in \delta_i$, such that $q = q_i$, $q' = q_{i+1}$, $a_i \in [\![\varphi]\!]$, and $\theta_{i+1} = \theta_i$;

Call: if $a_i = \langle a$, for some a, there exists $(q, \varphi, q', p) \in \delta_c$, such that $q = q_i$, $q' = q_{i+1}$, $a \in [\![\varphi]\!]$, and $\theta_{i+1} = \theta_i(p, a)$;

Return: if $a_i = a\rangle$, for some a, there exists $(q, \varphi, p, q') \in \delta_r$, $b \in \Sigma$, and $\theta' \in \Gamma$, such that $q = q_i$, $q' = q_{i+1}$, $\theta_i = \theta'(p, b)$, $\theta_{i+1} = \theta'$, and $(b, a) \in [\![\varphi]\!]$;

Bottom: if $a_i = a\rangle$, for some a, there exists $(q, \varphi, q') \in \delta_b$, such that $q = q_i$, $q' = q_{i+1}$, $\theta_i = \theta_{i+1} = \varepsilon$, and $a \in [\![\varphi]\!]$.

A run is *accepting* if q_1 is an initial state in Q_0 and q_{k+1} is a final state in F. A nested word w is accepted by A if there exists an accepting run of A on w. The language $[\![A]\!]$ accepted by A is the set of nested words accepted by A.

Definition 4.2 (Deterministic S-VPA)**.** A symbolic visibly pushdown automaton A is deterministic iff $|Q_0| = 1$ and

- for each two transitions $t_1 = (q_1, \varphi_1, q'_1), t_2 = (q_2, \varphi_2, q'_2) \in \delta_i$, if $q_1 = q_2$ and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q'_1 = q'_2$;
- for each two transitions $t_1 = (q_1, \varphi_1, q'_1, p_1), t_2 = (q_2, \varphi_2, q'_2, p_2) \in \delta_c$, if $q_1 = q_2$ and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q'_1 = q'_2$ and $p_1 = p_2$;

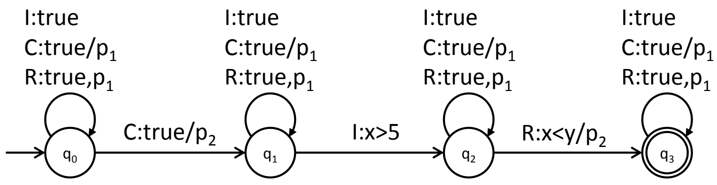


FIGURE 4.1: Example of S-VPA over the theory of integers.

- for each two transitions $t_1 = (q_1, \varphi_1, p_1, q'_1), t_2 = (q_2, \varphi_2, p_2, q'_2) \in \delta_r$, if $q_1 = q_2$, $p_1 = p_2$, and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q'_1 = q'_2$;
- for each two transitions $t_1 = (q_1, \varphi_1, q'_1), t_2 = (q_2, \varphi_2, q'_2) \in \delta_b$, if $q_1 = q_2$, and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q'_1 = q'_2$.

Example 4.3. *Consider the following property of a program trace over the integers: there exists a well-matched sub-trace $w = \langle x\ w_1\ y\ w_2\ z \rangle$ such that $z < x$ and $y < 5$ and both w_1 and w_2 are well-matched nested words. This property can easily be expressed using the nondeterministic S-VPA shown in Fig. 4.1. In the figure the transitions are labeled with I, C, and R for Internal, Call and Return respectively. In the return transition the values x and y respectively refer to the symbols at the return and call respectively.*

For a deterministic S-VPA A we use q_0 to denote the only initial state of A.

We now define complete S-VPAs, which we will use to prove that S-VPAs are closed under complement.

Definition 4.4 (Complete S-VPA)**.** A deterministic symbolic visibly pushdown automaton A is complete iff for each $q \in Q$, $a, b \in \Sigma$, and $p \in P$, there exist

- a transition $(q, \varphi, q') \in \delta_i$, such that $a \in [\![\varphi]\!]$;
- a transition $(q, \varphi, q', p') \in \delta_c$, such that $a \in [\![\varphi]\!]$;
- a transition $(q, \varphi, p, q') \in \delta_r$, such that $(a, b) \in [\![\varphi]\!]$;
- a transition $(q, \varphi, q') \in \delta_b$, such that $a \in [\![\varphi]\!]$.

4.4 Closure Properties and Decision Procedures

In this section we describe the closure and decidability properties of S-VPAs. We first introduce few preliminary concepts and then show how S-VPAs are equivalent in expressiveness to deterministic S-VPAs, and complete S-VPAs. We then prove that S-VPAs are closed under Boolean operations. Last, we provide an algorithm for checking emptiness of S-VPAs over decidable label theories and use it to prove the decidability of S-VPA language equivalence. For each construction we provide a complexity parameterized by the underlying theory, and we assume that transitions are only added to a construction when satisfiable.

4.4.1 Closure Properties

Before describing the determinization algorithm we introduce the concept of a minterm. The notion of a minterm is fundamental for determinizing symbolic automata, and it captures the set of equivalence classes of the input alphabet for a given symbolic automaton. Intuitively, for every state q of the symbolic automaton, a minterm is a set of input symbols that q will always treat in the same manner. Given a set of predicates Φ a *minterm* is a minimal satisfiable Boolean combination of all predicates that occur in Φ. We use the notation $Mt(\Phi)$ to denote the set of minterms of Φ. For example the set of predicates $\Phi = \{x > 2, x < 5\}$ over the theory of linear integer arithmetic has minterms $Mt(\Phi) = \{x > 2 \wedge x < 5,\ \neg x > 2 \wedge x < 5,\ x > 2 \wedge \neg x < 5\}$. While in the case of symbolic finite automata this definition is simpler (see [DV14]), in our setting we need to pay extra attention to the presence of binary predicates. We need therefore to define two types of minterms, one for unary predicates and one for binary predicates. Given an S-VPA A we define

- the set Φ_1^A of unary predicates of A as the set $\{\varphi \mid \exists q, q', p.(q, \varphi, q') \in \delta_i \vee (q, \varphi, q', p) \in \delta_c \vee (q, \varphi, q') \in \delta_b\}$;
- the set Φ_2^A of binary predicates of A as the set $\{\varphi \mid \exists q, q', p.(q, \varphi, p, q') \in \delta_r\}$;
- the set Mt_1^A as the set $Mt(\Phi_1^A)$ of unary predicate minterms of A;
- the set Mt_2^A as the set $Mt(\Phi_2^A)$ of binary predicate minterms of A.

The goal of minterms is that of capturing the equivalence classes of the label theory in the current S-VPA. Let Φ be the set of minterms of an S-VPA A. Consider two nested words $s = a_1 \ldots a_n$ and $t = b_1 \ldots b_n$ of equal length and such that for every i, a_i has the same tag as b_i (both internals, etc.). Now assume the following is true: for every $1 \leq i \leq n$, if a_i is internal there exists a minterm $\varphi \in Mt_1^A$ such that both a_i and b_i are models of φ, and, if a_i is a call with corresponding return a_j, then there exists a minterm $\psi \in Mt_2^A$ such that both (a_i, a_j) and (b_i, b_j) are models of ψ. If the previous condition holds, the two nested words will be indistinguishable in the S-VPA A, meaning that they will have exactly the same set of runs. Following, this intuition we have that even though the alphabet might be infinite, only a finite number of predicates is *interesting*. We can now discuss the determinization construction.

Theorem 4.5 (Determinization)**.** *For every S-VPA A there exists a deterministic S-VPA B accepting the same language.*

Proof. The main difference between the determinization algorithm from Alur et al. [AM09] and the symbolic version is in the use of minterms. Similarly to the approach used to minimize symbolic automata [DV14], we use the minterm computation to generate a finite set of relevant predicates. After we have done so, we can generalize the determinization construction shown by Alur et al. [AM09].

We now describe the intuition behind the construction. Given a nested word n, A can have multiple runs over n. Thus, at any position, the state of B needs to keep track of all possible states of A, as in the case of the subset construction for determinizing nondeterministic word automata. However, keeping only a set of states of A is not enough: at a return position, B needs to use the information on the top of the stack and in the state to figure out which pairs of states (starting in state q you can reach state q') belong to the same run. The main idea behind the construction is to do a subset construction over summaries (pairs of states) but postpone handling the call-transitions by storing the set of summaries before the call, along with the minterm containing the call symbol, in the stack, and simulate the effect of the corresponding call-transition at the time of the matching return for every possible minterm.

Consider a nested word with k pending calls. Such a word can be represented as $n = n_1 \langle a_1 n_2 \langle a_2 \cdots n_k \langle a_k n_{k+1}$, where each n_j , for $1 \leq j \leq k+1$, is a nested word with no pending calls (the initial nested word n_1 can have pending returns, and the nested words $n_2, \ldots, n_{k+1}$ are well-matched). Then after processing n, the determinized automaton B we construct will be in state S_{k+1}, with the pair (S_k, φ_k) on top of the stack, where φ_k is the minterm predicate containing the symbol a_k. Here S_k contains all the pairs (q, q') of states of A such that the nondeterministic automaton A can process the nested word n_k starting in state q and end up in state q'. Note that the execution of the automaton A while processing the nested word n_k depends solely on the state at the beginning of the word n_k, and not on the states labeling the pending nesting edges. The construction ensures that only "reachable" summaries are being tracked: if a pair (q, q') belongs to S_k , it is guaranteed that there is a run of A on the nested word n in which the state at the beginning of processing of the subword n_k is q and at the end is q'. Due to this property, in order to check whether A accepts n or not, corresponds to checking if the current state S_{k+1} contains a pair (q_0, q') such that q' is a final state.

The components of the deterministic automaton B equivalent to $A = (Q, Q_0, P, \delta_c, \delta_i, \delta_r, \delta_b, Q_F)$ are the following.

- The states of B are $Q' = 2^{Q \times Q}$.

- The initial state is the set $Q_0 \times Q_0$ of pairs of initial states.
- *A* state $S \in Q'$ is accepting iff it contains a pair of the form (q, q') with $q' \in Q_f$.
- The stack symbols of B are $P' = Q' \times Mt_1^A$.
- The internal transition function δ'_i is given by: for $S \in Q'$, and $\varphi \in Mt_1^A$, $\delta'_i(S, \varphi)$ consists of pairs (q, q'') such that there exists $(q, q') \in S$ and an internal transition $(q', \varphi', q'') \in \delta_i$ such that $IsSat(\varphi \wedge \varphi')$.
- The call transition function δ'_c is given by: for $S \in Q'$ and $\varphi \in Mt_1^A$, $\delta'_c(S, \varphi) = (S', (S, \varphi))$, where S' consists of pairs (q'', q'') such that there exists $(q, q') \in S$, a stack symbol $p \in P$, and a call transition $(q', \varphi', q'', p) \in \delta_c$ such that $IsSat(\varphi \wedge \varphi')$.
- The return transition function δ'_r is given by: for $S, S' \in Q'$ and $\varphi_1 \in Mt_1^A, \varphi_2 \in Mt_2^A$, the state $\delta'_r(S, (S', \varphi_1), \varphi_2)$ consists of pairs (q, q'') such that there exists $(q, q') \in S'$, $(q_1, q_2) \in S$, a stack symbol $p \in P$, a call transition $(q', \varphi'_1, q_1, p) \in \delta_c$, and a return transition $(q_2, p, \varphi'_2, q'') \in \delta_r$ such that $IsSat(\varphi_1 \wedge \varphi'_1)$ and $IsSat(\varphi_2 \wedge \varphi'_2)$.
- The empty-stack return transition function δ'_b is given by: for $S \in Q'$ and $\varphi \in Mt_1^A$, the state $\delta'_b(S, \varphi)$ consists of pairs (q, q'') such that there exists $(q, q') \in S$ and a return transition $(q', \varphi', q'') \in \delta_b$ such that $IsSat(\varphi \wedge \varphi')$.

Our construction differs from the one by Alur et al. [AM09] in two aspects.

- In the case of finite alphabets, each stack symbol contains an element from Σ. This technique cannot be used in our setting and in our construction each stack symbol contains a predicate from the set of unary minterms.
- The construction by Alur et al. [AM09] builds on the notion of reachability and looks for matching pairs of calls and returns. In our construction, this operation has to be performed symbolically, by checking whether the unary predicate stored by the call on the stack and the binary predicate at the return are not disjoint.

We finally discuss the complexity of the determinization procedure. Assume A has n states, m stack symbols, and p different predicates of size at most ℓ. We first observe that the number of minterms is at most 2^p and each minterm has size $O(p\ell)$. Notice that if the alphabet is finite the number of minterms is bounded by $min(2^p, |\Sigma|)$. If $f(a)$ is the cost of checking the satisfiability of a predicate of size a, then the minterm computation has complexity $O(2^p f(\ell p))$. The resulting automaton B has $O(2^{n^2})$ states, and $O(2^p 2^{n^2})$ stack symbols. The determinization procedure has complexity $O(2^p 2^{n^2} m + 2^p f(p\ell))$. □

Theorem 4.6 (Completeness). *For every S-VPA A there exists a complete S-VPA B accepting the same language.*

Proof. An S-VPA can be completed by adding a sink state q_{sink} and adding a new transition from each state q to q_{sink} covering the inputs for which q has no transition. Finally q_{sink} has a loop transition on the predicate *true*. Given an S-VPA A we use Theorem 4.5 to build an equivalent deterministic S-VPA $A' = (Q, q_0, P, \delta_i, \delta_c, \delta_r, \delta_b, Q_F)$. Next we construct a complete S-VPA $B = (Q \cup \{q_{sink}\}, q_0, P, \delta'_i, \delta'_c, \delta'_r, \delta'_b, Q_F)$ equivalent to A'. The new state q_{sink} is used as a sink for the missing transitions. We now define the new transition relation. We only show δ'_i, since the other transition functions are analogous. Intuitively, for every state, all the symbols for which no internal transition exists are directed to the sink state. Let $I_q = \{\varphi \mid (q, \varphi, q') \in \delta_i\}$. Then $\delta'_i = \delta_i \cup \{(q, \neg\psi, q_{sink}) \mid \psi = \bigvee_{\varphi \in I_q} \varphi\} \cup \{(q_{sink}, true, q_{sink})\}$. The fact that A' is equivalent to B follows from the definitions of acceptance and run.

Assume A has n states, m stack symbols, and p different predicates of size at most ℓ. Let $f(a)$ be the cost of checking the satisfiability of a predicate of size a. The procedure has complexity $O(nmf(\ell p))$. □

Theorem 4.7 (Boolean Closure). *S-VPAs are closed under Boolean operations.*

Proof. We prove that S-VPAs are closed under complement and intersection. We first prove that S-VPAs are closed under complement. Given an S-VPA A we construct a complete S-VPA C such that C accepts a nested word n iff n is not accepted by A. First, we use Theorem 4.6 to construct an equivalent deterministic S-VPA $B = (Q, q_0, P, \delta_i, \delta_c, \delta_r, \delta_b, Q_F)$. We can now construct the S-VPA $C = (Q, q_0, P, \delta_i, \delta_c, \delta_r, \delta_b, Q \setminus Q_F)$ in which the set of accepting states is complemented.

We next prove that S-VPAs are closed under intersection. Given two deterministic S-VPAs $A_1 = (Q^1, q^1_0, P^1, \delta^1_i, \delta^1_c, \delta^1_r, \delta^1_b, Q^1_F)$ and $A_2 = (Q^2, q^2_0, P^2, \delta^2_i, \delta^2_c, \delta^2_r, \delta^2_b, Q^2_F)$ (using Theorem 4.5) we construct an S-VPA B such that B accepts a nested word n iff n is accepted by both A_1 and A_2. The construction of B is a product construction. The S-VPA B will have state set $Q' = Q^1 \times Q^2$, initial state $q'_0 = (q^1_0, q^2_0)$, stack symbol set $P' = P^1 \times P^2$, and final state set $Q'_F = Q^1_F \times Q^2_F$, The transition function will simulate both A_1 and A_2 at the same time.

- for each $(q_1, \varphi_1, q'_1) \in \delta^1_i$, and $(q_2, \varphi_2, q'_2) \in \delta^2_i$, δ'_i will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (q'_1, q'_2))$;
- for each $(q_1, \varphi_1, q'_1, p_1) \in \delta^1_c$, and $(q_2, \varphi_2, q'_2, p_2) \in \delta^2_c$, and δ'_c will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (q'_1, q'_2), (p_1, p_2))$;

- for each $(q_1, \varphi_1, p_1, q_1') \in \delta_r^1$, and $(q_2, \varphi_2, p_2, q_2') \in \delta_r^2$, δ_r' will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (p_1, p_2), (q_1', q_2'))$;
- for each $(q_1, \varphi_1, q_1') \in \delta_b^1$, and $(q_2, \varphi_2, q_2') \in \delta_b^2$, δ_b' will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (q_1', q_2'))$.

Assume each S-VPA A_i has n_i states, m_i stack symbols, and p_i different predicates of size at most ℓ_i. Let $f(a)$ be the cost of checking the satisfiability of a predicate of size a. The intersection procedure has complexity $O(n_1 n_2 m_1 m_2 + p_1 p_2 f(\ell_1 + \ell_2))$. □

4.4.2 Decision Procedures

We conclude this section with an algorithm for checking emptiness of S-VPAs over decidable label theories, which we finally use to prove the decidability of S-VPA equivalence.

Theorem 4.8 (Emptiness). *Given an S-VPA A over a decidable label theory it is decidable whether $L(A) = \emptyset$.*

Proof. The algorithm for checking emptiness is a symbolic variant of the algorithm for checking emptiness of a pushdown automaton. We are given an S-VPA $A = (Q, q_0, P, \delta_i, \delta_c, \delta_r, \delta_b, Q_F)$ over a decidable theory Ψ. First, for every two states $q, q' \in Q$ we compute the reachability relation $R_{wm} \subseteq Q \times Q$ such that $(q, q') \in R_{wm}$ iff there exists a run $\rho_q(w)$ that, starting in state q, after reading a well matched nested word w, ends in state q'. We define R_{wm} as follows:

- for all $q \in Q$, $(q, q) \in R_{wm}$;
- if $(q_1, q_2) \in R_{wm}$, and there exists $q, q' \in Q$, $p \in P$, $\varphi_1 \in \mathbb{P}_x(\Psi)$, $\varphi_2 \in \mathbb{P}_{x,y}(\Psi)$, such that $(q, \varphi_1, q_1, p) \in \delta_c$, $(q_2, \varphi_2, p, q') \in \delta_r$, and $IsSat(\varphi_1 \wedge \varphi_2)$, then $(q, q') \in R_{wm}$. Observe that unary and binary predicates unify on the first variable x;
- if $(q_1, q_2) \in R_{wm}$, and there exists $q \in Q$, $\varphi \in \mathbb{P}_x(\Psi)$, such that $(q, \varphi, q_1) \in \delta_i$, and $IsSat(\varphi)$, then $(q, q_2) \in R_{wm}$;
- if $(q_1, q_2) \in R_{wm}$ and $(q_2, q_3) \in R_{wm}$, then $(q_1, q_3) \in R_{wm}$.

The above reachability relation captures all the runs over well-matched nested words. Unmatched calls and returns can be handled using a similar set of rules: we can define the relation $R_c \subseteq Q \times Q$ such that $(q, q') \in R_c$ iff there exists a run $\rho_q(n)$ that ends in state q' after reading a nested word n with zero or more unmatched calls and no unmatched returns.

- if $(q_1, q_2) \in R_{wm}$, then $(q_1, q_2) \in R_c$;
- if $(q_1, q_2) \in R_c$, and there exists $q \in Q$, $p \in P$, $\varphi \in \mathbb{P}_x(\Psi)$, such that $(q, \varphi, p, q_1) \in \delta_c$, and $IsSat(\varphi)$, then $(q, q_2) \in R_c$;
- if $(q_1, q_2) \in R_c$ and $(q_2, q_3) \in R_c$, then $(q_1, q_3) \in R_c$.

Next, we handle unmatched returns, and define the relation $R_r \subseteq Q \times Q$ such that $(q, q') \in R_r$ iff there exists a run $\rho_q(n)$ that ends in state q' after reading a nested word n with zero or more unmatched returns and no unmatched calls.

- if $(q_1, q_2) \in R_{wm}$, then $(q_1, q_2) \in R_r$;
- if $(q_1, q_2) \in R_r$, and there exists $q \in Q$, $\varphi \in \mathbb{P}_x(\Psi)$, such that $(q, \varphi, q_1) \in \delta_b$, and $IsSat(\varphi)$, then $(q, q_2) \in R_r$;
- if $(q_1, q_2) \in R_r$ and $(q_2, q_3) \in R_r$, then $(q_1, q_3) \in R_r$.

Finally we can combine R_c and R_r into a final reachability relation $R \subseteq Q \times Q$ such that $(q, q') \in R$ iff there exists a run $\rho_q(w)$ that ends in state q' after reading a nested word w: if $(q_1, q_2) \in R_r$, and $(q_2, q_3) \in R_c$, then $(q_1, q_3) \in R$. The S-VPA A is empty iff $(Q_0 \times Q_F) \cap R = \emptyset$.

Assume A has n states, m stack symbols, t transitions, and p predicates of size at most ℓ. Let $f(a)$ be the cost of checking the satisfiability of a predicate of size a. The emptiness procedure has complexity $O(n^3 mt + p^2 f(\ell))$. □

We can now combine the closure under Boolean operations and the decidability of emptiness to show that equivalence of S-VPAs is decidable.

Corollary 4.9 (Equivalence)**.** *Given two S-VPAs A and B over a decidable label theory it is decidable whether $L(A) \subseteq L(B)$ and whether $L(A) = L(B)$.*

VPA universality, inclusion, and equivalence problems are EXPTIME-hard [AM09]. If the function *IsSat* can be computed in polynomial time the same complexity bounds hold for S-VPAs.

4.5 Applications and Evaluation

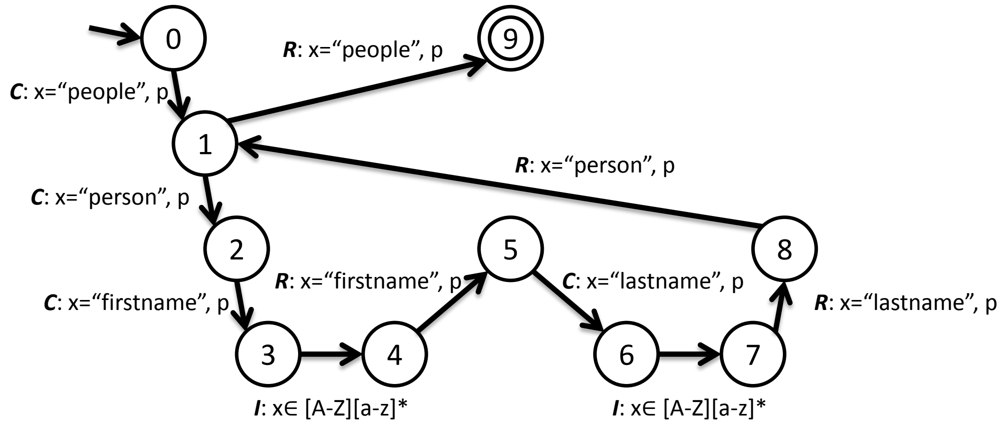
In this section we present potential applications of S-VPAs together with experimental results. First, we illustrate how the presence of symbolic alphabets and closure properties enables complex XML validation, HTML sanitization, and runtime monitoring

```
<xs:schema>
<xs:element name="people"  type="PeopleType"/>
<xs:complexType name="PeopleType"><xs:sequence>
 <xs:element name="person" minOccurs="0">
  <xs:complexType><xs:sequence>
   <xs:element name="firstname">
    <xs:simpleType><xs:restriction base="xs:string">
     <xs:pattern value="[A-Z]([a-z])*"/>
    </xs:restriction></xs:simpleType>
   </xs:element>
   <xs:element name="lastname">
    <xs:simpleType><xs:restriction base="xs:string">
     <xs:pattern value="[A-Z]([a-z])*"/>
    </xs:restriction></xs:simpleType>
   </xs:element>
  </xs:sequence></xs:complexType>
 </xs:element>
</xs:sequence></xs:complexType>
</xs:schema>
```

(1) XML schema S

```
<people>
 <person>
  <firstname>
    Mark
  </firstname>
  <lastname>
    Red
  </lastname>
 </person>
 <person>
  <firstname>
    Mario
  </firstname>
  <lastname>
    Rossi
  </lastname>
 </person>
</people>
```

(2) Document example



(3) S-VPA A_s

FIGURE 4.2: Example XML Schema with an equivalent S-VPA.

of recursive programs. Finally, we present some experimental results on executing S-VPA and their algorithms. All the experiments were run on a 4 Cores Intel i7-2600 CPU 3.40GHz, with 8GB of RAM. and with the SVPAlib library configured for 32 bits architecture. The SVPAlib library is available at `https://github.com/lorisdanto/symbolicautomata`.

4.5.1 XML Validation

XML and HTML documents are ubiquitous. Validating an XML document is the task of checking whether such a document meets a given specification. XML Schema is the most common language for writing XML specifications and their properties have been studied in depth [MNS09, DZL03]. The XML schema S shown in Figure 4.2 describes the format of XML documents containing first and last names. In words the document should start with the tag `people` and then contain a sequence of `person` each of which

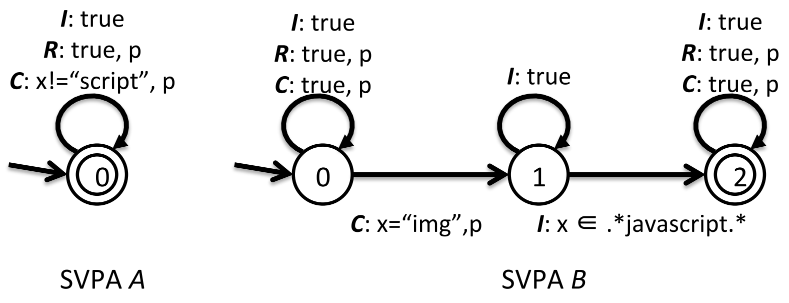


FIGURE 4.3: S-VPAs for HTML filtering.

has a first and last name. First and last name should be both strings belonging to the regular expression `[A-Z]([a-z])*`.

Dealing with infinite alphabets. Although the XML Schema in Figure 4.2 only uses a finite set of possible nodes (`people`, `firstname`, etc.), it allows the content of the leaves to be any string accepted by the regular expression `[A-Z]([a-z])*`. This kind of constraints can be easily captured using an S-VPA over the theory of strings. Such a S-VPA A_S is depicted in Figure 4.2.3. We encode each XML document as a nested word over the theory of strings. For each open tag, close tag, attribute, attribute value, and text node the nested word contains one input symbol with the corresponding value. The letters *I*, *C*, and *R* on each transition respectively stand for internal, call, and return transitions.

Although in this particular setting the alphabet could be made finite by linearizing each string, such an encoding would not be natural and would cause the corresponding VPA to be very complex. Moreover previous models that use such an encoding require the parser to further split each node value into separate characters [KCTV07, MNS09]. In the case of S-VPAs, as can be observed in Figure 4.2, there is a clear separation between the constraints on the tree structure (captured by the states) and the constraints on the leaves (captured by the predicates). This natural representation makes the model succinct and easy to operate on, since it reflects the typical representation of XML via events (SAX parser, etc.).

4.5.2 HTML Filters

A central concern for secure web applications is untrusted user inputs. These lead to cross-site scripting (XSS) attacks, which may echo an untrusted input verbatim back to the browser. HTML filters aim at blocking potentially malicious user HTML code from being executed on the server. For example, a security-sensitive application might

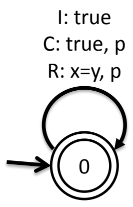


FIGURE 4.4: S-VPA W accepting well-formed HTML documents.

want to discard all documents containing `script` nodes which might contain malicious JavaScript code (this is commonly done in HTML sanitization). Since HTML5 lets the users define custom tags, the set of possible node names is infinite and cannot be known a priori. In this particular setting, an HTML schema would not be able to characterize such an HTML filter. This simple property can be checked using an S-VPA over the theory of strings. Such an S-VPA A is depicted on the left of Figure 4.3. The S-VPA A only accepts nested words that do not contain `script` nodes. Notice that the call transition is triggered by any string different from `script` and the alphabet is therefore infinite.

Since S-VPAs can be intersected, complemented, and determinized, we can take advantage of these properties to make the design of HTML filters modular. We now consider an example for which it is much simpler to specify what it means for a document to be malicious rather than to be safe. On the right of Figure 4.3 it is shown a non-deterministic S-VPA B for checking whether a `img` tag may call JavaScript code in one of its attributes. To compute our filter (the set of safe inputs) we can now compute the complement B' of B that only accepts HTML documents that do not contain malicious `img` tags.

We can now combine A and B' into a single filter. This can be easily done by computing the intersection $F = A \cap B'$. If necessary, the S-VPA F can then be determinized, obtaining an executable filter that can efficiently process HTML documents with a single left-to-right pass.

The power of binary predicates. The previous HTML filter is meant to process only well-formed HTML documents. A well-formed HTML document is one in which all the open tags are correctly matched (every open tag is closed by a close tag containing the same symbol). In practice the input document goes first through a well-formedness checker and then through a filter. This causes the input HTML to be processed multiple times and in performance-critical applications this is not feasible. This check can however be performed by the S-VPA W in Figure 4.4.

Algorithm 1 Recursive implementation of Fibonacci.

```
function $Fib$(int $x$)
    if $x < 2$ then return $x$
    return $Fib(x-1) + Fib(x-2)$
```

4.5.3 Runtime Program Monitors

We already discussed in Section 4.2 how S-VPAs are useful for defining monitors for dynamic analysis of programs. In this section we present an example of how S-VPAs can be used to express complex properties about programs over infinite domains such as integers. Consider the recursive implementation of Fibonacci shown in Algorithm 1. Let's assume we are interested into monitoring the values of x at every call of *Fib*, and the values returned by *Fib*. For example for the input 5, our monitored nested word will be $\langle 2\ \langle 1\ 1\rangle\ \langle 0\ 0\rangle\ 1\rangle$. The following properties can all be expressed using S-VPAs:

1. if the input of *Fib* is greater or equal than 0, then the same holds for all the subsequent inputs of *Fib*;
2. if the output of *Fib* is negative, than *Fib* was called exactly once in the whole execution and with a negative input;
3. the output of *Fib* is greater or equal than the corresponding input.

We can then intersect the S-VPAs corresponding to each property and generate a single pass linear time monitor for *Fib*. As we discussed in Section 4.2, S-VPAs cannot express properties that relate all the values in the computation such as: the value of a variable x increases monotonically throughout the computation. However, thanks to the presence of binary predicates at returns, S-VPAs provide a model for describing pre and post conditions of programs over decidable theories (see property 3). In particular, S-VPAs can describe post conditions that relate the values of the inputs and the outputs of a function.

4.5.4 Experimental results

Execution performance. We implemented the filter $F = A \cap \bar{B} \cap W$ and analyzed the performance of filtering HTML documents with size between 8 and 1293 KB, depth between 3 and 11, number of tokens between 1305 and 84242, and average token length between 11 and 14 characters. To solve the underlying theory of equality plus regular constraints, we implemented a solver on top of the Microsoft Automata library [VB12a].

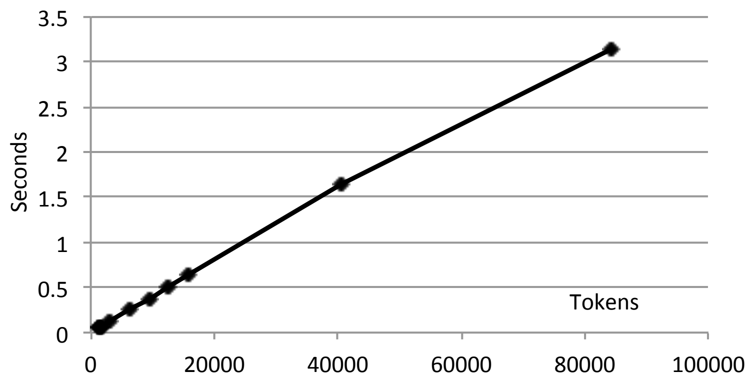


FIGURE 4.5: Time to filter an HTML file.

We decided not to use a full blown solver such as Hampi [KGG$^+$09], since it only supports strings of fixed length. Constructing the S-VPA F took 435 milliseconds. The running times per number of tokens (in seconds) are shown in the figure on the right. We observed that the depth of the input does not affect the running time, while the length affects it linearly. Surprisingly, the running time is also not affected by the average length of the tokens. This is due to the fact that most tokens can be rejected by partially reading them.

Algorithms performance: data. We evaluated the determinization and equivalence algorithms on a representative set of S-VPAs over three different alphabet theories: strings, integers, and bitvectors (characters). The characters and strings solver are implemented on top of the Automata library [VB12a] which is based on BDDs, while the integer solver is Z3 [DMB08]. For each theory t we generated an initial set of S-VPAs S_1^t containing 5 nondeterministic S-VPAs for properties of the following form:

1. the input contains a call and matching return with different symbols;
2. the input contains an internal symbol satisfying a predicate φ_0;
3. the input contains a subword $\langle a\ b\ c\rangle$ such that $a \in [\![\varphi_1]\!]$, $b \in [\![\varphi_2]\!]$, and $a = c$;
4. the input contains a subword $\langle a\ \langle b$ such that $a \in [\![\varphi_3]\!]$, $b \in [\![\varphi_4]\!]$;
5. for every internal symbol a in the input, $a \in [\![\varphi_4]\!]$, or $a \in [\![\varphi_5]\!]$.

The predicates $\Phi = \{\varphi_0, \ldots, \varphi_5\}$ vary for each theory and are all different. For each theory we then computed the set $S_2^t = \{A \cap B \mid A, B \in S_1^t\} \cup \{A \cap B \cap C \mid A, B, C \in S_1^t\}$, since S-VPAs generated through Boolean operations are representative of S-VPAs appearing in applications such as program monitoring. We used the sets S_2^t to evaluate the determinization algorithm, and computed the corresponding set of deterministic S-VPAs D_2^t. Finally we checked equivalence of any two S-VPAs A and B in D_2^t.

The results of our experiments are shown in Figure 4.6: the left column shows the size of each test set and the number of instances for which the algorithms timed out (5 minutes). The right column shows the running time for the instances in which the algorithms did not time out. For both algorithms we plot against number of states and number of transitions. For the determinization, the sizes refer to the automaton before determinization, while in the case of equivalence, the sizes refer to the sum of the corresponding metrics of the two input S-VPAs. The distribution of the sizes of the S-VPAs differed slightly when varying the theories, but since the differences are very small we show the average sizes. For each theory we determinized a total of 65 S-VPAs and checked for equivalence 241 pairs of S-VPAs. For both operation, on average, 96% of the time is spent in the theory solver. For the theory of characters we also compared our tool to the VPALib library, a Java implementation of VPAs.[1] The application timed out for all the inputs considered in our experiments.

Algorithms performance: data analysis. Except for few instances involving the theory of integers, our implementation was able to determinize all the considered S-VPAs in less than 5 minutes. The situation was different in the case of equivalence, where most of the input pairs with more than 250 transitions or 13 states timed out. Most of such pairs were required to check satisfiability of more than 15000 predicates that were generated when building the intersected S-VPAs necessary to check equivalence. We could observe that the theory of characters is on average 6 times faster than the theory of strings and 10 times faster than the theory of integers. However, we did observe that the theory of integers timed out less often in the case of equivalence. We believe that this is due to the different choices of predicates in Φ and to the fact that Z3 uses a caching mechanism that avoids checking for satisfiability of the same predicate twice. While during determinization such a technique is not very beneficial due to the limited number of different minterms, in the case of equivalence, especially for bigger inputs, many of the predicates are repeated, making caching useful in practice.

When comparing against an existing implementation of VPAs, we observed that the benefit of using S-VPA is immense: due to the large size of the alphabet (2^{16} characters), VPALib timed out for each input we considered.

[1]Available http://www.emn.fr/z-info/hnguyen/vpa/

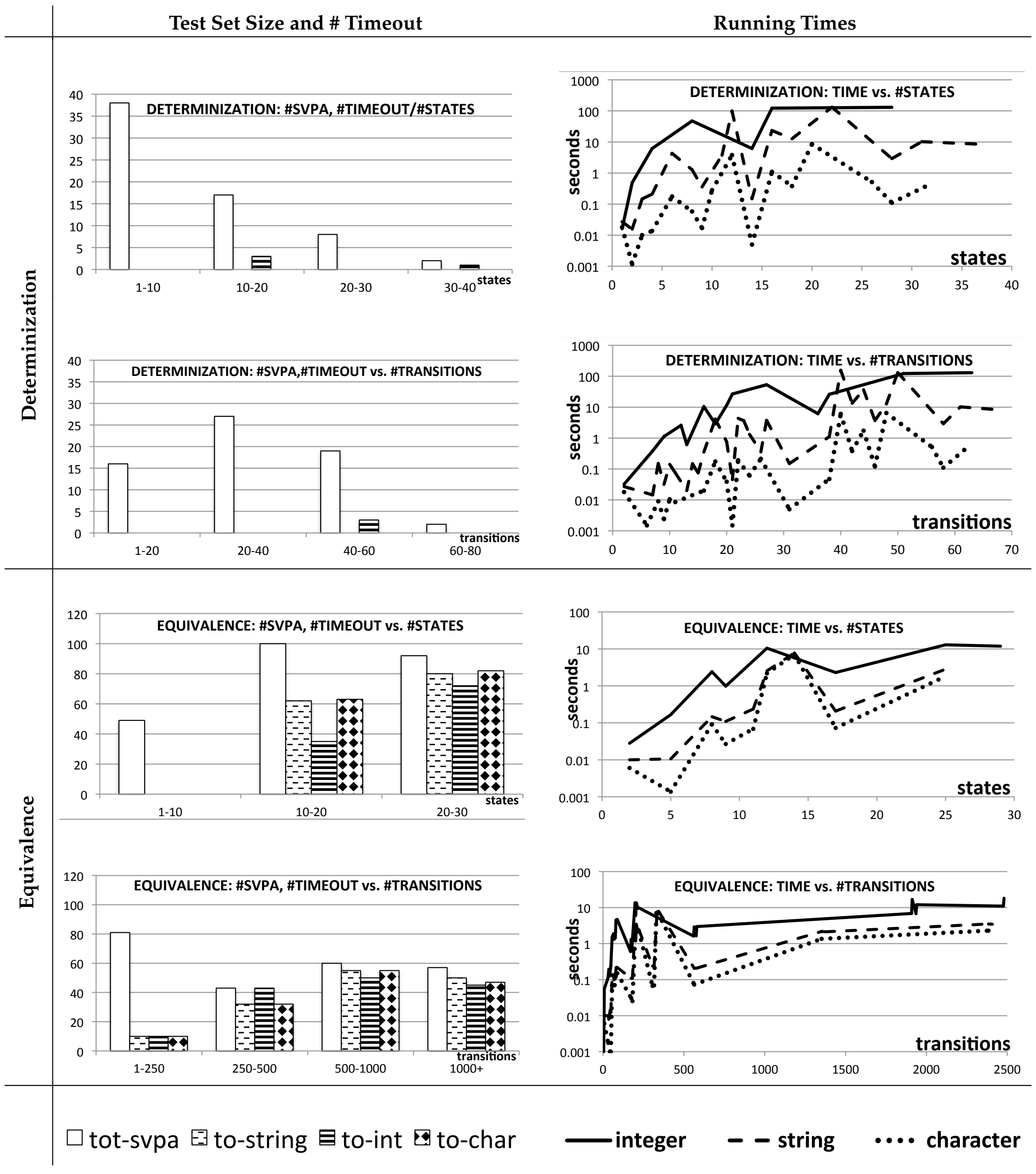


FIGURE 4.6: Running times for S-VPA equivalence and determinization.

Chapter 5

Streaming tree transducers

"I will go anywhere, provided it be forward."
— David Livingstone

5.1 Introduction

As we discussed in Chapter 3, tree structures are ubiquitous. While we showed that FAST can be used to answer many verification questions, due to the use of regular look-ahead, S-TTRs require two passes to process an input tree. In Chapter 4, on the other hand, we argued about the importance of processing XML documents in a single pass [NSV04, MSV00, MGHK09]. In this chapter, we present *streaming tree transducers*, an expressive and analyzable single-pass transducer model for tree transformations.

5.1.1 Existing models of tree transducers

An ideal tree transducer model should:

- capture a large class of transformations;
- enjoy decidable equivalence and type-checking;
- be closed under composition and regular look-ahead;
- operate over strings, ranked trees, and unranked trees;
- compute the output in a single linear-time pass over the input.

The first three requirements ask us to strike the right balance between expressiveness and decidability: *What is the largest class of tree transformations that enjoys decidable procedures and good closure properties?* The most widely accepted answer to this question is the notion of monadic-second-order-logic-definable tree transductions [Cou94]. This formalism represents tree transductions as graph transformations expressed using monadic second-order (MSO) logic formulas over nodes and edges. The class of MSO-definable transductions enjoys decidable equivalence and type-checking, and is closed under sequential composition and regular look-ahead. Moreover, strings, ranked trees, and unranked trees are naturally expressible in this formalism. MSO-definable transformations include complex ones such as swapping subtrees and reversing the order of children in an unranked tree. However, due to the declarative nature, transformations expressed using MSO are hard to execute efficiently on a given input.

Several transducer models have been proposed to address this limitation, but these models typically sacrificed other properties. Executable models for tree transducers include bottom-up tree transducers, visibly pushdown transducers [RS09], and multi bottom-up tree transducers [ELM08]. Each of these models computes the output in a single left-to-right pass in linear time. However, none of these models can compute all MSO-definable transductions.

Finite copying Macro Tree Transducers (MTTs) with regular look-ahead [EM99] can compute all MSO-definable ranked-tree-to-ranked-tree transductions. MTTs are top-down transducers enriched with parameters that can store intermediate computations. In this model, regular look-ahead cannot be eliminated without sacrificing expressiveness and the executing the model requires multiple passes [Man02].

Finally, most models of tree transducers do not naturally generalize to unranked trees. Exceptions include visibly pushdown transducers [RS09] and macro forest transducers [PS04], but these suffer the other limitations we just discussed.

5.1.2 Contributions

We propose the model of *streaming tree transducers* (STT), which has many desirable properties.

Expressiveness: STTs capture exactly the class of MSO-definable tree transductions.

Analyzability: Decision problems such as type-checking and checking whether two STTs are functionally equivalent are decidable.

Closure properties: STTs are closed under composition and regular look-ahead.

Flexibility: STTs can operate over strings, ranked trees, and unranked trees.

Single-pass linear-time processing: An STT is a deterministic machine that computes the output using a single left-to-right pass through the linear encoding of the input tree, processing each symbol in constant time.

The transducer model integrates features of *visibly pushdown automata*, equivalently *nested word automata* [AM09], and *streaming string transducers* [AC10, AC11]. In our model, the input tree is encoded as a *nested word*, which is a string over alphabet symbols, tagged with open/close brackets (or equivalently, call/return types) to indicate the hierarchical structure. For example, the tree $a(b, c(d, d))$ is encoded by the nested word

$$\langle a \ \langle b \ b \rangle \ \langle c \ \langle d \ d \rangle \ \langle d \ d \rangle \ c \rangle \ a \rangle.$$

The streaming tree transducer reads the input nested word left-to-right in a single pass. It uses finitely many states, together with a stack, but the type of operation applied to the stack at each step is determined by the hierarchical structure of the tags in the input. The output is computed using a finite set of variables with values ranging over output nested words, possibly with *holes* that are used as place-holders for inserting subtrees. At each step, the transducer reads the next symbol of the input. If the symbol is an internal symbol, then the transducer updates its state and the output variables. If the symbol is a call symbol, then the transducer pushes a stack symbol along with updated values of variables, updates the state, and reinitializes the variables. While processing a return symbol, the stack is popped, and the new state and new values for the variables are determined using the current state, current variables, popped symbol, and popped values from the stack. In each type of transition, the variables are updated using expressions that allow adding new symbols, *string concatenation*, and *tree insertion* (simulated by replacing the hole with another expression). A key restriction is that variables are updated in a manner that ensures that each value can contribute at most once to the eventual output, without duplication. This *single-use restriction* is enforced via a binary *conflict* relation over variables: no output term combines conflicting variables, and variable occurrences in right-hand sides during each update are consistent with the conflict relation. The transformation computed by the model can be implemented as a single-pass linear-time algorithm.

We show that the model can be simplified in natural ways if we want to restrict either the input or the output to either strings or ranked trees. For example, to compute transformations that output strings it suffices to consider variable updates that allow only concatenation, and to compute transformations that output ranked trees it suffices to consider variable updates that allow only tree insertion. The restriction to the case of ranked trees as inputs gives the model of *bottom-up ranked-tree transducers*. As far as we know, this is the only transducer model that processes trees in a bottom-up manner and can compute all MSO-definable tree transformations.

The main technical result in the chapter is that the class of transductions definable using streaming tree transducers is exactly the class of MSO-definable tree transductions. The starting point for our result is the known equivalence of MSO-definable tree transductions and Macro Tree Transducers with regular look-ahead and single-use restriction, over ranked trees [EM99]. Our proof proceeds by establishing two key properties of STTs: STTs are closed under *regular look-ahead* and under *sequential composition*. These proofs are challenging due to the requirement that a transducer can use only a fixed number of variables and such variables can only be updated by assignments which obey the single-use-restriction rules. We develop the proofs in a modular fashion by introducing intermediate results (for example, we establish that allowing variables to range over trees containing multiple parameters does not increase expressiveness). In this chapter we do not address the problem of representing infinite alphabets.

We show a variety of analysis questions to be decidable for STTs. We establish an EXPTIME upper bound for type-checking, and provide a NEXPTIME upper bound for checking functional inequivalence of two STTs. This is the first elementary upper bound for checking equivalence of a model that captures MSO-definable transformations. When the number of variables is bounded the upper bound on the complexity becomes NP.

5.1.3 Organization

This chapter is structured as follows:

- Section 5.2 illustrates the main features of STTs using an example;
- Section 5.3 formally defines streaming tree transducers;
- Section 5.4 gives examples of STTs transformations;
- Section 5.5 defines several variants of STTs;
- Section 5.6 shows how STTs behave when considering the input is a string or a ranked tree;
- Section 5.7 shows how STTs behave when the output is a string or a ranked tree;

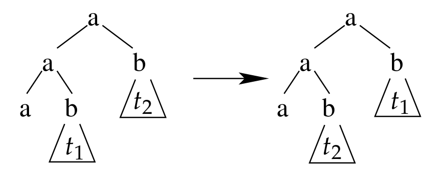


FIGURE 5.1: Depiction of the transformation tree swap.

- Section 5.8 presents the proof that the class of transductions definable using streaming tree transducers is exactly the class of MSO-definable tree transductions;
- Section 5.9 establishes upper bounds for type-checking and functional inequivalence;
- Section 5.10 compares STTs with prior models.

5.2 The features of streaming tree transducers

Before we define streaming tree transducers formally, we introduce the novel features of our model using a high-level example. Figure 5.1 shows a transduction that transforms the input tree by swapping the first (in inorder traversal) b-rooted subtree t_1 with the next (in inorder traversal) b-rooted subtree t_2 not contained in t_1.

An STT performing the transformation depicted in Figure 5.1 processes the nested word left-to-right and uses a stack to record the current depth in the tree. The STT starts in an initial state q_0 that indicates that the transducer has not yet encountered a b-label.

- In state q_0, the STT records the tree traversed so far using a variable x: when reading a *call* $\langle a$, x is stored on the stack, and is reset to ε; when reading a *return* $a\rangle$, x is updated to $x_p\langle a\, x\, a\rangle$ where x_p is the variable stored on the top of the stack. In state q_0, when reading a *call* $\langle b$, the STT pushes q_0 along with the current value of x on the stack, resets x to ε, and updates its state to q'.
- In state q', the STT constructs the first b-labeled subtree t_1 in the variable x: as long as it does not pop the stack symbol q_0, at a call it pushes q' and x, and at a return, updates x to $x_p\langle a\, x\, a\rangle$ or $x_p\langle b\, x\, b\rangle$, depending on whether the current return symbol is a or b. When it pops q_0, it updates x to $\langle b\, x\, b\rangle$ (at this point, x contains the tree t_1, and its value will be propagated), sets another variable y to $x_p\,?$, and changes its state to q_1. The value ? in y is a hole that can be replaced with a value v using the operation $y[v]$.

- In state q_1, the STT is searching for the next b-labeled call, and processes a-labeled calls and returns exactly as in state q_0, but now using a variable y that contains a hole ?. We use $y[s]$ to denote the substitution of the hole ? in y with the value s. At a b-labeled call, it pushes q_1 along with y on the stack, resets x to ε, and updates the state to q'. Now in state q', the STT constructs the second b-labeled subtree t_2 in variable x as before. When it pops q_1, the subtree t_2 corresponds to $\langle b\, x\, b \rangle$. The transducer updates x to $y_p[\langle b\, x\, b \rangle]x_p$ capturing the desired swapping of the two subtrees t_1 and t_2 (the variable y is no longer needed and is reset to ε to ensure the single use restriction), and switches to state q_2.
- In state q_2, the remainder of the tree is traversed, adding it to x. The output function is defined only for the state q_2 and maps q_2 to x.

This example illustrates the main features of STTs: states, variables containing nested words, a stack for storing states and variable values, and holes that can act as place-holders inside variables. These features are formally defined in the next section.

5.3 Model definition

We introduce a few preliminary concepts and then define streaming tree transducers.

5.3.1 Nested words

Data with both linear and hierarchical structure can be encoded using nested words [AM09]. Given a set Σ of symbols, the *tagged alphabet* $\hat{\Sigma}$ consists of the symbols a, $\langle a$, and $a\rangle$, for each $a \in \Sigma$. A *nested word* over Σ is a finite sequence over $\hat{\Sigma}$. For a nested word $a_1 \cdots a_k$, a position j, for $1 \leq j \leq k$, is said to be a *call* position if the symbol a_j is of the form $\langle a$, a *return* position if the symbol a_j is of the form $a\rangle$, and an *internal* position otherwise. The tags induce a natural matching relation between call and return positions, and in this paper, we are interested only in *well-matched* nested words in which all calls/returns have matching returns/calls. A string over Σ is a nested word with only internal positions. Nested words naturally encode ordered trees. The empty tree is encoded by the empty string ε. The tree with a-labeled root with subtrees $t_1, \ldots t_k$ as children, in that order, is encoded by the nested word $\langle a \, \langle\langle t_1 \rangle\rangle \, \cdots \, \langle\langle t_k \rangle\rangle \, a\rangle$, where $\langle\langle t_i \rangle\rangle$ is the encoding of the subtree t_i. This transformation can be viewed as an inorder traversal of the tree. The encoding extends to *forests* also: the encoding of a forest is obtained by concatenating the encodings of the trees it contains. An a-labeled leaf corresponds to the nested word $\langle aa \rangle$, we will use $\langle a \rangle$ as its abbreviation. Thus, a binary tree with a-labeled root for which the left-child is an a-labeled leaf and the right-child is a b-labeled leaf is encoded by the string $\langle a\, \langle a\rangle\, \langle b\rangle\, a\rangle$.

5.3.2 Nested words with holes

A key operation that our transducer model relies on is *insertion* of one nested word within another. In order to define this, we consider nested words with holes, where a hole is represented by the special symbol ?. For example, the nested word $\langle a\, ?\, \langle b\rangle\;\, a\rangle$ represents an incomplete tree with a-labeled root whose right-child is a b-labeled leaf such that the tree can be completed by adding a nested word to the left of this leaf. We require that a nested word can contain at most one hole, and we use two types to keep track of whether a nested word contains a hole or not. A type-0 nested word does not contain any holes, while a type-1 nested word contains exactly one hole. We can view a type-1 nested word as a unary function from nested words to nested words. The set $W_0(\Sigma)$ of type-0 nested words over the alphabet Σ is defined by the grammar

$$W_0 \;:=\; \varepsilon \mid a \mid \langle a\, W_0\, b\rangle \mid W_0\, W_0,$$

for $a, b \in \Sigma$. The set $W_1(\Sigma)$ of type-1 nested words over the alphabet Σ is defined by the grammar

$$W_1 \;:= ? \mid \langle a\, W_1\, b\rangle \mid W_1\, W_0 \mid W_0\, W_1,$$

for $a, b \in \Sigma$. A *nested-word language* over Σ is a subset L of $W_0(\Sigma)$, and a *nested-word transduction* from an input alphabet Σ to an output alphabet Γ is a partial function f from $W_0(\Sigma)$ to $W_0(\Gamma)$.

5.3.3 Nested word expressions

In our transducer model, the machine maintains a set of variables that range over output nested words with holes. Each variable has an associated binary type: a type-k variable has type-k nested words as values, for $k = 0, 1$. The variables are updated using typed expressions, where variables can appear on the right-hand side, and we also allow substitution of the hole symbol by another expression. Formally, a set X of typed variables is a set that is partitioned into two sets X_0 and X_1 corresponding to the type-0 and type-1 variables. Given an alphabet Σ and a set X of typed variables, a *valuation* α is a function that maps X_0 to $W_0(\Sigma)$ and X_1 to $W_1(\Sigma)$. Given an alphabet Σ and a set X of typed variables, we define the sets $E_k(X, \Sigma)$, for $k = 0, 1$, of type-k expressions by the grammars

$$\begin{array}{rcl} E_0 & := & \varepsilon \mid a \mid x_0 \mid \langle a\, E_0\, b\rangle \mid E_0\, E_0 \mid E_1[E_0], \\ E_1 & := & ? \mid x_1 \mid \langle a\, E_1\, b\rangle \mid E_0\, E_1 \mid E_1\, E_0 \mid E_1[E_1] \end{array}$$

where $a, b \in \Sigma$, $x_0 \in X_0$, and $x_1 \in X_1$. The clause $e[e']$ corresponds to substitution of the hole in a type-1 expression e by another expression e'. A valuation α for the variables X naturally extends to a type-consistent function that maps the expressions $E_k(X, \Sigma)$ to values in $W_k(\Sigma)$, for $k = 0, 1$. Given an expression e, $\alpha(e)$ is obtained by replacing each variable x by $\alpha(x)$: in particular, $\alpha(e[e'])$ is obtained by replacing the symbol ? in the type-1 nested word $\alpha(e)$ by the nested word $\alpha(e')$.

5.3.4 Single use restriction

The transducer updates variables X using type-consistent assignments. To achieve the desired tractability, we need to restrict the reuse of variables in right-hand sides. In particular, we want to disallow the assignment $x := xx$ (which would double the length of x), but allow the assignment $(x, y) := (x, x)$, provided the variables x and y are guaranteed not to be combined later. For this purpose, we assume that the set X of variables is equipped with a binary relation η: if $\eta(x, y)$, then x and y cannot be combined. This "conflict" relation is required to be reflexive and symmetric (but need not be transitive). Two conflicting variables cannot occur in the same expression used in the right-hand side of an update or as output. During an update, two conflicting variables can occur in multiple right-hand sides for updating conflicting variables. Thus, the assignment $(x, y) := (\langle a\, xa\rangle[y], a?)$ is allowed, provided $\eta(x, y)$ does not hold; the assignment $(x, y) := (ax[y], y)$ is not allowed; and the assignment $(x, y) := (ax, x[b])$ is allowed, provided $\eta(x, y)$ holds. Formally, given a set X of typed variables with a reflexive symmetric binary conflict relation η, and an alphabet Σ, an expression e in $E(X, \Sigma)$ is said to be *consistent* with η if (1) each variable x occurs at most once in e, and (2) if $\eta(x, y)$ holds, then e does not contain both x and y. Given sets X and Y of typed variables, a conflict relation η, and an alphabet Σ, a *single-use-restricted assignment* is a function ρ that maps each type-k variable x in X to a right-hand side expression in $E_k(Y, \Sigma)$, for $k = 0, 1$, such that (1) each expression $\rho(x)$ is consistent with η, and (2) if $\eta(x, y)$ holds, $\rho(x')$ contains x, and $\rho(y')$ contains y, then $\eta(x', y')$ must hold. The set of such single-use-restricted assignments is denoted $\mathcal{A}(X, Y, \eta, \Sigma)$.

At a return, the transducer assigns the values to its variables X using the values popped from the stack as well as the values returned. For each variable x, we will use x_p to refer to the popped value of x. Thus, each variable x is updated using an expression over the variables $X \cup X_p$. The conflict relation η extends naturally to variables in X_p: $\eta(x_p, y_p)$ holds exactly when $\eta(x, y)$ holds. Then, the update at a return is specified by assignments in $\mathcal{A}(X, X \cup X_p, \eta, \Sigma)$.

When the conflict relation η is the purely reflexive relation $\{(x, x) \mid x \in X\}$, the single-use-restriction means that a variable x can appear at most once in at most one right-hand side. We refer to this special case as "copyless".

5.3.5 Transducer definition

A streaming tree transducer is a deterministic machine that reads the input nested word left-to-right in a single pass. It uses finitely many states, together with a stack. The use of the stack is dictated by the hierarchical structure of the call/return tags in the input. The output is computed using a finite set of typed variables that range over nested words. Variables are equipped with a conflict relation that restricts which variables can be combined. Moreover, the stack can be used to store variable values. At each step, the transducer reads the next symbol of the input. If the symbol is an internal symbol, then the transducer updates its state and the nested-word variables. If the symbol is a call symbol, then the transducer pushes a stack symbol, updates the state, stores updated values of variables in the stack, and reinitializes the variables. While processing a return symbol, the stack is popped, and the new state and new values for the variables are determined using the current state, current variables, popped symbol, and popped variables from the stack. In each type of transition, the variables are updated in parallel using assignments in which the right-hand sides are nested-word expressions. We require that the update is type-consistent and meets the single-use-restriction with respect to the conflict relation. When the transducer consumes the entire input word, the output nested word is produced by an expression that is consistent with the conflict relation. These requirements ensure that, at every step, at most one copy of any value is contributed to the final output.

5.3.6 STT syntax

A *deterministic streaming tree transducer* (STT) § from input alphabet Σ to output alphabet Γ consists of

- a finite set of states Q;
- a finite set of stack symbols P;
- an initial state $q_0 \in Q$;

- a finite set of typed variables X with a reflexive symmetric binary conflict relation η;
- a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that each expression $F(q)$ is consistent with η;[1]
- an internal state-transition function $\delta_i : Q \times \Sigma \mapsto Q$;
- a call state-transition function $\delta_c : Q \times \Sigma \mapsto Q \times P$;
- a return state-transition function $\delta_r : Q \times P \times \Sigma \mapsto Q$;
- an internal variable-update function $\rho_i : Q \times \Sigma \mapsto \mathcal{A}(X, X, \eta, \Gamma)$;
- a call variable-update function $\rho_c : Q \times \Sigma \mapsto \mathcal{A}(X, X, \eta, \Gamma)$;
- a return variable-update function $\rho_r : Q \times P \times \Sigma \mapsto \mathcal{A}(X, X \cup X_p, \eta, \Gamma)$.

5.3.7 STT semantics

To define the semantics of a streaming tree transducer, we consider configurations of the form (q, Λ, α), where $q \in Q$ is a state, α is a type-consistent valuation from variables X to typed nested words over Γ, and Λ is a sequence of pairs (p, β) such that $p \in P$ is a stack symbol and β is a type-consistent valuation from variables in X to typed nested words over Γ. The initial configuration is $(q_0, \varepsilon, \alpha_0)$, where α_0 maps each type-0 variable to ε and each type-1 variable to $?$. The transition function δ over configurations is defined as follows. Given an input $a \in \hat{\Sigma}$:

1. **Internal transitions:** if a is internal, and $\delta_i(q, a) = q'$, then $\delta((q, \Lambda, \alpha), a) = (q', \Lambda, \alpha')$, where
   - q' is the state resulting from applying the internal transition that reads a in state q;
   - the stack Λ remains unchanged;
   - the new evaluation function $\alpha' = \alpha \cdot \rho_i(q, a)$ is the result of applying the variable update function $\rho_i(q, a)$ using the variable values in α.
2. **Call transitions:** if for some $b \in \Sigma$, $a = \langle b$, and $\delta_c(q, b) = (q', p)$, then $\delta((q, \Lambda, \alpha), a) = (q', \Lambda', \alpha_0)$, where

[1] It is possible to relax the definition of the output function to support expressions that are not consistent with the conflict relation η, and this will not affect the expressiveness of the model. We use this restriction to simplify the presentation.

- q' is the state resulting from applying the call transition that reads b in state q;
- $\Lambda' = (p, \alpha \cdot \rho_c(q, b))\Lambda$ is the new stack resulting from pushing the pair (p, α') on top of the old stack Λ, where the stack state p is the one pushed by the call transition function, and $\alpha' = \alpha \cdot \rho_c(q, b)$ is the new evaluation function α' resulting from applying the variable update function $\rho_c(q, b)$ using the variable values in α;
- α_0 is the evaluation function that sets every type-0 variable to ε and every type-1 variable to ?.

3. **Return transitions:** if for some $b \in \Sigma$, $a = b\rangle$, and $\delta_r(q, p, b) = q'$, then $\delta((q, (p, \beta)\Lambda, \alpha), a) = (q', \Lambda', \alpha')$ where

   - q' is the state resulting from applying the return transition that reads b in state q with p on top of the stack;
   - the stack $\Lambda' = \Lambda$ is the result of popping the top of the stack value from the current stack;
   - the new evaluation function $\alpha' = \alpha \cdot \beta_p \cdot \rho_r(q, p, b)$, where β_p is the valuation for variables X_p defined by $\beta_p(x_p) = \beta(x)$ for $x \in X$, is the result of applying the variable update function $\rho_r(q, p, b)$ using the variable values in α, and the stack variable values in β_p.

We define the closure of δ, δ^*, as $\delta^*((q, \Lambda, \alpha), \varepsilon) = (q, \Lambda, \alpha)$ and $\delta^*((q, \Lambda, \alpha), aw) = \delta^*(\delta((q, \Lambda, \alpha), a), w)$. For an input nested word $w \in W_0(\Sigma)$, if $\delta^*((q_0, \varepsilon, \alpha_0), w) = (q, \varepsilon, \alpha)$ then if $F(q)$ is undefined then so is $[\![S]\!](w)$, otherwise $[\![S]\!](w) = \alpha(F(q))$. We say that a nested-word transduction f from input alphabet Σ to output alphabet Γ is *STT-definable* if there exists an STT § such that $[\![S]\!] = f$.

5.4 Examples

Streaming tree transducers can easily implement standard tree-edit operations such as insertion, deletion, and relabeling. We illustrate the interesting features of our model using operations such as reverse and sorting based on a fixed number of tags. In each of these cases, the transducer mirrors the natural algorithm for implementing the desired operation in a single pass.

5.4.1 Reverse

Given a nested word $a_1a_2 \cdots a_k$, its *reverse* is the nested word $b_k \cdots b_2b_1$, where for each $1 \leq j \leq k$, $b_j = a_j$ if a_j is an internal symbol, $b_j = \langle a$ if a_j is a return symbol $a\rangle$, and $b_j = a\rangle$ if a_j is a call symbol $\langle a$. As a tree transformation, *reverse* corresponds to recursively reversing the order of children at each node: the reverse of $\langle a \, \langle b \, \langle d\rangle \, \langle e\rangle \, b\rangle \, \langle c\rangle \, a\rangle$ is $\langle a \, \langle c\rangle \, \langle b \, \langle e\rangle \, \langle d\rangle \, b\rangle \, a\rangle$. This transduction can be implemented by a streaming tree transducer with a single state, a single type-0 variable x, and stack symbols Σ: the internal transition on input a updates x to $a\,x$; the call transition on input a pushes a onto the stack, stores the current value of x on the stack, and resets x to the empty nested word; and the return transition on input b, while popping the symbol a and stack value x_p from the stack, updates x to $\langle b\, x\, a\rangle \, x_p$.

5.4.2 Tag-based sorting

Suppose that given a sequence of trees $t_1t_2 \cdots t_k$ (a forest) and a regular pattern, we want to rearrange the sequence so that all trees that match the pattern appear before the trees that do not match the pattern. For example, given an address book where each entry has a tag that denotes whether the entry is "private" or "public", we want to sort the address book based on this tag: all private entries should appear before public entries, while maintaining the original order for entries with the same tag value. Such a transformation can be implemented naturally using an STT: variable x collects entries that match the pattern, while variable y collects entries that do not match the pattern. As the input is scanned, the state is used to determine whether the current tree t satisfies the pattern; a variable z is used to store the current tree, and once t is read in its entirety, based on whether or not it matches the pattern, the update $\{x := xz, z := \varepsilon\}$ or $\{y := yz, z := \varepsilon\}$ is executed. The output of the transducer is the concatenation xy.

5.4.3 Conditional swap

Suppose that we are given a ternary tree, in which nodes have either three children or none, and the third child of a ternary node is always a leaf. We want to compute the transformation f:

$$f(\langle c_1 x_1 x_2 \langle c_2\rangle c_1\rangle) = \begin{cases} \langle c_1 \, f(x_2) \, f(x_1) \, c_1\rangle, & \text{if } c_2 = a; \\ \langle c_1 \, id(x_2) \, id(x_1) \, c_1\rangle, & \text{if } c_2 = b; \end{cases}$$

$$id(\langle c_1 x_1 x_2 \langle c_2\rangle c_1\rangle) = \langle c_1 \, id(x_1) \, id(x_2) \, c_1\rangle.$$

Informally, while going top-down, f swaps the first two children and deletes the third one. Whenever f finds a node for which the third child is a b, it copies the rest of the tree omitting the third child of each node. The STT implementing f uses four variables x_1, y_1, x_2, y_2. Given a tree $c(t_1, t_2, t_3)$, after finishing processing the first two children t_1 and t_2, the variables x_1 and x_2 respectively contain the trees $f(t_1)$ and $f(t_2)$, and the variables y_1 and y_2 respectively contain the trees $id(t_1)$ and $id(t_2)$. When starting to process the third child t_3, all the variables are stored on the stack. At the corresponding return the variable values are retrieved from the stack (for every $v \in X, v = v^p$), and the state is updated to q_a or q_b representing whether t_3 is labeled with a or b respectively. When processing the return symbol s of a subtree t, we have the following possibilities.

1. The current state is q_a and t is the first child of some node t'. The variables are updated as follows: $x_1 := \langle s\; x_2\; x_1\; s\rangle, x_2 := \varepsilon, y_1 := \langle s\; y_1\; y_2\; s\rangle, y_2 := \varepsilon$.

2. The current state is q_a and t is the second child of some node t'. The variables are updated as follows: $x_1 := x_1^p, x_2 := \langle s\; x_2\; x_1\; s\rangle, y_1 := y_1^p, y_2 := \langle s\; y_1\; y_2\; s\rangle$.

3. The current state is q_b and $t = c(t_1, t_2, t_3)$ is the first child of some node t'. In this case, since t_3 is labeled with b we have $f(t) = \langle s\; t_2\; t_1\; s\rangle$ and $id(t) = \langle s\; t_1\; t_2\; s\rangle$. In order to maintain the invariants that $x_i = f(t_i)$ and $y_i = id(t_i)$, we need to copy the values of y_1 and y_2. The variables are updated as follows: $x_1 := \langle s\; y_2\; y_1\; s\rangle$, $x_2 := \varepsilon, y_1 := \langle s\; y_1\; y_2\; s\rangle, y_2 := \varepsilon$.

4. the current state is q_b and t_2 is the second child of some node t'. Following the same reasoning as for the previous case the variables are updated as follows: $x_1 := x_1^p, x_2 := \langle s\; y_2\; y_1\; s\rangle, y_1 := y_1^p, y_2 := \langle s\; y_1\; y_2\; s\rangle$.

Since y_1 and y_2 can be copied, the conflict relation η is such that $\eta(x_1, y_1)$ and $\eta(x_2, y_2)$ hold.

5.5 Properties and variants

In this section, we note some properties and variants of streaming tree transducers aimed at understanding their expressiveness. First, STTs compute *linearly-bounded* outputs, that is, the length of the output nested word is within at most a constant factor of the length of the input nested word. The single-use restriction ensures that, at every step of the execution of the transducer on an input nested word, the sum of the sizes of all the variables that contribute to the output term at the end of the execution, can increase only by an additive constant.

Proposition 5.1 (Linear-bounded outputs). *For an STT-definable transduction f from Σ to Γ, for all nested words $w \in W_0(\Sigma)$, $|f(w)| = \mathcal{O}(|w|)$.*

Proof. Given an STT-definable transduction f from Σ to Γ, let $A_f = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be an STT defining f. We will show that for every input w, $|[\![A_f]\!](w)| = \mathcal{O}(|A_f|^2|X||w|)$, and since $|A_f|$ and $|X|$ do not depend on w, $|f(w)| = \mathcal{O}(|w|)$.

Given a set $X' \subseteq X$ of variables, we say that X' is non-conflicting iff for all variables x and y in X', it is not the case that $\eta(x, y)$. Given a set of non-conflicting variables X_1 and a variable assignment $v \in \mathcal{A}(X, Y, \eta, \Sigma)$, it is easy to see that the set of variables X_2 appearing in the right-hand sides of v for variables in X_1 are also non-conflicting; if two variables in X_2 were conflicting, then X_1 should also contain two conflicting variables.

We now show that for every word w, if $\delta^*((q_0, \varepsilon, \alpha_0), w) = (q, (p_1, \beta_1) \ldots (p_n, \beta_n), \alpha)$, then for every set X' of non-conflicting variables

$$\sum_{x \in X'} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) = \mathcal{O}(|A_f||X||w|).$$

We proceed by induction on w. The case $w = \varepsilon$ is trivial since every variable has a value of constant size. Now assume $w = w'a$, such that

$$\delta^*((q_0, \varepsilon, \alpha_0), w') = (q, (p_1, \beta_1) \ldots (p_n, \beta_n), \alpha)$$

and by I.H. for every set X_1 of non-conflicting variables

$$\sum_{x \in X_1} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) = \mathcal{O}(|A_f||X||w'|).$$

We have three possible cases:

a is internal: given a set of non-conflicting variables X_1, let X_2 be the set of variables appearing in the right-hand sides of $\rho_i(q, a)$ for variables in X_1. Since both X_1 and X_2 are non-conflicting we know that each variable $x \in X_2$ appears at most once in the right-hand side of X_1. The right hand side also contains $\mathcal{O}(|A_f|)$ symbols in Γ. Let $\alpha' = \alpha \cdot \rho_i(q, a)$, then

$$\sum_{x \in X_1} (|\alpha'(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) \leq \sum_{x \in X_2} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) + \mathcal{O}(|A_f|) =$$

$$\mathcal{O}(|A_f||X||w'| + |A_f|) = \mathcal{O}(|A_f||X||w|).$$

Notice that we can apply the induction hypothesis since X_2 is non-conflicting.

$a = \langle b$ is a call: given a set of non-conflicting variables X_1, let X_2 be the set of variables appearing in the right-hand sides of $\rho_c(q, b)$ for variables in X_1. The right hand side also contains $\mathcal{O}(|A_f|)$ symbols in Γ. Let $\beta_{n+1} = \alpha \cdot \rho_c(q, b)$. Then

$$\sum_{x \in X_1} (|\alpha'(x)| + \sum_{1 \le i \le n+1} |\beta_i(x)|) \le$$

$$\sum_{x \in X_2} (|\alpha_0(x)| + |\alpha(x)| + \sum_{1 \le i \le n} |\beta_i(x)|) + \mathcal{O}(|A_f|) =$$

$$\sum_{x \in X_2} |\alpha_0(x)| + \sum_{x \in X_2} (|\alpha(x)| + \sum_{1 \le i \le n} |\beta_i(x)|) + \mathcal{O}(|A_f|) = \mathcal{O}(|A_f||X||w|).$$

The equality follows since for every x, $|\alpha_0(x)| \le 1$.

$a = b\rangle$ is a return: given a set of non-conflicting variables X_1, let X_2 be the set of variables appearing in the right-hand sides of $\rho_r(q, b, p_n)$ for variables in X_1. Since both X_1 and X_2 is non-conflicting, each variable $x \in X_2$ can appear at most once as x and once as x_p in the right-hand sides of variables in X_1 in $\rho_r(q, b, p_n)$. The right hand side also contains $\mathcal{O}(|A_f|)$ symbols in Γ. Let $\alpha' = \alpha \cdot \beta_n^p \cdot \rho_r(q, a, p_n)$. Then

$$\sum_{x \in X_1} (|\alpha'(x)| + \sum_{1 \le i \le n-1} |\beta_i(x)|) \le$$

$$\sum_{x \in X_2} (\alpha(x) + \beta_n(x) + \sum_{1 \le i \le n-1} \beta_i(x)) + \mathcal{O}(|A_f|) =$$

$$\sum_{x \in X_2} (|\alpha(x)| + \sum_{1 \le i \le n} |\beta_i(x)|) + \mathcal{O}(|A_f|) = \mathcal{O}(|A_f||X||w|).$$

As a consequence, after processing a word w, every variable has a value of size $\mathcal{O}(|A_f||X||w|)$. Since the output function has $\mathcal{O}(|A_f|)$ symbols, the final output has size $\mathcal{O}(|A_f|^2|X||w|)$. This concludes the proof. $\square$

We now examine some of the features in the definition of STTs in terms of how they contribute to expressiveness. First, having multiple variables is essential: this follows from results on streaming string transducers [AC10, AC11]. Consider the transduction that rewrites a nested word w to w^n (that is, w repeated n times). An STT with n variables can implement this transduction. It is easy to prove that an STT with less than n variables cannot implement this transduction. Second, the ability to store symbols in the stack at calls is essential. This is because nested word automata are more expressive than finite state automata over strings.

5.5.1 Regular nested-word languages

A streaming tree transducer with empty sets of string variables can be viewed as an *acceptor* of nested words: the input is accepted if the output function is defined in the terminal state, and rejected otherwise. In this case, the definition coincides with (deterministic) nested word automata (NWA). The original definitions of NWAs and regular nested-word languages do not need the input nested word to be well-matched (that is, the input is a string over $\hat{\Sigma}$), but this distinction is not relevant for our purpose. A nested word automaton A over an input alphabet Σ is specified by a finite set of states Q; a finite set of stack symbols P; an initial state $q_0 \in Q$; a set $F \subseteq Q$ of accepting states; an internal state-transition function $\delta_i : Q \times \Sigma \mapsto Q$; a call state-transition function $\delta_c : Q \times \Sigma \mapsto Q \times P$; and a return state-transition function $\delta_r : Q \times P \times \Sigma \mapsto Q$. A language $L \subseteq W_0(\Sigma)$ of nested words is *regular* if it is accepted by such an automaton. This class includes all regular string and tree languages, and is a subset of deterministic context-free languages [AM09].

Given a nested-word transduction f from input alphabet Σ to output alphabet Γ, the *domain* of f is the set $Dom(f) \subseteq W_0(\Sigma)$ of input nested words w for which $f(w)$ is defined, and the *image* of f is the set $Img(f) \subseteq W_0(\Gamma)$ of output nested words w' such that $w' = f(w)$ for some w. It is easy to establish that, for STT-definable transductions, the domain is a regular language, but the image is not necessarily regular:

Proposition 5.2 (Domain-image regularity)**.** *For an STT-definable transduction f from Σ to Γ, $Dom(f)$ is a regular language of nested words over Σ. There exists an STT-definable transduction f from Σ to Γ, such that $Img(f)$ is not a regular language of nested words over Γ.*

Proof. Given an STT-definable transduction f from Σ to Γ, let $A_f = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be an STT defining it. The NWA A accepting the domain $Dom(f)$ of f has set of states $Q' = Q$, initial state $q'_0 = q_0$, set of stack states $P' = P$, set of final states $F' = \{q \mid F(q) \text{ is defined}\}$, and transition function $\delta' = \delta$.

We now construct an STT B from $\Sigma = \{a, b\}$ to $\Gamma = \{a, b\}$ computing a function f' for which the image $Img(f')$ is not regular. The STT B only has one state q which is also initial and only has transition function $\delta_i(q, a) = \delta_i(q, b) = q$, and has only one type-0 variable x that is updated as follows: $\rho_i(q, a, x) = \delta_i(q, b, x) = axb$. The output function F_B of B is defined as $F_B(q) = x$. The STT B computes the following transduction f': if the input word w has length n, B outputs the word $a^n b^n$. The image $Img(f')$ of f' is the language $\{a^n b^n \mid n \geq 0\}$, which is not a regular language of nested words over Γ. $\square$

It is interesting to observe that the output language might not necessarily be context-free either, since it is easy to build an STT that on an input string w produces the string

ww. We refer the reader to Engelriet et al. [EM02] for an analysis of the output languages of many classes of tree transformations.

5.5.2 Copyless STTs

When the conflict relation η is purely reflexive (i.e. $\{(x, x) \mid x \in X\}$) we call an STT *copyless*. The set of copyless assignments from Y to X is denoted by $\mathcal{A}(X, Y, \Sigma)$ where we drop the relation η. We now define the notion of atomic assignment, which will be fundamental in many proofs.

Definition 5.3. A copyless assignment $\rho \in \mathcal{A}(X, X \cup Y, \Sigma)$ is *atomic* iff it has one of the following forms:

Reset: for some variable $x \in X$, and some $a, b \in \Sigma$, $x := \varepsilon$, $x :=?$, $x := \langle a?b \rangle$, or $x := a$, and for every variable $y \in X$, if $y \neq x$, then $y := y$;

Concatenation: for some two distinct variables $x, y \in X$, $x := xy$ or $x := yx$, $y := \varepsilon$ or $y :=?$, and for every variable $z \in X$, if $z \neq x$ and $z \neq y$, then $z := z$;

Substitution: for some two distinct variables $x, y \in X$, $x := x[y]$ or $x := y[x]$, $y :=?$ or $y := \varepsilon$, and for every variable $z \in X$, if $z \neq x$ and $z \neq y$, then $z := z$;

Swap: for some two distinct variables $x, y \in X$, $x := y$, $y := x$, and for every variable $z \in X$, if $z \neq x$ and $z \neq y$, then $z := z$.

We then show that every copyless assignment can be broken into a sequence of simpler atomic assignments.

Lemma 5.4. *For every copyless assignment $\rho \in \mathcal{A}(X, X, \Sigma)$ there exists a set of variables Y disjoint from X, and a sequence of assignments $s = \rho_1, \ldots, \rho_n$ such that:*

1. *for every variable $x \in X$, $s(x) = \rho(x)$;*
2. *the assignment ρ_1 belongs to $\mathcal{A}(X \cup Y, X, \Sigma)$ and it is atomic;*
3. *for every $2 \leq j \leq n$, the assignment ρ_j belongs to $\mathcal{A}(X \cup Y, X \cup Y, \Sigma)$, and it is atomic.*

Proof. We sketch the proof and gloss over the fact the variables can be of both type-0 and type-1. Given an expression $e \in E = E_0 \cup E_1$ we define the size of e, $\textsc{size}(e)$, as the size of its parse tree:

- if $e \in \{\varepsilon, a, \langle a?b \rangle, ?\}$, then $\textsc{size}(e) = 1$;

- if for some e_1 different from $?$, $e = \langle a e_1 b \rangle$, then $\textsc{size}(e) = 1 + \textsc{size}(e_1)$;
- if for some e_1, e_2, $e = e_1 e_2$, or $e = e_1[e_2]$, then $\textsc{size}(e) = 1 + \textsc{size}(e_1) + \textsc{size}(e_2)$.

Given an assignment $\rho \in \mathcal{A}(X_1, X_2, \Sigma)$, we define $\textsc{size}(\rho) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ to be the value (a, b, c) such that $a = max_{x \in X_1} \textsc{size}(\rho(x))$ is the maximum size of a expression on the right hand side of a variable, b is the number of variables for which the right-hand side is an expression of size a, and c is the size of the set $\{x \mid \exists y . y \neq x \wedge \rho(x) = y\}$. We define the total order between two triplets $(a, b, c), (a', b', c') \in \mathbb{N}^3$ as $(a, b, c) < (a', b', c')$ if $a < a'$, or $a = a'$ and $b < b'$, or $a = a'$, $b = b'$, and $c < c'$.

Given an assignment ρ that is not atomic, we show that ρ can always be transformed into a sequence of atomic assignments $s = \rho_1 \rho_2$, such that $\textsc{size}(\rho_1) < \textsc{size}(\rho)$, ρ_2 is atomic, and for every $x \in X$, $s(x) = \rho(x)$. The new assignments can have new variables. We proceed by case analysis on $size(\rho) = (s_1, s_2, s_3)$.

- If $s_1 = 0$ the assignment is already atomic.
- If $s_1 = 1$ we have two possibilities.
  - If ρ is atomic, then we are done; or
  - $s_3 \geq 1$ and there exist two distinct variables x and y, such that $\rho(x) = y$. Replace ρ with the sequence $\rho_1 \rho_2$, such that $\rho_1(x) = \rho(y)$, $\rho_1(y) = y$, $\rho_2(x) = y$, $\rho_2(y) = x$, and for every z different from x and y, $\rho_1(z) = \rho(z)$, and $\rho_2(z) = z$. The size of ρ_1 is $(s_1, s_2, s_3 - 1)$ which concludes this case.
- If $s_1 > 1$ we have three possibilities.
  - One variable x has right-hand side $e = w_1 w_2$, and e has size s_1. Replace ρ with the sequence $\rho_1 \rho_2$, such that $\rho_1, \rho_2 \in \mathcal{A}(X \cup \{v\}, X \cup \{v\}, \Sigma)$: $\rho_1(x) = w_1$, $\rho_1(v) = w_2$, $\rho_2(x) = xv$, $\rho_2(v) = \varepsilon$, and for each y such that $y \neq x$ and $y \neq n$, $\rho_1(y) = \rho(y)$ and $\rho_2(y) = y$. We then have that ρ_2 is atomic, and since w_1 and w_2 both have smaller size than e, $size(\rho_1)$ is smaller than $size(\rho)$.
  - One variable x has right-hand side $e = w_1[w_2]$, and e has size s_1. Replace ρ with the sequence $\rho_1 \rho_2$ of assignment over $X \cup \{n\}$ such that: $\rho_1(x) = w_1$, $\rho_1(n) = w_2$, $\rho_2(x) = x[n]$, $\rho_2(x) = \varepsilon$, and for each y such that $y \neq x$ and $y \neq n$, $\rho_1(y) = \rho(y)$ and $\rho_2(y) = y$. We then have that ρ_2 is atomic, and since w_1 and w_2 both have smaller size than e, $size(\rho_1)$ is smaller than $size(\rho)$.
  - One variable x has right-hand side $e = \langle a w b \rangle$, e has size s_1, and $w \neq ?$. Replace ρ with the sequence $\rho_1 \rho_2$ of assignment over $X \cup \{n\}$ such that: 1) $\rho_1(x) = \langle a ? b \rangle$, $\rho_1(n) = w$, and for each $y \in X$, if $y \neq x$, $\rho_1(y) = y$, 2)

$\rho_2(x) = x[n]$, $\rho_2(n)\varepsilon$, and for each $y \in X$, such that $y \neq x$, $\rho_3(y) = y$. The assignment ρ_2 is atomic. Since w has smaller size than e, $size(\rho_1)$ is smaller than $size(\rho)$.

It is easy to observe that every atomic assignment over n variables has size $(0, n, 0)$ or $(1, n', k)$ with $n \leq n'$ and $k \leq 2$. We can therefore repeat the procedure we just described until the first assignment is also atomic. This concludes the proof. □

Corollary 5.5. *For every copyless assignment $\rho \in \mathcal{A}(X, X \cup Y, \Sigma)$, with X disjoint from Y, there exists a set of variables Z disjoint from $X \cup Y$ and a sequence of assignments $s = \rho_1, \ldots, \rho_n$ such that:*

1. *for every variable $x \in X$, $s(x) = \rho(x)$;*
2. *the assignment ρ_1 belongs to $\mathcal{A}(X \cup Y \cup Z, X \cup Y, \Sigma)$ and it is atomic;*
3. *for every $2 \leq j \leq n$, the assignment ρ_j belongs to $\mathcal{A}(X \cup Y \cup Z, X \cup Y \cup Z, \Sigma)$ and it is atomic;*
4. *the assignment ρ_n belongs to $\mathcal{A}(X, X \cup Y \cup Z, \Sigma)$ and it is atomic.*

Moreover, if two copyless assignments are composed, the resulting assignment is still copyless.

Lemma 5.6. *Given a copyless assignment $\rho \in \mathcal{A}(Y, X, \Sigma)$, and a copyless assignment $\rho' \in \mathcal{A}(Z, Y, \Sigma)$, the composed assignment $\rho_1 = \rho \cdot \rho' \in \mathcal{A}(Z, X, \Sigma)$ is a copyless assignment in $\mathcal{A}(Z, X, \Sigma)$.*

Proof. Assume this is not the case. Then there exists a variable $x \in X$ that appears twice in the right hand side of ρ_1. This means that there exists two variables $y_1, y_2 \in Y$ appearing in the right hand side of ρ', such that both $\rho(y_1)$ and $\rho(y_2)$ contain x. If $y_1 \neq y_2$, the assignment ρ cannot be copyless, since it would contain two occurrences of x. If $y_1 = y_2$, ρ' cannot be copyless, since it would contain two occurrences of y_1. □

5.5.3 Bottom-up transducers

A nested-word automaton is called *bottom-up* if it resets its state along the call transition: if $\delta_c(q, a) = (q', p)$ then $q' = q_0$. The well-matched nested word sandwiched between a call and its matching return is processed by a bottom-up NWA independent of the outer context. It is known that bottom-up NWAs are as expressive as NWAs over well-matched nested words [AM09]. We show that a similar result holds for transducers also: there is no loss of expressiveness if the STT is disallowed to propagate information at a call to the linear successor. Notice that STTs reinitialize all the variables at every call. An STT S is said to be a *bottom-up STT* if for every state $q \in Q$ and symbol $a \in \Sigma$, if $\delta_c(q, a) = (q', p)$ then $q' = q_0$, and for every variable $x \in X$, $\rho_c(q, a, x) = x$.

Theorem 5.7 (Bottom-up STTs)**.** *Every STT-definable transduction is definable by a bottom-up STT.*

Proof. Let $S = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be an STT. We construct an equivalent bottom-up STT $S' = (Q', q'_0, P', X', \eta', F', \delta', \rho')$. Intuitively, S' delays the application of a call transition of S to the corresponding return. This is done by computing a summary of all possible executions of S on the subword between a call and the corresponding matching return. At the return this summary can be combined with the information stored on the stack to continue the summarization.

Auxiliary notions. Given a nested word $w = a_1 a_2 \ldots a_n$, for each position $1 \leq i \leq n$, let $\textsc{wms}(w, i)$ be the longest well-matched subword $a_j \ldots a_i$ ending at position i. Formally, given a well-matched nested word $w = a_1 a_2 \ldots a_n$, we define $\textsc{wms}(w, i)$ as follows:

- $\textsc{wms}(w, 0) = \varepsilon$;
- if a_i is internal, then $\textsc{wms}(w, i) = \textsc{wms}(w, i-1)a_i$;
- if a_i is a call, then $\textsc{wms}(w, i) = \varepsilon$;
- if a_i is a return with matching call a_j, then $\textsc{wms}(w, i) = \textsc{wms}(w, j-1)a_j a_{j+1} \ldots a_i$.

The nested word $\textsc{wms}(w, i)$ is always well-matched, and represents the subword from the innermost unmatched call position up to position i. For a well-matched nested word w of length n, $\textsc{wms}(w, n)$ equals w. Moreover let $\textsc{lc}(w, i)$ denote the last unmatched call at position i:

- $\textsc{lc}(w, 0) = \perp$ is undefined;
- if a_i is internal, then $\textsc{lc}(w, i) = \textsc{lc}(w, i-1)$;
- if a_i is a call, then $\textsc{lc}(w, i) = i$;
- if a_i is a return, and $\textsc{wms}(w, i) = a_j \ldots a_i$, then $\textsc{lc}(w, i) = \textsc{lc}(w, j-1)$.

State components and invariants. Each state f of Q' is a function from Q to Q. After reading the i-th symbol of w, S' is in state f such that $f(q) = q'$ iff when S processes $\textsc{wms}(w, i)$ starting in state q, it reaches the state q'. The initial state of S' is the identity function f_0 mapping each state $q \in Q$ to itself. A stack state in P' is a pair (f, a) where f is a function mapping Q to Q and a is a symbol in Σ.

Next, we define the transition relation δ'. When reading an internal symbol a, starting in state f, S' goes to state f' such that for each $q \in Q$, $f'(q) = \delta_i(f(q), a)$. When reading a call symbol $\langle a$, starting in state f, S' stores f on the stack along with the symbol a and goes to state f_0. When reading a return symbol $b\rangle$, starting in state f, and with (f', a) on top of the stack, S' goes to state f'' defined as: for each $q \in Q$, if $f'(q) = q_1$ (the state reached by S when reading $\textsc{wms}(w, \textsc{lc}(w, i-1))$ starting in q), $\delta_c(q_1, \langle a) = (q_2, p)$, and $f(q_2) = q_3$ (the state reached by S when reading $\textsc{wms}(w, i-1,)$ starting in q_2), then $f''(q) = \delta_r(q_3, p, b)$.

Variable updates and invariants. We now explain how S' achieves the summarization of the variable updates of S. For each variable $x \in X$ and state $q \in Q$, X' contains a variable x_q. After reading the i-th symbol, x_q contains the value of x computed by S, when reading $\textsc{wms}(w, i)$ starting in state q. Given an assignment $\alpha \in \mathcal{A}(X, X, \Sigma) \cup \mathcal{A}(X, X \cup X_p, \Sigma)$, a state $q \in Q$, and a set of variables $Y \subseteq X \cup X_p$, we define $\textsc{sub}_{q,Y}(\alpha)$ to be the assignment $\alpha' \in \mathcal{A}(X', X', \Sigma) \cup \mathcal{A}(X', X' \cup X'_p, \Sigma)$, where $X' = (X \setminus Y) \cup Y'$ with $Y' = \{y_q \mid y \in Y\}$, and each variable $y \in Y$ is replaced by $y_q \in Y'$.

Initially, and upon every call, each variable x_q is assigned the value ? or ε, if x is a type-1 or type-0 variable respectively. We use $e\{x/x'\}$ to denote the expression e in which every variable x is replaced with the variable x'. When processing the input symbol a, starting in state f, each variable x_q is updated as follows:

a is internal: if $f(q) = q'$, then $x_q := \textsc{sub}_{q,X}(\rho_i(q', a, x))$ is the result of applying the variable update function of S in state q' where each variable $x \in X$ is renamed to x_q;

a is a call: since S' is bottom-up, every variable is simply stored on the stack, and the update function at the call is delayed to the corresponding return, $x_q := x_q$;

$a = b\rangle$ is a return: if (f', c) is the state popped from the stack, $f'(q) = q_1$, $\delta_c(q_1, c) = (q_2, p)$, and $f(q_2) = q_3$, then

$$x_q := \textsc{sub}_{q_2,X}(\textsc{sub}_{q,X_p}(\rho_r(q_3, b, p, x)\{y_p/\rho_c(q_1, c, y)\{z/z_p\}\}))$$

is the result of applying the call variable update function in state q_1, followed by the return variable update function of S in state q' where each variable $x \in X$ is renamed to x_{q_2}, and each variable $x \in X_p$ is renamed to x_q.

Output function. The output function F' of S' is defined as follows: for each state $f \in Q'$, $F'(f) = \text{SUB}_{q_0,X}(F(f(q_0)))$ is the result of applying the output function of S in state $f(q_0)$ where each variable $x \in X$ is renamed to x_{q_0}.

Conflict relation. The conflict relation η' contains the following rules:

1. Variable summarizing different states are in conflict: for all $x, y \in X$, for all $q \neq q' \in Q$, $\eta'(x_q, y_{q'})$;
2. Variables that conflict in S also conflict in S', for every possible summary: for all $q \in Q$, for all $x, y \in X$, if $\eta(x, y)$, then $\eta'(x_q, y_q)$.

Next, we prove that η' is consistent with the update function ρ'. We assume the current state $f \in Q'$ to be fixed. We first show that two conflicting variables never appear in the same right hand side. Each assignment of S' has the form $\text{SUB}_{q,Y}(\rho(q, a, x))$. Therefore if no variables of S are conflicting in $\rho(q, a, x)$ with respect to η, no variables are conflicting in $\text{SUB}_q(\rho(q, a, x))$ with respect to η'. Secondly, we show that for each $x, y, x', y' \in X'$, if $\eta'(x, y)$ holds, x appears in $\rho'(q, x')$, and y appears in $\rho'(q, y')$, then $\eta(x', y')$ holds. From the definition of η', we have that two variables in X' can conflict for one of the following two reasons.

- Two variables $x_{q_1}, y_{q_1'} \in X'$ such that $q_1 \neq q_1'$ appear in two different assignments to w_{q_2} and $z_{q_2'}$ respectively, for some $w, z \in X$ and $q_2, q_2' \in Q$. We need to show $\eta'(w_{q_2}, z_{q_2'})$ and we have two possibilities.
  - If $q_2 = q_2'$, assuming the current symbol is internal, we have that w_{q_2} and z_{q_2} are updated to $\text{SUB}_{q_2}(\rho_i(f(q_2), a, w))$ and $\text{SUB}_{q_2}(\rho_i(f(q_2), a, z))$, where all variables are labeled with q_2. This violates the assumption that $q_1 \neq q_1'$. If the current symbol is a call or a return a similar reasoning holds.
  - If $q_2 \neq q_2'$, $\eta'(w_{q_2}, z_{q_2'})$ follows from the first rule of η'.
- Two variables $x_{q_1}, y_{q_1} \in X'$, such that $\eta(x, y)$, appear in two different assignments to w_{q_2} and $z_{q_2'}$ respectively, for some $w, z \in X$, and $q_2, q_2' \in Q$. We need to show $\eta'(w_{q_2}, z_{q_2'})$. If $q_2 = q_2'$, $\eta'(w_{q_2}, z_{q_2'})$ follows from the second rule of η'. If $q_2 \neq q_2'$, $\eta'(w_{q_2}, z_{q_2'})$ follows from the first rule of η'.

This concludes the proof. □

5.5.4 Regular look-ahead

Now we consider an extension of the STT model in which the transducer can make its decisions based on whether the remaining (well-matched) suffix of the input nested word belongs to a regular language of nested words. Such a test is called *regular look-ahead* (RLA). A key property of the STT model is the closure under regular look-ahead. Furthermore, in the presence of regular-look-ahead, the conflict relation can be trivial, and thus, copyless STTs suffice.

Definition of regular look-ahead. Given a nested word $w = a_1 a_2 \dots a_n$, for each position $1 \leq i \leq n$, let $\text{WMP}(w, i)$ be the longest well-matched subword $a_i \dots a_j$ starting at position i. Formally given a well-matched nested word $w = a_1 a_2 \dots a_n$, $\text{WMP}(w, n + 1) = \varepsilon$, and for each position i such that $1 \leq i \leq n$:

1. if a_i is internal, $\text{WMP}(w, i) = a_i \; \text{WMP}(w, i + 1)$;
2. if a_i is a call with matching return a_j, $\text{WMP}(w, i) = a_i \; \text{WMP}(w, i + 1) a_j \; \text{WMP}(w, j + 1)$;
3. if a_i is a return, $\text{WMP}(w, i) = \varepsilon$.

Given a symbol $a \in \Sigma$, we define the reverse of a tagged symbol as $\text{REV}(a) = a$, $\text{REV}(\langle a) = a\rangle$, and $\text{REV}(a\rangle) = \langle a$. We use $\text{REV}(w) = \text{REV}(a_n) \dots \text{REV}(a_1)$, for the reverse of $w = a_1 \dots a_n$, and $\text{REV}(L) = \{\text{REV}(w) \mid w \in L\}$ for the language of reversed strings in L.

When reading the i-th symbol of w, a look-ahead checks whether a regular property of the nested word $\text{WMP}(w, i)$ holds. Let L be a regular language of nested words, and let A be a (deterministic) bottom-up NWA for $\text{REV}(L)$ (such a NWA exists, since regular languages are closed under the reverse operation [AM09]). Then, while processing a nested word, testing whether the nested word $\text{WMP}(w, i)$ belongs to L corresponds to testing whether the state of A after processing $\text{REV}(\text{WMP}(w, i))$ is an accepting state of A. Since regular languages of nested words are closed under intersection, the state of a single bottom-up NWA A reading the input nested word in reverse can be used to test membership of the well-matched suffix at each step in different languages. Also note that, since A is bottom-up, its state after reading $\text{REV}(\text{WMP}(w, i))$, is the same as its state after reading $\text{REV}(a_i \dots a_n)$. This motivates the following formalization. Let $w = a_1 \dots a_n$ be a nested word over Σ, and let A be a bottom-up NWA with states R processing nested words over Σ. Given a state $r \in R$, we define the (r, A)-look-ahead labeling of w to be the nested word $w_r = r_1 r_2 \dots r_n$ over the alphabet R such that for each position $1 \leq j \leq n$, the call/return/internal type of r_j is the same as the type of a_j, and the corresponding symbol is the state of the NWA A after reading $\text{REV}(a_j \dots a_n)$ starting in state r. Then, the *A-look-ahead labeling of w*, is the nested word $w_A = w_{r_0}$. An *STT-with-regular-look-ahead* (STTR) consists of a bottom-up NWA A over Σ with states R, and an STT S from R to Γ. Such a transducer defines a streaming tree transduction from Σ to Γ: for an input nested word $w \in W(\Sigma)$, the output $[\![S, A]\!](w)$ is defined to be $[\![S]\!](w_A)$.

Closure under regular look-ahead. The critical closure property for STTs is captured by the next theorem, which states that regular look-ahead does not add to the expressiveness of STTs. This closure property is key to establishing that STTs can compute all MSO-definable transductions.

Theorem 5.8 (Closure under regular look-ahead)**.** *The transductions definable by STTs with regular look-ahead are STT-definable.*

Proof. Let A be an NWA with states R, initial state r_0, stack symbols P'', and state-transition function δ''. Let S_A be an STT from R to Γ. We construct an STT $S' = (Q', q_0', P', X', \eta', F', \delta', \rho')$ equivalent to the STTR (S_A, A). Using Theorem 5.7, let $S = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be a bottom-up STT equivalent to S_A. We use few definitions from the proof of Theorem 5.7: 1) $\text{WMS}(w, i)$ is the longest well-matched subword $a_j \dots a_i$ ending at position i; 2) $\text{SUB}_{q,Y}(\alpha)$ is the function that substitutes each variable $x \in Y$ in an assignment α with the variable x_q.

First of all, we observe that for a well-matched nested word w, and an STT S, if $\delta^*((q, \Lambda, \alpha), w) = (q', \Lambda', \alpha')$, then $\Lambda = \Lambda'$, and the value Λ does not influence the execution of S. Hence, for a well-matched nested word w, we can omit the stack component from configurations, and write $\delta^*((q, \alpha), w) = (q', \alpha')$.

State components and invariants. Given the input nested word $w = a_1 \dots a_n$, when processing the symbol a_i, the transition of the STT S depends on the state of A after reading $\text{REV}(\text{WMP}(w, i))$. Since the STT S' cannot determine this value based on the prefix read so far, it needs to simulate S for every possible choice of $r \in R$. We do this by keeping some extra information in the states of S'.

Each state $q' \in Q'$ is a pair (f, g), where $f : R \mapsto R$, and $g : R \mapsto Q$. After reading the input symbol a_i, for every state $r \in R$, $f(r)$ is the state reached by A after reading $\text{REV}(\text{WMS}(w, i))$ when starting in state r, and $g(r)$ is the state reached by S after reading $\text{WMS}(w, i)_r$ starting in state q_0. Recall that w_r is the state labeling produced by A, when reading w starting in state r. The initial state is $q'_0 = (f_0, g_0)$, where, for every $r \in R$, $f_0(r) = r$ and, $g_0(r) = q_0$. Each stack state $p' \in P'$ is a triplet (f, g, a), where the components f and g are the same as for Q', and a is a symbol in Σ.

We now describe the transition relation δ'. We assume S' to be in state $q = (f, g)$, and processing the input symbol a_i. There are three possibilities:

a_i is internal: $\delta'_i(q, a_i) = (f', g')$ where, for each $r \in R$, if $\delta''_i(r, a_i) = r'$

- if $f(r') = r''$, then $f'(r) = r''$; the state reached by A when reading $\text{REV}(\text{WMS}(w, i))$, starting in r, is the same as the state reached when reading $\text{REV}(\text{WMS}(w, i-1))$ starting in r';
- if $g(r') = q$, and $\delta_i(q, r') = q'$, then $g'(r) = q'$; the state reached by S when reading $\text{WMS}(w, i)_r$, is the same as the state it reaches when reading $\text{WMS}(w, i-1)_{r'} r'$;

$a_i = \langle a$ is a call: $\delta'_c(q, a) = (q'_0, p)$, where $p = (f, g, a)$; the current state (f, g) is stored on the stack along with the current symbol a, and the control state is reset to q'_0;

$a_i = a\rangle$ is a return: let $a_j = \langle b$ be the matching call of a_i, and let $p = (f', g', b)$ be the state popped from the stack. Since A is bottom-up for every $r \in R$, $\delta''_c(r, a) = (r_0, p_r)$ for some $p_r \in P''$. Let $f(r_0) = r_1$ be the state reached by A when reading $\text{REV}(\text{WMS}(w, i-1))$ (i.e., the reversed subword sandwiched between the matching call and return) starting in state r_0, and $g(r_0) = q_1$ is the state reached by S after processing $\text{WMS}(w, i-1)_{r_0}$. Finally, $\delta'_r(q, p, a) = (f'', g'')$, where for each $r \in R$, if $\delta''_r(r_1, p_r, b) = r_2$,

- if $f'(r_2) = r'$, then $f''(r) = r'$;
- if $g'(r_2) = q_2$, $\delta'_c(q_2, r_2) = (q_0, p')$, and $\delta'_r(q_1, p', r_0) = q'$, then $g''(r) = q'$.

Variable updates and invariants. The STT S' has variable set $X' = \{x_r \mid x \in X, r \in R\}$. After processing the i-th symbol a_i, the value of x_r is the same as the value of x computed by S after reading $\text{WMS}(w, i)_r$. We can now describe the variable update function ρ'. We assume that S' is in state $q = (f, g)$ and it is processing the symbol a_i. For each variable $x_r \in X'$, S' performs the following update:

a_i is internal: if $\delta''_i(r, a) = r'$, and $g(r') = q'$, then $\rho'(q, a, x_r) = \text{SUB}_{r'}(\rho_i(q', r', x))$ is the assignment of S to x where each variable x is replaced with $x_{r'}$;

$a_i = \langle a$ is a call: S' performs the assignment $\rho'_c(q, b, x_r) = x_r$, where we store all variable values on the stack, and delay the update to the matching return;

$a_i = a\rangle$ is a return: let $a_j = \langle b$ be the corresponding call, and let $p = (f', g', b)$ be the state popped from the stack. The update follows a similar reasoning to that of the transition function δ'. Assume $\delta''_c(r, a) = (r_0, p)$, $f(r_0) = r_1$, $\delta''_r(r_1, p, b) = r_2$, $g(r_0) = q_1$, $g'(r_2) = q_2$, $\delta_c(q_2, r_2) = (q_3, p')$. For each $x \in X$, let $t_c(x)$ be the expression $\rho_c(q_2, r_2, x)$, and $t_r(x)$ be the expression $\rho_r(q_1, r_0, x)$;
now for every $x \in X$, let $t'_c(x) = t_c(x)\{y/y_p\}$ be the expression $t_c(x)$ in which every variable $y \in X$ is replaced with the corresponding stack variable y_p, and let $t''(x) = t_r(x)\{y_p/t'_c(y)\}$ be the expression $t_r(x)$ in which every variable $y_p \in X_p$ is replaced with the expression $t'_c(y)$. The final update will be the expression $\rho'(q, a, x_r) = \text{SUB}_{r_2, X_p}(\text{SUB}_{r', X}(t''(x)))$ where each non stack variable x is replaced with x'_r, and each stack variable y is replaced with y_{r_2}.

Output function. The function output F' only outputs variables labeled with r_0: for every state $(f, g) \in Q'$, if $g(r_0) = q$, then $F'(f, g) = \text{SUB}_{r_0} F(q)$.

Conflict relation. Finally, we define the conflict relation η' as follows:

1. Variables summarizing different look-ahead states are in conflict: for all $x, y \in X$, for all $r_1 \neq r_2 \in R$, then $\eta'(x_{r_1}, y_{r_2})$;

2. Variables that conflict in S also conflict in S' for every possible summary: for all $x, y \in X$, such that $\eta(x, y)$, and for all $r \in R$, $\eta'(x_r, y_r)$.

The proof that ρ' is consistent with η' is analogous to the proof of Theorem 5.7. □

Copyless STTs with RLA. Recall that an STT is said to be copyless if η is the reflexive relation. In an STT, an assignment of the form $(x, y) := (z, z)$ is allowed if x and y are guaranteed not to be combined, and thus, if only one of x and y contributes to the final output. Using regular look-ahead, the STT can check which variables contribute to the final output, avoid redundant updates, and thus be copyless.

Theorem 5.9 (Copyless STT with RLA)**.** *A nested-word transduction f is STT-definable iff it is definable by a copyless STT with regular look-ahead.*

Proof. The proof of the $\Leftarrow$ direction is straightforward: given a copyless STT with regular look-ahead, we can use Theorem 5.8 to construct an equivalent STT.

We now prove the $\Rightarrow$ direction. Let S_1 be an STT from Σ to Γ. Using theorem 5.7, let $S = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be a bottom-up STT, equivalent to S_1. We construct a bottom-up NWA $A = (R, r_0, P'', \delta'')$, and a copyless STT S' from R to Γ, such that $[\![S', A]\!]$ is equivalent to $[\![S]\!]$.

The NWA A keeps in the state information about which variables will contribute to the final output. The STT S' uses such information to update only the variables that will contribute to the final output and reset all the ones that will not. This allows S' to be copyless.

Auxiliary notions. We use the notion $\textsc{wmp}(w, i)$ of longest well-matched subword $a_i \dots a_j$ starting at position i. Given a nested word $w = a_1 \dots a_n$, we also define $\textsc{nr}(w, i)$ to be the position of the first unmatched return in $a_i \dots a_n$. By definition, $\textsc{nr}(w, n+1) = n + 1$, and for each position i such that $1 \leq i \leq n$:

1. if a_i is internal, $\textsc{nr}(w, i) = \textsc{nr}(w, i+1)$;
2. if a_i is a call, with matching return a_j, $\textsc{nr}(w, i) = \textsc{nr}(w, j+1)$;
3. if a_i is a return, $\textsc{nr}(w, i) = i$.

RLA automaton intuition. Since A needs to reset its state at every call, it is not enough to consider the variables appearing in the output function of S as contributing variables. After reading the i-th input symbol in $\textsc{rev}(w)$, the state $r \in R$ of A contains the following information: for every set of variables Y, if Y is the set of relevant variables after reading $\textsc{nr}(w, i)$, r contains the set of variables Y' that must be updated when reading the i-th symbol, in order to have all necessary information to update the variables Y when processing $\textsc{nr}(w, i)$.

Before presenting the construction in detail, we need few more definitions. We define the set of subsets of non-conflicting variables of X as follows: $U_X \overset{\text{def}}{=} \{Y \mid Y \subseteq X \wedge \forall x, y \in Y, \text{if } x \neq y,\ (x, y) \notin \eta\}$. We then enrich it with a special variable x_F that represents the final output: $U_X^F \overset{\text{def}}{=} U_X \cup x_F$. Moreover, given an expression $s \in E(X, \Sigma)$ (i.e. an assignment's right hand side), we use $x \in_a s$ to say that a variable $x \in X$ appears in s.

Given a nested word $w = a_1 \dots a_n$, and an STT S, we define the function $\textsc{cv}_{S,w} : Q \times \{0, \dots, n\} \times U_X^F \mapsto U_X^F$, such that, for every state $q \in Q$, position i, and set of variables Y, $\textsc{cv}_{S,w}(q, i, Y) = Y'$ iff Y' is the set of variables that must be updated by S' when reading a_{i+1} in state q, if the set of relevant variables at $\textsc{nr}(w, i)$ is Y. For every $Y \in U_X^F$, $\textsc{cv}_{S,w}(q, n, Y) = Y$. For every $0 \leq i \leq n-1$ we have the following possibilities.

a_{i+1} *is internal:* if $\delta_i(q, a_{i+1}) = q'$, and $\textsc{cv}_{S,w}(q', i+1, Y_1) = Y_2$, then $\textsc{cv}_{S,w}(q, i, Y_1) = Y_3$ where

- if $Y_2 = x_F$, then $Y_3 = \{x \mid x \in_a F(q')\}$;
- if $Y_2 \neq x_F$ and a_{i+2} is a call $\langle a$, then $Y_3 = \{x \mid \exists y \in Y_2 . x \in_a \rho_c(q', a, y)\}$; Y_3 contains the variables that appear on the right-hand side of the variables Y_2 while reading the symbol a_{i+2};
- if $Y_2 \neq x_F$ and a_{i+2} is internal, then $Y_3 = \{x \mid \exists y \in Y_2 . x \in_a \rho_i(q', a_{i+2}, y)\}$;
- if a_{i+2} is a return, then $Y_3 = Y_2$.

$a_{i+1} = \langle a$ *is a call:* let $a_{j+1} = b\rangle$ be the matching return; if $\delta_c(q, a) = (q_0, p)$, $\delta^*(q_0, (a_{i+2} \dots a_j)) = q_1$, $\delta_r(q_1, a_{j+1}, p) = q_2$, and $\textsc{cv}_{S,w}(q_2, j+1, Y_1) = Y_2$, then $\textsc{cv}_{S,w}(q, i, Y_1) = Y_3$ where

- if $Y_2 = x_F$, then $Y_3 = \{x \mid \exists y \in_a F(q_2) . x^p \in_a \rho_r(q_1, b, p, y)\}$;
- if $Y_2 \neq x_F$, then $Y_3 = \{x \mid \exists y \in Y_2 \ \wedge x^p \in_a \rho_r(q_1, b, p, y)\}$.

a_{i+1} *is a return:* $\textsc{cv}_{S,w}(q, i, Y) = Y$ if $Y \neq x_F$, and undefined otherwise.

Before continuing we prove that the function $\textsc{cv}_{S,w}$ always returns a set of non-conflicting variables.

Lemma 5.10. *For every $i \in \{0, \dots, n-1\}$, $q \in Q$, $Y \in U^F_X$, if $\textsc{cv}_{S,w}(q, i, Y) = Y'$ then $Y' \in U^F_X$.*

Proof. We proceed by induction on i. The base case, $i = n$ and $\textsc{cv}_{S,w}(q, n, Y) = Y$, is trivial. We now have to show that for all $i < n$, if $\textsc{cv}_{S,w}(q, i, Y) = Y'$, then $Y' \in U^F_X$. We assume by induction hypothesis that, for all $q' \in Q$, $Z \in U^F_X$, if $\textsc{cv}_{S,w}(q', i+1, Z) = Z'$ then $Z' \in U^F_X$. We have the following three cases.

a_{i+1} *is internal:* we need to prove that $Y_3 \in U^F_X$. By IH, we know that $Y_2 \in U^F_X$. If $Y_2 = x_F$, $Y_3 = \{x \mid x \in_a F(q')\}$ must be a set non-conflicting for $F(q')$ to be well defined. If $Y_2 \neq x_F$, and $a_{i+2} = \langle a$ is a call, $Y_3 = \{x \mid \exists y \in Y_3 . x \in_a \rho_c(q', a, y))\}$. Let's assume by way of contradiction that there exist $x, y \in Y_2$, such that $\eta(x, y)$ holds. If this is the case there must exist either two variables $x', y' \in Y_3$ such that $x \in_a \rho_c(q', a_{i+2}, x')$ and $y \in_a \rho_c(q', a, y')$, or a variable $x' \in Y_2$ such that $x, y \in_a \rho_c(q', a, x')$. In both cases, using the definition of conflict relation, we can show that the hypothesis that $Y_2 \in U^F_X$ contains only conflict free variables is violated.

a_{i+1} *is a call:* similar to the previous case;

a_{i+1} *is a return:* trivial.

This concludes the proof. □

RLA automaton construction. We now construct the NWA A that computes the function $\text{CV}_{S,w}$. The NWA A mimics the definition of $\text{CV}_{S,w}$ while reading the input nested word backward. At every call (return for the input) the state of A is reset to ensure that the A is bottom-up and the current value of $\text{CV}_{S,w}$ is stored on the stack. At the matching return (call for the input), the value popped from the stack is used to compute the new value of $\text{CV}_{S,w}$.

In order for the construction to work, we also need A to "remember", while reading backward, what is the set of variables that will be relevant at the next call. Given a nested word $w = a_1 \dots a_n$, and a position i, we define $\text{NC}(w,i)$ as the next call in $\text{WMP}(w,i)$. Formally, $\text{NC}(w,n+1) = \bot$, and, for each $1 \leq i \leq n$,

- if a_i is internal, then $\text{NC}(w,i) = \text{NC}(w,i+1)$;
- if a_i is a call, then $\text{NC}(w,i) = i$;
- if a_i is a return, then $\text{NC}(w,i) = \bot$.

Next, we define the set of states R of A. Each state $r \in R$, is a quadruple (s, f, g, h) where $s \in \Sigma \cup \{\bot\}$, $f : Q \times U_X^F \mapsto U_X^F$, $g : Q \mapsto Q$, and $h : Q \times U_X^F \mapsto (U_X^F \cup \bot)$, where f computes the function $\text{CV}_{S,w}(q,i,Y) = Y'$, g summarizes the execution of S on $\text{WMP}(w,i)$, and h computes which variables will be necessary at the return matching the next call, and therefore which variables must be stored on the stack. We formally present the invariants maintained by A and then show its construction. Given the input nested word $w = a_1 \dots a_n$, after processing $\text{REV}(a_i \dots a_n)$, A is in state (s, f, g, h), where:

1. for all $Y \in U_X^F$, and $q \in Q$, if $\text{CV}_{S,w}(q,i,Y) = Y'$, then $f(q,Y) = Y'$;
2. if a_i is a return, then $s = \bot$, otherwise $s = a_i$;
3. if $q' = \delta^*(q, \text{WMP}(w,i))$, then $g(q) = q'$;
4. for all $Y \in U_X^F$, and $q \in Q$, $h(q,Y) = Y_1$, where
   - if $\text{NC}(w,i) = \bot$, then $Y_1 = \bot$,
   - if $\text{NC}(w,i) = i_c$, $a_{i_c} = \langle a$ has matching return $a_{i_r} = b\rangle$, $\delta^*(q, a_i \dots a_{i_c-1}) = q_1$, $\delta_c(q_1, a_{i_c}) = (q_0, p)$, $\delta^*(q_0, \text{WMP}(w, i_c+1)) = q_2$, $\delta_r(q_2, p, a_{i_r}) = q_3$, and $\text{CV}_{S,w}(q_3, i_r, Y) = Y_2$, then

- if $Y_2 = x_F$, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in_a F(q_3).\ x \in_a \rho_r(q_2, b, p, y)\}$;
- if $Y_2 \neq x_F$ and $a_{i_r+1} = \langle c$ is a call, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in Y_2. \exists z \in_a \rho_c(q_3, c, y). x \in_a \rho_r(q_2, b, z)\}$;
- if $Y_2 \neq x_F$ and a_{i_r+1} is internal, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in Y_2. \exists z \in_a \rho_i(q_3, a_{i_r+1}, y). x \in_a \rho_r(q_2, b, z)\}$;
- if $Y_2 \neq x_F$ and $a_{i_r+1} = c\rangle$ is a return, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in_a Y_2.\ x \in_a \rho_r(q_2, a_{i_r}, p, y)\}$.

The initial state of A is $r_0 = (\bot, f_0, g_0, h_0)$, where $f_0(q, Y) = Y$, $g_0(q) = q$ and $h_0(q, Y) = \bot$, for every $q \in Q$, and $Y \in U_X^F$.

Next we define the transition relation δ'' of A, that preserves the invariants presented above. For each $r = (s, f, g, h) \in R$, $a \in \Sigma$, $\delta''(r, a)$ is defined as follows:

a is internal: $\delta_i''(r, a) = (a, f_1, g_1, h_1)$ where for each $q \in Q$, $Y_1 \in U_X^F$, if $\delta_i(q, a) = q'$, then $g_1(q) = g(q')$, $h_1(q, Y_1) = h(q', Y_1)$, and $f_1(q, Y_1) = Y_3$ where

- if $Y_2 = x_F$, then $Y_3 = \{x \mid x \in_a F(q')\}$;
- if $Y_2 \neq x_F$ and $s = \langle c$ is a call, then $Y_3 = \{x \mid \exists y \in Y_2. \exists z \in_a \rho_c(q_3, c, y). x \in_a \rho_r(q_2, b, z))\}$;
- if $Y_2 \neq x_F$ and s is internal, then $Y_3 = \{x \mid \exists y \in Y_2. x \in_a \rho_i(q', s, y)\}$;
- if $s = \bot$, then $Y_3 = Y_2$;

a is a call $\langle b$ (return reading backward): let $r_1 = ((s_1, f_1, g_1, h_1), s)$ be the state popped from the stack, then, $\delta_r''(r, r_1, b) = (b, f_2, g_2, h_2)$, where for each $q \in Q$, $Y \in U_X^F$, if $\delta_c(q, b) = (q_0, p)$, $g(q_0) = q_1$, $\delta_r(q_1, s, p) = q_2$, and $f_1(q_2, Y_1) = Y_2$, then, $g_2(q) = g(q_2)$, $h_2(q, Y_1) = Y_3 \cap X$, and, $f_2(q, Y_1) = \{x \mid x_p \in Y_3\}$, where

- if $Y_2 = x_F$, then $Y_3 = \{x \mid \exists y \in_a F(q_2) \wedge x \in_a \rho_r(q_1, s, p, y)\}$;
- if $Y_2 \neq x_F$ and $s_1 = \langle c$ is a call, then $Y_3 = \{x \mid \exists y \in Y_2.\ \exists z \in_a \rho_c(q_1, s_1, y).\ x \in_a \rho_r(q_2, s, z)\}$;
- if $Y_2 \neq x_F$ and s_1 is internal, then $Y_3 = \{x \mid \exists y \in Y_2.\ \exists z \in_a \rho_c(q_1, s_1, y).\ x \in_a \rho_r(q_2, s, z)\}$;
- if $Y_2 \neq x_F$ and $s_1 = c\rangle$ is a return, then $Y_3 = \{x \mid \exists y \in Y_2.\ x \in_a \rho_r(q_1, s, p, y)\}$;

a is a return $b\rangle$ (call reading backward): $\delta_c''(r, b) = (r_0, (r, b))$.

STT construction. We finally need to define the STT S' from R to Γ. When reading an input symbol in $(a, f, g, h) \in R$, S' uses the information stored in the function f to update only the variables that are relevant to the final output. The set of states of S' is $Q' = Q \times U_X^F$, with initial state $q'_0 = (q_0, x_F)$. When processing the symbol $r_i = (a, f, g, h)$, S' is in state (q, Y) iff S reaches the state q when processing $a_1 \ldots a_{i-1}$, starting in q_0, and the set of relevant variables at the end of $\textsc{wmp}(w, i)$ is Y. Similarly, the set of stack states is $P' = P \times U_X^F$. The set of variables is $X' = X$. The transition function δ' is defined as follows. For each state $(q, Y) \in Q'$, stack state $(p, Y') \in P'$, and symbol $r = (a, f, g, h) \in R$ we have the following possibilities:

a is internal: if $\delta_i(q, a) = q'$, then $\delta'_i((q, Y), r) = (q', Y)$;

a is a call: if $\delta_c(q, a) = (q', p)$, and $h(q, Y) = Y'$, then $\delta'_c((q, Y), r) = (q', (p, Y'))$;

a is a return: if $\delta_r(q, p, a) = q'$, then $\delta'_r((q, Y), (p, Y'), r) = (q', Y')$.

Next, we define the variable update function ρ'. For each state $(q, Y) \in Q'$, stack state $(p, Y') \in P'$, symbol $r = (a, f, g, h) \in R$, variable $x \in X'$;

- if $x \in f(q, Y)$, then we have the following possibilities:

  a is internal: $\rho'_i(q, r, x)$ is the same as $\rho_i(q, a, x)$;

  a is a call: $\rho'_c(q, r, x)$ is the same as $\rho_c(q, a, x)$;

  a is a return: $\rho'_r(q, p, r, x)$ is the same as $\rho_r(q, p, a, x)$;

- if $x \notin f(q, Y)$, then we have the following possibilities:

  a is internal: if x is a type-0 variable then $\rho'_i(q, r, x) = \varepsilon$, otherwise $\rho'_i(q, r, x) = ?$;

  a is a call: if x is a type-0 variable then $\rho'_c(q, r, x) = \varepsilon$, otherwise $\rho'_c(q, r, x) = ?$;

  a is a return: if x is a type-0 variable then $\rho'_r(q, r, p, x) = \varepsilon$, otherwise $\rho'_r(q, r, p, x) = ?$.

Last, the output function F' is the same as F. From the definition of $\textsc{cv}_{S,w}$ we have that S' is copyless, and by construction $[\![S', A]\!]$ is equivalent to $[\![S]\!]$. □

5.5.5 Multi-parameter STTs

In our basic transducer model, the value of each variable can contain at most one hole. Now we generalize this definition to allow a value to contain multiple parameters.

Such a definition can be useful in designing an expressive high-level language for transducers, and it is also used to simplify constructions in later proofs.

We begin by defining nested words with parameters. The set $H(\Sigma, \Pi)$ of *parameterized nested words* over the alphabet Σ using the parameters in Π, is defined by the grammar

$$H := \varepsilon \mid a \mid \pi \mid \langle a\, H\, b\rangle \mid H\, H \qquad \text{for } a, b \in \Sigma \text{ and } \pi \in \Pi.$$

For example, the nested word $\langle a\, \pi_1\, \langle b\rangle\; \pi_2\, a\rangle$ represents an incomplete tree with a-labeled root that has a b-labeled leaf as a child, such that trees can be added to its left as well as right by substituting the parameter symbols π_1 and π_2 with nested words. We can view such a nested word with 2 parameters as a function of arity 2 that takes two well-matched nested words as inputs and returns a well-matched nested word.

In the generalized transducer model, the variables range over parameterized nested words over the output alphabet. Given an alphabet Σ, a set X of variables, and a set Π of parameters, the set $E(\Sigma, X, \Pi)$ of expressions is defined by the grammar

$$E := \varepsilon \mid a \mid \pi \mid x \mid \langle a\, E\, b\rangle \mid E\, E \mid E[\pi \mapsto E] \qquad \text{for } a, b \in \Sigma, x \in X, \text{ and } \pi \in \Pi.$$

A valuation α from X to $H(\Sigma, \Pi)$ naturally extends to a function from the expressions $E(\Sigma, X, \Pi)$ to $H(\Sigma, \Pi)$.

To stay within the class of regular transductions, we need to ensure that each variable is used only once in the final output and each parameter appears only once in the right-hand side at each step. To understand how we enforce single-use restriction on parameters, consider the update $x := xy$ associated with a transition from state q to state q'. To conclude that each parameter can appear at most once in the value of x after the update, we must know that the sets of parameters occurring in the values of x and y before the update are disjoint. To be able to make such an inference statically, we associate, with each state of the transducer, an occurrence-type that limits, for each variable x, the subset of parameters that are allowed to appear in the valuation for x in that state. Formally, given parameters Π and variables X, an *occurrence-type* φ is a function from X to 2^{Π}. A valuation α from X to $H(\Sigma, \Pi)$ is said to be *consistent* with the occurrence-type φ if for every parameter $\pi \in \Pi$ and variable $x \in X$, if $\pi \in \varphi(x)$, then the parametrized nested word $\alpha(x)$ contains exactly one occurrence of the parameter π, and if $\pi \notin \varphi(x)$, then π does not occur in $\alpha(x)$. An occurrence-type from X to Π naturally extends to expressions in $E(\Sigma, X, \Pi)$: for example, for the expression $e_1 e_2$, if the parameter-sets $\varphi(e_1)$ and $\varphi(e_2)$ are disjoint, then $\varphi(e_1 e_2) = \varphi(e_1) \cup \varphi(e_2)$, else the expression $e_1 e_2$ is not consistent with the occurrence-type φ. An occurrence-type φ_X

from variables X to 2^Π is said to be *type-consistent* with an occurrence-type φ_Y from Y to 2^Π and an assignment ρ from Y to X, if for every variable x in X:

- the expression $\rho(x)$ is consistent with the occurrence-type φ_Y, and
- the parameters resulting from performing the assignment to x are consistent: $\varphi_Y(\rho(x)) = \varphi_X(x)$.

Type-consistency ensures that for every valuation α from Y to $H(\Sigma, \Pi)$ consistent with φ, the updated valuation $\alpha \cdot \rho$ from X to $H(\Sigma, \Pi)$ is guaranteed to be consistent with φ'.

Now we can define the transducer model that uses multiple parameters. A *multi-parameter STT* § from input alphabet Σ to output alphabet Γ consists of:

- a finite set of states Q;
- an initial state q_0;
- a set of stack symbols P;
- state-transition functions δ_i, δ_c, and δ_r, that are defined in the same way as for STTs;
- a finite set of typed variables X equipped with a reflexive symmetric binary conflict relation η;
- for each state q, an occurrence-type $\varphi(q) : X \mapsto 2^\Pi$, and for each stack symbol p, an occurrence-type $\varphi(p) : X \mapsto 2^\Pi$;
- a partial output function $F : Q \mapsto E(X, \Gamma, \Pi)$ such that for each state q, the expression $F(q)$ is consistent with η, and $\varphi(q)(F(q))$ is the empty set;
- for each state q and input symbol a, the update function $\rho_i(q, a)$ from variables X to X over Γ is consistent with η and is such that the occurrence-type $\varphi(\delta_i(q, a))$ is type-consistent with the occurrence-type $\varphi(q)$ and the update $\rho_i(q, a)$;
- for each state q and input symbol a, the update function $\rho_c(q, a)$ from variables X to X over Γ is consistent with η and it is such that, if $\delta_c(q, a) = (q', p)$ the occurrence-types $\varphi(p)$ and $\varphi(q')$ are type-consistent with the occurrence-type $\varphi(q)$ and the update $\rho_c(q, a)$;
- for each state q and input symbol a and stack symbol p, the update function $\rho_r(q, p, a)$ from variables $X \cup X_p$ to X over Γ is consistent with η and it is such that the occurrence-type $\varphi(\delta_r(q, p, a))$ is type-consistent with the occurrence-type $\varphi(q)$ and $\varphi(p)$ and the update $\rho_r(q, p, a)$.

We can assume that $\varphi(q_0) = \emptyset$, and therefore all variables are initialized to ε.

Configurations of a multi-parameter STT are of the form (q, Λ, α), where $q \in Q$ is a state, α is a valuation from variables X to $H(\Gamma, \Pi)$ that is consistent with the occurrence-type $\varphi(q)$, and Λ is a sequence of pairs (p, β) such that $p \in P$ is a stack symbol and β is a valuation from variables X to $H(\Gamma, \Pi)$ that is consistent with the occurrence-type $\varphi(p)$. The clauses defining internal, call, and return transitions are the same as for STTs, and the transduction $[\![S]\!]$ is defined as before. In the same way as before we define a copyless multi-parameter STT as a multi-parameter STT with a purely reflexive reflexive conflict relation (i.e.$\eta = \{(x, x) \mid x \in X\}$) .

Now we establish that multiple parameters do not add to expressiveness. We first prove the property for copyless STTs. Then we add regular look-ahead and show and use it to show that the property holds for general STTs.

Theorem 5.11 (Copyless multi-parameter STTs). *A nested-word transduction is definable by a copyless STT iff it is definable by a copyless multi-parameter STT.*

Proof. Given a copyless STT S constructing a multi-parameter copyless STT S' is trivial. The parameter set Π of S' is the singleton $\{?\}$. For every state q of S, there is a corresponding state q in S'. For every type-0 variable x and state q in S, $\varphi(q, x) = \emptyset$ while for every type-1 variable y, $\varphi(q, y) = \{?\}$.

We now prove the other direction of the iff. Let S be a multi-parameter copyless STT with states Q, initial state q_0, stack symbols P, parameters Π with $|\Pi| = k$, variables X with $|X| = n$, occurrence-type φ, output function F, state-transition functions δ_i, δ_c, and δ_r, and variable-update functions ρ_i, ρ_c, and ρ_r. We construct an equivalent copyless STT $S' = (Q', q'_0, P', X', F', \delta', \rho')$.

Variable summarization intuition. We need to simulate the multi-parameter variables using only variables with a single hole. We do this by using multiple variables to represent a single multi-parameter variable, and by maintaining in the state extra information on how to combine them.

The construction maintains a compact representation of every multi-parameter variable. To understand the construction, consider a variable x with value $\langle a\ \langle b\ \pi_1\ b\rangle\ \langle c\rangle\ \langle b\ \pi_2\ b\rangle\ a\rangle$. One possible way to represent x using multiple variables, each with only one parameter in its value, is the following: $x_1 = \langle a?a\rangle$, $x_2 = \langle b?b\rangle\langle c\rangle$, and $x_3 = \langle b?b\rangle$. Next, we need to maintain in the state some information regarding how to combine these three values to reconstruct x. For this purpose, we use a function of the form $f(x_1) = (x_2, x_3), f(x_2) = \pi_1, f(x_3) = \pi_2$, that tells us to replace the ? in x_1

with x_2x_3 and the holes in x_2 and x_3, with π_1 and π_2 respectively. The state will also maintain a function g that remembers the starting variable (the root of the tree): in this case $g(x) = x_1$ means that x_1 is the root of the symbolic tree representing the variable x.

We now formalize this idea. The set of variables X' contains $(2k-1)n$ variables of type-1 and n variables of type-0. When S' is in state q, for every variable $x \in X$,

- if $\varphi(x) \neq \emptyset$, the value of x is represented by $2|\varphi(x)| - 1$ type-1 variables in X';
- if $\varphi(x) = \emptyset$, the value of x is represented by one type-0 variable in X'.

Since $\varphi(x) \leq k$, we can assume that for every variable $x \in X$, there are exactly $2k-1$ type-1 variables and one type-0 variable in S' corresponding to it. We denote this set by $V(x) = \{x_0, x_1, \ldots, x_{2k-1}\}$, where x_0 is the only type-0 variable. Therefore, the STT S' has the set of variables $X' = \bigcup_{x \in X} V(x)$.

State components and invariants. Each state of Q' is a triplet (q, g, f), where $q \in Q$ keeps track of the current state of S, $g : X \mapsto X'$ keeps track of the root of the symbolic tree representing each variable, and $f : X' \mapsto (X' \times X') \cup \Pi \cup \{\varepsilon\} \cup \bot$ maintains information on the symbolic tree representing each variable. Given a variable $x \in X'$, $f(x) = \bot$ means that x is not being used in any symbolic tree.

We now define the unfolding f^* of the function f that, given a variable in $x \in X'$ provides the value in $H(\Sigma, \Pi)$ corresponding to the symbolic tree rooted in x:

- if $f(x) = \varepsilon$, then $f^*(x) = x$;
- if $f(x) = \pi_i$, then $f^*(x) = x[\pi_i]$;
- if $f(x) = (y, z)$, then $f^*(x) = x[f^*(y)f^*(z)]$;
- if $f(x) = \bot$, then $f^*(x) = \bot$.

Our construction maintains the following invariant: at every point in the computation, the value of $f^*(g(x))$ in S' is exactly the same as the value of x in S. We can assume that at the beginning every variable $x \in X'$ has value ε, and we represent this with $g(x) = x_0$ and $f(x_0) = \varepsilon$. For this base case the invariant holds.

Similarly to what we did for STTs (Corollary 5.5), we observe that every assignment can be expressed as a sequence of elementary updates of the following form:

Constant assignment: $x := w$ where w is a constant (w is of the form $a, \pi, x, \langle a\pi b\rangle$);

Concatenation: $\{x := xy; y := \varepsilon\}$ (and similar cases such as $\{x := yx; y := \varepsilon\}$);

Parameter substitution: $\{x := x[\pi \mapsto y]; y := \varepsilon\}$ (and similar cases such as $\{x := y[\pi \mapsto x]; y := \varepsilon\}$);

Swap: $\{x := y; y := x\}$.

Update functions. We now describe at the same time the transition relation δ' and the variable update function ρ' of S'. Consider a state (q, f, g). We call (q', f', g') the target state and we only write the parts that are updated skipping the trivial cases. Every time a variable v is unused we set $f'(v)$ to $\perp$. We show that the state invariant is inductively preserved:

$\{x := w\}$*:* where w is a constant. Similarly to what we showed earlier in the informal description, the content of x can be summarized using $2|\varphi(x)| - 1$ variables.

$\{x := xy; y := \varepsilon\}$*:* in order for the assignment to be well-defined $\varphi(q', x)$ must be the same as $\varphi(q, x) \cup \varphi(q, y)$, and $|\varphi(q', x)| = |\varphi(q, x)| + |\varphi(q, y)| \leq k$. Let's assume wlog that both $\varphi(q, x)$ and $\varphi(q, y)$ are not empty. By IH x and y use $2|\varphi(x)| - 1 + 2|\varphi(y)| - 1 = 2(|\varphi(q, x)| + |\varphi(q, y)|) - 2 \leq 2k - 2$ variables. First we assign each y_i in the tree rooted in $g(y)$ to some unused $x_{i'}$ in $V(x)$. Let $a(y_i) = x_{i'}$ be such a mapping. From the IH we know that at least one type-1 variable $x_j \in V(x)$ is unused. We can use x_j to concatenate the variables summarizing x and y. We can reflect such an update in the tree shape of x as follows: for every $z \in V(x)$,

- if there exists y' such that $a(y') = z$, we copy the summary from y' to z and replace each variable in the summary with the corresponding one in $V(x)$: $f(z) = f(y)\{y'/a(y')\}$, $z = y$, and $y' = \varepsilon$;
- if $z = x_j$ we concatenate the previous summaries of x and y: if $g(x) = x'$, and $g(y) = y'$, and $a(y') = x''$, then $g'(x) = z$, and $f'(z) = x'x''$. Finally the variable z needs to hold the ?: $\rho(z) =?$.

Finally if y_0 is the type-0 variable in $V(y)$, $g'(y) = y'$, $f(y') = \varepsilon$, and $y_0 = \varepsilon$.

$\{x := x[\pi \mapsto y]; y := \varepsilon\}$*:* in order for the assignment to be well-defined $\varphi(q', x)$ must be the same as $(\varphi(q, x) \setminus \{\pi\}) \cup \varphi(q, y)$, and $|\varphi(q', x)| = (|\varphi(q, x)| - 1) + |\varphi(q, y)| \leq k$. Let's assume wlog that both $\varphi(q, x)$ and $\varphi(q, y)$ are not empty. By IH x and y use $2|\varphi(x)| - 1 + 2|\varphi(y)| - 1 = 2(|\varphi(q, x)| + |\varphi(q, y)|) - 2 \leq 2(k + 1) - 2 = 2k$ variables. First we assign each $y_i \neq g(y)$ in the tree rooted in $g(y)$ to some unused $x_{i'}$ in $V(x)$. Let $a(y_i) = x_{i'}$ be such a mapping. So far we used $2k - 1$ variables

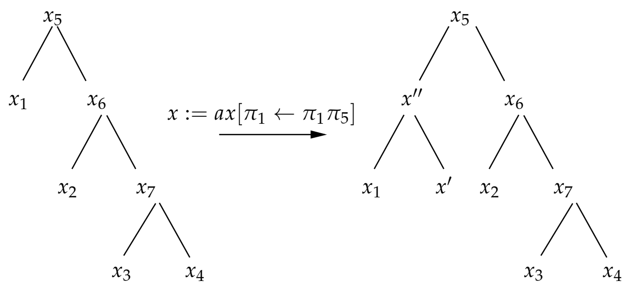


FIGURE 5.2: Example of paramter tree update.

in $V(x)$. When performing the updates we show how the variable representing the root $g(y)$ need not be copied allowing us to use at most $2k-1$ variables to summarize the value of x. The root of the tree summarizing x will be the same as before: if $g'(x) = g(x)$. Every variable $z \in V(x)$ is updated as follows:

- if there exists y such that $a(y) = z$, we copy the summary from y to z and replace each variable in the summary with the corresponding one in $V(x)$: $f(z) = f(y)\{y'/a(y')\}, z = y$, and $y' = \varepsilon$;
- if $z = x_\pi$ we append the summary of y to it: if $g(y) = y'$, then
  - if $f(y') = y_1y_2$, $a(y') = x'$, $a(y_1) = x_1$ and $a(y_2) = x_2$, then $f'(z) = (x_1, x_2)$, and $\rho(z) = z[y']$;
  - if $f(y') = \pi'$, and $a(y') = x'$, then $f'(z) = \pi'$, and $\rho(z) = z[y']$.

Finally if y_0 is the type-0 variable in $V(y)$, $g'(y) = y'$, $f(y') = \varepsilon$, and $y_0 = \varepsilon$.

$\{x := y; y := x\}$: we simply swap the summaries of x and y. Let $a : V(x) \mapsto V(y)$ be a bijection from $V(x)$ to $V(y)$, and let $b : V(y) \mapsto V(x)$ be the inverse of a. Then $g'(x) = a(g(y)), g'(y) = b(g(x), f'(x) = f(y)\{y'/a(y')\}, f'(y) = f(x)\{x'/b(x')\}$, for each $x' \in V(x)$, $x' = a(x')$, and for each $y' \in V(y)$, $y' = b(y')$.

Figure 5.2 shows an example of update involving a combination of elementary updates. The parameter tree on the left shows represents the content $\pi_1\pi_2\pi_3\pi_4$ for a variable x. Therefore, for the tree on the left, $g(x) = x_5$ and $f(x_5) = (x_1, x_6), f(x_6) = (x_2, x_7), f(x_7) = (x_3, x_4), f(x_1) = \pi_1, f(x_2) = \pi_2, f(x_3) = \pi_3, f(x_4) = \pi_4$. Each variable is of type-1. After the update we have $x_5 := ax_5$, and we take two fresh variables x', x'' to update the tree to the one on the right where we set $f(x'') = (x_1, x'), f(x') = \pi_5$. Since we have 5 parameters and 9 nodes, the counting argument still holds. Before the update $f^*(x_5)$ evaluates to $\pi_1\pi_2\pi_3\pi_4$, while after the update $f^*(x_5)$ evaluates to $a\pi_1\pi_5\pi_2\pi_3\pi_4$.

We finally show how δ' and ρ' are defined at calls and returns. The functions maintained in the state are stored on the stack at every call, and this information is used at the corresponding return to create the updated tree. Since all variables are reset at calls, this step is quite straightforward and we omit it. By inspection of the variable update function, it is easy to see that the assignments are still copyless.

Output function. Last, for every state $(q, f, g) \in Q'$, the output function $F'(q, f, g) = f^*(F(q))$, where f^* is naturally extended to sequences: $f^*(ab) = f^*(a)f^*(b)$. □

We can then equip Theorem 5.11 with regular look-ahead and get the following result.

Corollary 5.12 (Copyless multi-parameter STTs RLA)**.** *A nested-word transduction is definable by a copyless STT with regular look-ahead iff it is definable by a copyless multi-parameter STT with regular look-ahead.*

We then extend the result of Theorem 5.9 to multi-parameter STTs.

Lemma 5.13. *A nested-word transduction is definable by a copyless multi-parameter STT with regular look-ahead iff it is definable by a multi-parameter STT.*

Proof. The proof of Theorem 5.9 does not use parameter assignment and can therefore be used for this theorem as well. □

Finally, we can conclude that multi-parameter STTs capture the class of STT definable transformations.

Theorem 5.14 (Multi-parameter STTs)**.** *A nested-word transduction is definable by an STT iff it is definable by a multi-parameter STT.*

Proof. From Theorems 5.13, 5.12, and 5.14. □

5.5.6 Closure under composition

We proceed to show that STTs are closed under sequential composition. Many of our results rely on this crucial closure property.

Theorem 5.15 (Closure under composition)**.** *Given two STT-definable transductions, f_1 from Σ_1 to Σ_2 and f_2 from Σ_2 to Σ_3, the composite transduction $f_2 \cdot f_1$ from Σ_1 to Σ_3 is STT-definable.*

Proof. Using Theorem 5.9, we consider S_1 and S_2 to be copyless STTs with regular look-ahead.

- $S_1 = (Q_1, q_{01}, P_1, X_1, F_1, \delta_1, \rho_1)$ with regular look-ahead automaton A_1, and
- $S_2 = (Q_2, q_{02}, P_2, X_2, F_2, \delta_2, \rho_2)$ with regular look-ahead automaton $A_2 = (R, r_0, P_r, \delta_r)$.

We construct a multi-parameter STT S with regular look-ahead automaton A_1, that is equivalent to S_1 composed with S_2. Finally, we use Theorems 5.8 and 5.14, to remove the parameters and then regular look-ahead, proving that there exists an STT equivalent to S.

Intuition behind the construction. The STT S has to simulate all the possible executions of S_2 on the output of S_1 in a single execution. S keeps in each state a summarization of the possible executions S_2 and uses a larger set of variables to consider all the possible variable values of such executions. At every point in the execution, for every state $q \in Q_2$, and for every variable $x_1 \in X_1$ and $x_2 \in X_2$, the STT S has to remember what would be the value of x_2 if S_2 reads the content of x_1 starting in q. The construction relies on the fact that the content of a variable is a well-matched nested word (with parameters). Thanks to this property, S does not need to collect any information about the stack of S_2.

We show the intuition with a simple example. Let's assume for simplicity that S_1 has only one variable x and S_2 has only one variable y. We also assume that both the look-aheads consist of only one state, and therefore we ignore them. Let's say that at some point in the computation x has value ? and the next input symbol is a. When reading a, S_1 updates x to $ax[?b]$. We need to reflect this update on the variable y of S_2 — i.e. what is the value of y when S_2 reads $ax[?b]$. However, we do not know what is the current state of S_2, and what value S_1 stores in the hole ?. For every possible state q of S_2, and variable x of S_1, the STT S tracks what is the state reached by S_2 after processing the value in x, starting in state q. However, we still need to deal with the unknown value of the hole ?. We can extend the previous idea to solve this problem. Consider the value of x to be $a?b$, where a and b are the nested words respectively before and after the hole. The STT S maintains a function f that, for every two states q_1 and q_2 of S_2, keeps track of the state reached by S_2 when reading a starting in state q_1, and the state reached by S_2 when reading b starting in state q_2 knowing that a was read starting in state q_1. In order to compute the second part of the function, S needs the stack computed by the first one and therefore needs to know that a is processed starting in state q_1.

Next, we describe how we summarize the variable updates of S_2. Again, the update of y depends on the state in which S_2 starts reading the value of x. Similarly to before, we need to deal with the unknown value of the hole ?. However this is not the only problem. Let's assume the variable update function of S_2 is as follows: $\rho_2(q, y, b) = cy$. We want to simulate the execution of S_2, but at this point we do not know what is the previous value of y! We address this issue by treating the old value of y as a parameter. This tells us that the set of parameters contains a parameter x' for every variable in $x \in X_2$. Similarly to what we did for the transition relation, for every two states q_1 and q_2 of S_2, and every variable y of S_1, there is

- a variable $g_1^L(q_1, x, y)$ representing the value of y, when S_2 reads the value of x on the left of ?, starting in state q_1;
- a variable $g_1^R(q_1, q_2, x, y)$ representing the value of y, when S_2 reads the value of x on the right of ?, starting in state q_2, assuming that the value of y on the left of ? was read by S_2 starting in state q_1.

Both these variables at the beginning are set to y', a parameter that represents the value of y before processing the current input. The updates then mimic the transition relation. For example, for the case in which $\rho_2(q, y, b) = cy$, the value of $g_1^R(q', q, x, y)$ is set to cy'.

Since S_2 itself uses type-1 variables, we use the parameter ? for such variable values. Let's analyze this case in detail. When ? directly appears in the g representation of a variable, we can treat it as a normal parameter. The problem occurs in the following case: let's say at a particular step $g(q, x, y) = y'$ but y is a type-1 variable. This can only mean that the ? appears in y'. Now let's assume that the next update is of the form $y := y[a]$. As we can see, we still do not have the ? appearing in the representation of y. We record this fact with a function and delay the substitution using an extra variable for the parameters. As an example, suppose that at some point the values of x and y, both of type 1, are x', y'. We use the variables $x_? =?$ and $y_? =?$ to represent their parameters. Then, after processing a well-matched subword, we may have an update of this form $x := ax[cy[a?c]]b$ and $y := a?$. Notice that the reflexivity of η ensures that x' and y' can appear at most once in the valuation of a variable at any point. This configuration is captured by $x := axb$, $x_? = cy$, $y_? = a?c$ and $y = a?$. In addition we need to keep information about where the actual parameter of every variable is. Let's consider the case in which we are trying to summarize a type-0 variable y of S_2. We keep in the state a function p_0, such that $p_0(q, x, y) = \varepsilon$ if the ? appears in $g_0(q, x, y)$, while $p_0(q, x, y) = xy$, if, for example, in order to substitute the value of ? with the value v ($x := x[v]$), we need to perform the following update, where we omit the state q and the variable x for readability:

$$x := x[x' \mapsto x_?[y' \mapsto y_?[? \mapsto v]]].$$

Finally, we need to summarize the possible look-ahead values of S_2 when reading the variable contents. For example if a variable x of S_1 contains the value $s?t$, we need to know, for every r_1 and r_2 in R, what state would A_2 reach when reading s and t backward. We use two functions l_1^L and l_1^R, such that $l_1^R(r_2, x) = r_2'$ and $l_1^L(r_1, r_2, x) = r_1'$, iff $\delta_R^*(r_2, \text{REV}(t)) = (r_2', \Lambda)$, and $\delta_R^*(r_1, \Lambda, \text{REV}(s)) = r_1'$.

State components and invariants. We denote with $X_{i,j}$ the set of type-j variables in X_i. Each state of Q is a tuple $(q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$, where

- $q \in Q_1$ keeps track of the current state of S_1;
- $f_0 : Q_2 \times R \times X_{1,0} \mapsto Q_2$ is the summarization function for type-0 variable;
- $f_1^L : Q_2 \times R \times R \times X_{1,1} \mapsto Q_2$, and $f_1^R : Q_2 \times Q_2 \times R \times R \times X_{1,1} \mapsto Q_2$ are the summarization functions for type-1 variables;
- $l_0 : R \times X_{1,0} \mapsto R$ is the look-ahead summarization function for type-0 variable;
- $l_1^L : R \times R \times X_{1,1} \mapsto R$, and $l_1^R : R \times X_{1,1} \mapsto R$ are the look-ahead summarization functions for type-1 variables;
- $p_0 : Q_2 \times R \times X_{1,0} \times X_{2,1} \mapsto X_{2,1}^*$ is the function keeping track of the ? for type-0 variables;
- $p_1^L : Q_2 \times R \times R \times X_{1,1} \times X_{2,1} \mapsto X_{2,1}^*$, and $p_1^R : Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_{2,1} \mapsto X_{2,1}^*$ are the function keeping track of the ? for type-1 variables.

We first describe the invariants that S maintains for the first 7 components: given an input nested word $w = a_1 \dots a_n$, after reading the symbol a_i, S is in state $(q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, \_, \_, \_)$.

- The component q is the state reached by S_1 on the prefix $a_1 \dots a_i$, $\delta_1^*(q_{01}, a_1 \dots a_i) = q$.
- Given $q_1 \in Q_2$, and $r \in R$, if x contains the value $s \in W_0(\Sigma_2)$, and $\delta_2^*(q_1, s_r) = q_1'$, then $f_0(q_1, r, x) = q_1'$.
- Given $q_1, q_2 \in Q_2$, and $r_1, r_2 \in Q_2$, if x contains the value $s?t \in W_1(\Sigma_2)$, $\delta_2^*(q_1, s_{r_1,\Lambda_{r_2,t}}) = (q_1', \Lambda)$, and $\delta_2^*(q_2, \Lambda, t_{r_2}) = q_2'$, then $f_1^L(q_1, r_1, r_2, x) = q_1'$, and $f_1^R(q_1, q_2, r_1, r_2, x) = q_2'$. We use the notation $s_{r_1,\Lambda_{r_2,t}}$ to denote the run on s of A_2 starting in state r_1 assuming that A_2 has an initial stack value computed by A_2 on t starting in state r_2.

- Given $r_1 \in R$, if x contains the value $s \in W_0(\Sigma_2)$, and $\delta_2^*(r_1, \text{REV}(s)) = r_1'$, then $l_0(r_1, x) = r_1'$.
- Given $r_1, r_2 \in R$, if x contains the value $s?t \in W_1(\Sigma_2)$, $\delta_2^*(r_2, \text{REV}(t)) = (r_2', \Lambda)$, and $\delta_2^*(r_1, \Lambda, \text{REV}(s)) = r_1'$, then $l_1^L(r_1, r_2, x) = r_1'$, and $l_1^R(r_2, x) = r_2'$.

State transition function. We now show how we maintain the invariants defined above at every update. We first investigate the update of all the components different from p_0, p_1^L, and p_1^R. Let's assume S is in state $(q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$. We are only going to write the parts that are updated, and as before we only consider atomic updates of S_1. We analyze the type-1 case (the 0 case is easier). At every step we indicate with a prime sign the updated components.

$\{x := w\}$: we consider the case where $w = \langle a?b \rangle$ (the other cases are similar). For each q_1 and q_2 in Q_2, r_1, r_2 in R, if $\delta_R(r_2, \langle b) = (r_2', p)$ for some $p \in P_r$, $\delta_R(r_1, p, a\rangle) = r_1'$, $\delta_2^*(q_1, r_1') = (q_1', \Lambda)$, and $\delta_2^*(q_2, \Lambda, r_2') = q_2'$, then the new state has the following components: $f_1^{L'}(q_1, r_1, r_2, x) = q_1'$, $f_1^{R'}(q_1, q_2, r_1, r_2, x) = q_2'$, $l_1^{L'}(r_1, r_2, x) = r_1'$, and $l_1^{R'}(r_2, x) = r_2'$.

$\{x := xy; y := \varepsilon\}$: without loss of generality, let y be a type-0 variable, and x be a type-1 variable. For each q_l and q_r in Q_2, r_l, r_r in R, if $l_0(r_1, y) = r_1$, $l_1^R(r_1, x) = r_2$, $l_1^L(r_l, r_1, x) = r_3$, $f_1^L(q_l, r_l, r_1, x) = q_1$, $f_1^R(q_r, r_l, r_1, x) = q_2$, and $f_0(q_2, r_r, y) = q_3$, then for every $q \in Q_2$, and $r \in R$, the new state has the following components: $l_1^{L'}(r_l, r_r, x) = r_3$, $l_1^{R'}(r_r, x) = r_2$, $l_0'(r, y) = r$, $f_1^{L'}(q_l, r_l, r_r, x) = q_1$, $f_1^{R'}(q_l, q_r, r_l, r_r, x) = q_3$, and $f_0'(q, r, y) = q$.

$\{x := x[y]; y :=?\}$: let x and y be type-1 variables (the other cases are simpler). We need to "synchronize" the left and right parts to update the function f. For each q_l and q_r in Q_2, r_l, r_r in R, assume $l_1^R(r_r, x) = r_1$, $l_1^R(r_1, y) = r_2$, $l_1^L(r_l, r_1, y) = r_3$, $l_1^L(r_3, r_r, z) = r_4$, and $f_1^L(q_l, r_3, r_r, x) = q_1$, $f_1^L(q_1, r_l, r_1, y) = q_2$, $f_1^R(q_1, q_r, r_l, r_1, y) = q_3$, and $f_1^R(q_l, q_3, r_3, r_r, x) = q_4$. For every $q, q' \in Q_2$, and $r, r' \in R$, the new state has the following components: $l_1^{L'}(r_l, r_r, x) = r_4$, $l_1^{R'}(r_r, x) = r_2$, $l_1^{L'}(r, r', y) = r$, $l_1^{R'}(r, y) = r$, $f_1^{L'}(q_l, r_l, r_r, x) = q_2$, $f_1^{R'}(q_l, q_r, r_l, r_r, x) = q_4$, $f_1^{L'}(q, r, r', y) = q$, and $f_1^{R'}(q, q', r, r', y) = q'$.

$\{x := y; y := x\}$: every component involving x is swapped with the corresponding component involving y.

Variable summarization. Similarly to what we did for state summarization, S will have variables of the form $g_0(q, r, x, y)$ with the following meaning: if x contains a nested word w, $g_0(q,r,x,y)$ is the value contained in $y \in X_2$ after reading w_r starting in state q. As we described earlier, there will also be variables of the form $g_0(q,r,x,y)?$ representing the value of the parameter of y. The set of variables X of S described by the union of the following sets:

- $\{g_0(q,r,x,y) \mid (q,r,x,y) \in Q_2 \times R \times X_{1,0} \times X_2\}$;
- $\{g_0(q,r,x,y)? \mid (q,r,x,y) \in Q_2 \times R \times X_{1,0} \times X_{2,1}\}$;
- $\{g_1^L(q,r_1,r_2,x,y) \mid (q,r_1,r_2,x,y) \in Q_2 \times R \times R \times X_{1,1} \times X_2\}$;
- $\{g_1^L(q,r_1,r_2,x,y)? \mid (q,r_1,r_2,x,y) \in Q_2 \times R \times R \times X_{1,1} \times X_{2,1}\}$;
- $\{g_1^R(q,q',r_1,r_2,x,y) \mid (q,q',r_1,r_2,x,y) \in Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_2\}$;
- $\{g_1^R(q,q',r_1,r_2,x,y)? \mid (q,q',r_1,r_2,x,y) \in Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_{2,1}\}$.

Given a nested word w, and a stack Λ, we use $w_{r,\Lambda}$ to denote the regular look-ahead labeling of w when processing $\text{REV}(w)$ in the starting configuration (r,Λ). For every $q_1 \in Q_2$, $r_1, r_2 \in R$, $x \in X_{1,1}$, and $y \in X_2$, if x contains a nested word $v?w$, $\delta_r^*(r_2, w) = (r_2', \Lambda_{r_2})$, and $\delta_2^*(q_1, v_{r_1,\Lambda_{r_2}}) = (q_2, \Lambda_{q_2})$, then:

- $g_1^L(q_1,r_1,r_2,x,y)$ is the variable representing the value of y, after S_2 reads $v_{r_1,\Lambda_{r_2}}$ starting in state q_1;
- $g_1^R(q_1,q_2,r_1,r_2,x,y)$ is the variable representing the value of y, after S_2 reads w_{r_2} starting in the configuration (q_2, Λ_{q_2}).

The parameters of S are used to represent the values of the variables in X_2 when starting reading the values of a variable X_1. At this point we do not know what the values of the variables in X_2 are and for every variable $x \in X_2$, we use a parameter x' to represent the value of x before reading the value in X_1. The STT S has set of parameters $\Pi = \{x' | x \in X_2\} \cup \{?\}$.

At any point in the execution the variable values and the state of S will be related by the following invariant: for any starting valuation α of the variables in X_2, state $q \in Q_2$, look-ahead state $r \in R$, variable $y \in X_2$, and variable $x \in X_1$ with value w, the value of y after reading w_r with initial configuration α can be retrieved from the variables in X and the state components p_0, p_1^L, p_1^R.

Given a nested word $w = a_1 \dots a_n$, we consider the configuration of S and S_1 right after processing the symbol a_i. Let's call $(q_1, \Lambda_1, \alpha_1)$ the current configuration of S_1, and (q_S, Λ, α), with $q_S = (q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$, the current configuration of

S. For every two states $q_2, q_2' \in Q_2$, look-ahead states $r, r' \in R$, variables $x_0 \in X_{1,0}$, $x_1 \in X_{1,1}$, $y_0 \in X_{2,0}$, and $y_1 \in X_{2,1}$ we have the following possibilities.

x_0, y_0: let $\alpha_1(x_0) = s^{x_0}$ be the current valuation of x_0 in S_1, $\alpha(g_0(q_2, r, x_0, y_0)) = t$ be the current valuation of $g_0(q_2, r, x_0, y_0)$ in S, and $\{v_1', \dots, v_k'\} \subseteq \Pi$ be the set of parameters in $\varphi(g_0(q_2, r, x_0, y_0))$; for every valuation α_2 over X_2, if $\delta^*((q_2, \alpha_2), s_r^{x_0}) = (q_3, \alpha_2')$, then:

$$\alpha_2'(y_0) = t[v_1' \mapsto \alpha_2(v_1)] \dots [v_k' \mapsto \alpha_2(v_k)].$$

x_0, y_1: let $\alpha_1(x_0) = s^{x_0}$ be the current valuation of x_0 in S_1, $\alpha(g_0(q_2, r, x_0, y_1)) = t$ be the current valuation of $g_0(q_2, r, x_0, y_1)$ in S, $p_0(q_2, r, x_0, y_1) = z_1 \dots z_i$ be the sequence of variables to follow to reach the hole ? in the representation of y_1, and $\{v_1', \dots, v_k'\} \subseteq \Pi$ be the set of parameters in

$$(\varphi(g_0(q_2, r, x_0, y_1)) \cup \varphi(g_0(q_2, r, x_0, z_1)?) \cup \dots \cup \varphi(g_0(q_2, r, x_0, z_i)?)) \setminus \{z_1, \dots, z_i\}$$

then, for every valuation α_2 over X_2, if $\delta^*((q_2, \alpha_2), s_r^{x_0}) = (q_3, \alpha_2')$, then:

$$\alpha_2'(y_1) = t[z_1' \mapsto [g_0(q_2, x_0, z_1)?[\dots[z_i' \mapsto g_0(q_2, x_0, z_i)?]]]]) \\ [v_1' \mapsto \alpha_2(v_1)] \dots [v_k' \mapsto \alpha_2(v_k)].$$

x_1, y_0: let $\alpha_1(x_1) = s^{x_1^L}?s^{x_1^R}$ be the current valuation of x_1 in S_1, $\alpha(g_1^L(q_2, r, r', x_1, y_0)) = t_L$ be the current valuation of $g_1^L(q_2, r, r', x_1, y_0)$ in S, $\alpha(g_1^R(q_2, q_2', r, r', x_1, y_0)) = t_R$ be the current valuation of $g_1^R(q_2, q_2', r, r', x_1, y_0)$ in S, let $\{v_1^{L'}, \dots, v_k^{L'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^L(q_2, r, r', x_1, y_0))$, and let $\{v_1^{R'}, \dots, v_l^{r'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^R(q_2, q_2', r, r', x_1, y_0))$; for every valuations α_2, α_2' over X_2, if $\delta^*((q_2, \alpha_2), s^{x_1^L}_{r, \Lambda_r', s^{x_1}, R}) = (q_3, \Lambda_2, \alpha_3)$, and $\delta^*((q_2', \Lambda_2, \alpha_2'), s^{x_1^R}_{r'}) = (q_3', \alpha_3')$, then $\alpha_3(y_0) = t_L[v_1^{L'} \mapsto \alpha_2(v_1^L)] \dots [v_k^{L'} \mapsto \alpha_2(v_k^L)]$, and $\alpha_3'(y_0) = t_R[v_1^{R'} \mapsto \alpha_2(v_1^R)] \dots [v_l^{R'} \mapsto \alpha_2(v_l^R)]$.

x_1, y_1: let $\alpha_1(x_1) = s^{x_1^L}?s^{x_1^R}$ be the current valuation of x_1 in S_1, $\alpha(g_1^L(q_2, r, r', x_1, y_1)) = t_L$ be the current valuation of $g_1^L(q_2, r, r', x_1, y_1)$ in S, $\alpha(g_1^R(q_2, q_2', r, r', x_1, y_1)) = t_R$ be the current valuation of $g_1^R(q_2, q_2', r, r', x_1, y_1)$ in S, $p_1^L(q_2, r, r', x_1, y_1) = z_1^L \dots z_i^L$ be the sequence of variables to follow to reach the hole ? in the representation of y_1 on the right of the ?, $p_1^R(q_2, q_2', r, r', x_1, y_1) = z_1^R \dots z_j^R$ be the sequence of variables to follow to reach the hole ? in the representation of y_1 on the right of the ?, $\{v_1^{L'}, \dots, v_k^{L'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^L(q_2, r, r', x_1, y_1))$, and $\{v_1^{R'}, \dots, v_l^{R'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^R(q_2, q_2', r, r', x_1, y_1))$; for every valuations α_2, α_2' over X_2, if $\delta^*((q_2, \alpha_2), s^{x_1^L}_{r, \Lambda_r', s^{x_1^R}}) = (q_3, \Lambda_2, \alpha_3)$, and

$\delta^*((q_2', \Lambda_2, \alpha_2'), s_{r'}^{x_1,R}) = (q_3', \alpha_3')$, then

$$\alpha_3(y_1) = t_L[z_1^{L'} \mapsto [g_1^L(q_2,r,r',x_1,z_1^L)?[\dots[z_i^{L'} \mapsto g_1^L(q_2,r,r',x_1,z_i^L)?]]]])$$
$$[v_1^{L'} \mapsto \alpha_2(v_1^L)]\dots[v_k^{L'} \mapsto \alpha_2(v_k^L)]$$

$$\alpha_3'(y_1) = t_R[z_1^{R'} \mapsto [g_1^R(q_2,q_2',r,r',x_1,z_1^R)?[\dots[z_i^{R'} \mapsto g_1^R(q_2,q_2',r,r',x_1,z_i^R)?]]]])$$
$$[v_1^{R'} \mapsto \alpha_2(v_1^R)]\dots[v_k^{R'} \mapsto \alpha_2(v_k^R)].$$

Variable update function. We assume that S is reading the symbol a, starting in state $q_S = (q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$. We only describe the components that are updated, and assume w.l.o.g that $x_0, x_1 \in X_{1,0}$ are type-0 variable, and $x_2 \in X_{2,1}$ is a type-1 variable. We assume the occurrence-type function $\varphi : Q \times X \mapsto 2^\Pi$ to be well defined according to the following assignments (we will prove consistency later).

$\{x_0 := w\}$: where without loss of generality w is a constant without a ?. We fix the variable components to be the state $q \in Q_2$, look-ahead state $r \in R$, and we are summarizing the values of $y_0 \in X_{2,0}$ and $y_1, y_2 \in X_{2,1}$, when reading the value in $x_0 \in X_{1,0}$. We consider the variable updates performed by S_2 when reading the w, and assume $u = u_1 \dots u_n$ to be the sequence of atomic updates (using Theorem 5.4) performed by S_2 when reading w_r starting in state q. We now provide the updates of S corresponding to the updates in u. At the beginning of the procedure $g_0(q, r, x_0, y_1) = y_1'$, and $p_0(q, r, x_0, y_1) = y_1$.

We denote with a prime the new state values and we only write the parts that are updated. Let's assume the atomic update is of the following form:

$\{y_1 := w\}$: where $w = \langle a?b \rangle$ (the other cases are similar). We have $p_0'(q, r, x_0, y_1) = \varepsilon$, and we define the current assignment to be $g_0(q, r, x_0, y_1) := w$.

$\{y_0 := \varepsilon; y_1 := y_0 y_1\}$: $p_0'(q, r, x_0, y_1) = p_0(q, r, x_0, y_1)$, and we define the current assignment to be $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_0) g_0(q, r, x_0, y_1)$, and $g_0(q, r, x_0, y_0) = \varepsilon$.

$\{y_1 := y_1[y_2]; y_2 := ?\}$: the summary p_0 is updated as $p_0'(q, r, x_0, y_1) = p_0(q, r, x_0, y_1) p_0(q, r, x_0, y_2)$, $p_0(q, r, x_0, y_2) = \varepsilon$; if $p_0(q, r, x_0, y_1) = v_1 \dots v_k$, then we define the current assignment to be

- if $k = 0$, then
$g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_1)[g_0(q, r, x_0, y_2)]$, and $g_0(q, r, x_0, y_2) = ?$;
- if $k > 0$, then $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_1)$,
$g_0(q, r, x_0, v_k?) := g_0(q, r, x_0, v_k?)[g_0(q, r, x_0, y_2)]$, and
$g_0(q, r, x_0, y_2) = ?$.

$\{y_1 := y_2; y_2 := y_1\}$: every component involving y_1 is swapped with the corresponding component involving y_2.

The final variable update is the result of composing all the atomic updates. As shown in Lemma 5.6, the composition of copyless updates is itself copyless.

$\{x_0 := x_0 x_1; x_1 := \varepsilon\}$: we need to substitute the values of the variables after reading x_0 in the corresponding parameters in x_1 in order to simulate the concatenation. Informally (we omit the states for readability) if $g_0(x_1, y)$ contains the value az', and $g_0(x_0, z)$ contains the value bw', the new summarization $g'_0(x_0, y)$ will contain the value abw' where the parameter z' is being replaced with the corresponding variable values.

For every state $q_2 \in Q_2$, look-ahead state $r \in R$, variable $y_1 \in X_{2,1}$, if $l_0(r, x_1) = r_1$, $f_0(q_2, r_1, x_0) = q_3$, $\varphi(q_S, g_0(q_3, r, x_1, y_1)) = \{v'_1, \ldots, v'_k\}$, and $p_0(q_2, r, x_1, y_1) = b_1 \ldots b_{k'}$, then $p_0{}'(q_2, r, x_0, y_1) := p_0(q_2, r_1, x_0, b_1) \ldots p_0(q_2, r_1, x_0, b_{k'})$, and $p_0{}'(q_2, r, x_1, y_1) := y_1$. The next step is collapsing all the parameters chains that can now be resolved with proper parameter substitution. For each assignment to a variable $g'_0(q_2, r_1, x_0, v)$ we omit, unless interesting, the fact that every parameter v' is replaced with the corresponding variable value $g_0(q_2, r_1, x_0, v')$, and we iterate the following procedure starting with the sequence $P = b_1 \ldots b_{k'}$, and the variable $v = y_1$.

1. let p_i be the first element in $P = p_1 \ldots p_n$ such that $p_0{}'(q_2, r, x_0, p_i) \neq \varepsilon$;
2. perform the following assignment

$$\begin{aligned} g_0{}'(q_2, r, x_0, v) := g_0(q_3, r, x_1, v)[\\ p'_1 \mapsto g_0(q_3, r_1, x_0, p_1)[\\ ? \mapsto g_0(q_3, r_1, x_1, p_1)?[\\ \ldots p'_{i-1} \mapsto g_0(q_3, r_1, x_0, p_{i-1})[\\ ? \mapsto g_0(q_3, r_1, x_1, p_{i-1})?] \ldots]]]; \end{aligned}$$

$$g_0{}'(q_2, r, x_0, p_j)? :=? \text{ for each } 1 \leq j < i;$$

3. $P := p_{i+1} \ldots p_n$, and $v := p_{i+1}$.

Last, $g_0{}'(q_2, r, x_1, y_1) := x'_1$.

$\{x_0 := x_2[x_0]; x_2 :=?\}$: we need to "synchronize" the variables representing the left and right parts in a way similar to the previous case.

$\{x_0 := x_1; x_1 := x_0\}$: every component involving x_0 is swapped with the corresponding x_1 component.

Conflict relation and well-formedness. First of all we need to show that the assignments are *consistent* with respect to the parameters. Let's assume by contradiction that at some point in the computation, for some $q \in Q$ and $x \in X$ some parameter $u' \in \Pi$ appears twice in $\varphi(q, x)$. This means that there exists a run of S_2 in which a variable u appears twice in a right hand side, violating the copyless assignment.

Next, we show that there exists a conflict relation η over X consistent with ρ. We'll often use the fact that, in assignments of the form $x := yz$ or $x := y[z]$, it is always the case that $y \neq z$. The conflict relation η is defined as follows: for all $q_1, q_1', q_2, q_2' \in Q_2$, $r_1, r_1', r_2, r_2' \in R$, $x \in X_{1,0}$, $y \in X_{1,1}$, $u, v \in X_2$,

- if $(q_1, r_1) \neq (q_1', r_1')$, then $\eta(g_0(q_1, r_1, x, u), g_0(q_1', r_1', x, v))$;
- if $(q_1, r_1) \neq (q_1', r_1')$, then $\eta(g_0(q_1, r_1, x, u_?), g_0(q_1', r_1', x, v_?))$;
- if $(q_1, r_1, r_2) \neq (q_1', r_1', r_2')$, then $\eta(g_1^L(q_1, r_1, r_2, y, u), g_1^L(q_1', r_1', r_2', y, v))$;
- if $(q_1, r_1, r_2) \neq (q_1', r_1', r_2')$, then $\eta(g_1^L(q_1, r_1, r_2, y, u_?), g_1^L(q_1', r_1', r_2', y, v_?))$;
- if $(q_1, q_2, r_1, r_2) \neq (q_1', q_2', r_1', r_2')$, then $\eta(g_1^R(q_1, q_2, r_1, r_2, y, u), g_1^R(q_1', q_2', r_1', r_2', y, v))$;
- if $(q_1, q_2, r_1, r_2) \neq (q_1', q_2', r_1', r_2')$, then $\eta(g_1^R(q_1, q_2, r_1, r_2, y, u_?), g_1^R(q_1', q_2', r_1', r_2', y, v_?))$.

We now show that the variable update function ρ does not violate the conflict relation η. Inspecting the updates we perform, it is easy to see that the same variable never appears twice on the right-hand side of the same variable. Now, by way of contradiction, let's assume there exists an assignment which violates the constraints (we indicate in bold the meta-variables of S and in italic those of S_2). There are two possibilities:

1. $\mathbf{x} \neq \mathbf{y}$, $\eta(\mathbf{x}, \mathbf{y})$, and both $\mathbf{x}$ and $\mathbf{y}$ occur on the right-hand side of some variable;
2. $\eta(\mathbf{x}, \mathbf{y})$, and there exists two variables $\mathbf{x}'$ and $\mathbf{y}$, such that $\mathbf{x}' := \text{FUN}(\mathbf{x})$, $\mathbf{y}' = \text{FUN}(\mathbf{y})$ for which $\eta(\mathbf{x}', \mathbf{y}')$ doesn't hold.

Case (1) can be ruled out by simply inspecting all the possible assignments in the definition of ρ. The only interesting cases are $\{x_0 := x_0 x_1; x_1 := \varepsilon\}$ and $\{x_0 := x_2[x_0]; x_2 :=?\}$ where the reasoning is the following. We first need to show that, for every $q_2 \in Q_2$, $r \in R$, $x_0 \in X_{1,0}$, if $X_{2,1} = \{y_1, \ldots, y_n\}$ is the set of type-1 variables in S_2, then the sequence $p_0(q_2, r, x_0, y_1) \ldots p_0(q_2, r, x_0, y_n)$ is repetition free. It is easy to show that this property holds by induction. After we have this, it is easy to show that the assignments do not violate the conflict relation.

We now need to deal with the conflict case (2). Before starting we point out that every variable $x \in X_1$ appears in at most one of the assignments of S_2 due to the copyless restriction. We want to show that it cannot happen that two variables that are in conflict are assigned to two different variables that are not in conflict. Let's try to analyze when two variables $\mathbf{x}, \mathbf{y}$ assigned to different variables can be in conflict. The first case is $\mathbf{x} = \mathbf{y}$. The case for $\{x_0 := w\}$ can be ruled out by inspecting the assignments. For the cases $\{x_0 := x_0 x_1; x_1 := \varepsilon\}$ we observe the following: the only case in which two variables appearing on two different right hand sides conflict is when (looking at point two of the iteration) we perform the following update: $\{g_0(q_2, r, x_0, v) := g_0(q_3, r, x_1, v); g_0(q_3, r, x_1, v) := \ldots g_0(q_3, r, x_1, v) \ldots\}$. The two left-hand side are in conflict, therefore η is well defined. For the case $x_0 := x_2[x_0]$ the argument is analogous.

Next, we show how the construction deals with calls and returns. As in the proof of Theorem 5.14, at every call, S stores the state containing the information about the current variable values on the stack, and, at the corresponding return we use them to construct the new values for the state. Since at every call variables are reset, this construction is quite straightforward. Using an argument similar to that of Theorem 5.14, assignments do not violate the single use restriction. Notice that the fact that the variables are reset at calls is crucial for this construction.

Output function. Finally, we define the output function F. When reaching the last symbol, we need to construct the final output, but now at this point, we know what states to use. We illustrate the construction with an example, since the general one is very similar to the construction of ρ. Let's assume we are in state $q_S = (q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$, then $F(q_S)$ is defined as follows. We assume, without loss of generality, that $F_1(q) = x$ for some $x_0 \in X_{1,0}$, since the other cases are similar to our previous constructions. If $f_0(q_{02}, r_0, x_0) = q_f$, and $F_2(q_f) = y_0$, with $y_0 \in X_{2,0}$, then $F(q_S) = g_0(q_{02}, r_0, x_0, y_0)\{v' \mapsto \varepsilon\}$ for every $v' \in \varphi(q_S, g_0(q_{02}, r_0, x_0, y_0))$. This concludes the proof. □

5.6 Restricted inputs

A nested word captures both linear and hierarchical structure. There are two natural subclasses of nested words: strings are nested words with only linear structure, and ranked trees are nested words with only hierarchical structure. Let us consider how the definition of STT can be simplified when the input is restricted to these two special cases.

5.6.1 Mapping strings

Suppose we restrict the inputs to contain only internal symbols, that is, strings over Σ. Then the STT cannot use its stack, and we can assume that the set P of stack symbols is the empty set. This restricted transducer can still map strings to nested words (or trees) over Γ with interesting hierarchical structure, and hence, is called a *string-to-tree* transducer. This leads to the following definition: a *streaming string-to-tree transducer* (SSTT) § from input alphabet Σ to output alphabet Γ consists of a finite set of states Q; an initial state $q_0 \in Q$; a finite set of typed variables X together with a conflict relation η; a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that for each state q, a variable x appears at most once in $F(q)$; a state-transition function $\delta : Q \times \Sigma \mapsto Q$; and a variable-update function $\rho : Q \times \Sigma \mapsto \mathcal{A}(X, X, \eta, \Gamma)$. Configurations of such a transducer are of the form (q, α), where $q \in Q$ is a state, and α is a type-consistent valuation for the variables X. The semantics $[\![S]\!]$ of such a transducer is a partial function from Σ^* to $W_0(\Gamma)$. We notice that in this setting the copyless restriction is enough to capture MSO completeness since the model is closed under regular look-ahead (i.e. a reflexive η is enough).

Theorem 5.16 (Copyless SSTTs are closed under RLA)**.** *A string-to-tree transduction is definable by an SSTT iff it is definable by a copyless SSTT.*

Proof. The $\Leftarrow$ direction is immediate. For the $\Rightarrow$ direction, using Theorem 5.9 we consider the input to be a copyless SSTT with regular look-ahead. Given a DFA $A = (R, r_0, \delta_A)$ over the alphabet Σ, and a copyless string-to-tree STT $S = (Q, q_0, X, F, \delta, \rho)$ over R, we construct an equivalent copyless multi-parameter STT $S' = (Q', q'_0, X', \Pi, \varphi, F', \delta', \rho')$ over Σ. Thanks to Theorem 5.11 this implies the existence of a copyless SSTT.

Auxiliary notions. Given a finite set U, we inductively define the following sets:

$\mathcal{F}(U)$*:* the set of forests over U defined as: $\varepsilon \in \mathcal{F}(U)$, and if $s_0, \ldots, s_n \in U$, and $f_0, \ldots, f_n \in \mathcal{F}(U)$, then $s_0(f_0) \ldots s_n(f_n) \in \mathcal{F}(U)$;

$\mathcal{SF}(f')$*:* given a forest $f' \in \mathcal{F}(U)$, the set of sub-forests of f', for short $\mathcal{SF}(f')$, is defined as follows:

- if $f' \equiv s'_0(t'_0) \ldots s'_m(t'_m)$, and there exist $i \leq m$ such that $f \in \mathcal{SF}(t'_i)$, then $f \in \mathcal{SF}(f')$;
- if $f \in \mathcal{SFM}(f')$, then $f \in \mathcal{SF}(f')$;

- the empty forest ε belongs to $\mathcal{SM}(f')$;
- let $f \equiv s_0(t_0)\dots s_n(t_n)$, and $f' \equiv s'_0(t'_0)\dots s'_m(t'_m)$. If there exist $0 \leq i \leq m-n$ such that $s_0 \dots s_n = s'_i \dots s'_{i+n}$, and for every $0 \leq j \leq n$, $t_j \in \mathcal{SFM}(t'_{i+j})$, then $f \in \mathcal{SF}(f')$.

Intuitively, the first rule in the definition of $\mathcal{SF}$ allows to move down in the forest until an exact match in $\mathcal{SFM}$ is found by the other rules. Given two forests $f_1, f_2 \in \mathcal{F}(U)$, we write $\mathcal{S}(f_1, f_2) \equiv \mathcal{SF}(f_1) \cap \mathcal{SF}(f_2)$ for the set shared sub-forests of f_1 and f_2. Finally the set of maximal shared sub-forests is defined as

$$\mathcal{M}(t_1, t_2) = \{f \mid f \in \mathcal{S}(t_1, t_2) \wedge \neg\exists f' \in \mathcal{S}(t_1, t_2).f' \neq f \wedge f \in \mathcal{SF}(f')\}.$$

State components and invariants. The transition of the STT S at a given step depends on the state of A after reading the reverse of the suffix. Since the STT S' cannot determine this value based on the prefix, it needs to simulate S for every possible choice. We denote by X_i the type-i variables of X. Every state $q \in Q'$ is a tuple (l, f, g) where

- $l : R \mapsto R$, keeps track, for every possible state $r \in R$, of what would be the state of A after processing the string $\text{REV}(w)$, where w is the string read so far;
- $f : R \mapsto Q$, keeps track, for every possible state $r \in R$, of what would be the state of S after processing the string w_r, where w is the string read so far;
- $g : (R \times X) \mapsto F(X' \cup \{?\})$ keeps track of how the variables X' of S' need to be combined in order to obtain the value of a variable of S.

State summarization invariants. We first discuss the invariants of the first two components l and f of a state, and how they are preserved by the transition function δ' of S'. After reading a word w S' is in state (l, f, g) where

- for every look-ahead state $r \in R$, $l(r) = r'$, $\delta^*_A(r, \text{REV}(w)) = r'$;
- for every look-ahead state $r \in R$, $f(r) = q$, $\delta^*(q_0, w_r) = q$.

At the beginning S' is in state (l_0, f_0, g_0), where for every $r \in R$, $l_0(r) = r$ and $f_0(r) = q_0$. The component g_0 is discussed later.

Next we describe the transition function δ'. We assume S' to be in state (l, f, g), and to be reading the input symbol $a \in \Sigma$; we denote with l', f' the new values of the state components, and we only write the parts that change. For every look-ahead state $r \in R$, if $\delta_A(r, a) = r'$, $f(r) = q$, and $\delta(q, r') = q'$, then $l'(r) = l(r')$ and $f'(r) = q'$.

Variable summarization. Next, we describe how S' keeps track of the variable values. The natural approach for this problem would be that of keeping, for each state $r \in R$ and variable $x \in X$, a variable $g(r, x)$ containing the value of x in S, assuming the prefix read so far, was read by A starting in state r. This natural approach, however, would cause the machine not to be copyless. Consider, for example, the following scenario. Let r, r_1 and r_2 be look-ahead states in R such that, for some $a \in \Sigma$, $\delta(r_1, a) = \delta(r_2, a) = r$. Assume S only has one state $q \in Q$, and one variable $x \in X$. If S updates $\rho(q, r, x)$ to x, in order to perform the corresponding update in S' we would have to assign $g(r, x)$ to both $g(r_1, x)$ and $g(r_2, x)$, and this assignment is not copyless.

Our solution to this problem relies on a symbolic representation of the update and a careful analysis of sharing. In the previous example, a possible way to represent such update is by storing the content of $g(r, x)$ into a variable z, and then remembering in the state the fact that both $g(r_1, x)$ and $g(r_2, x)$ now contain z as a value. In the construction, the above update is reflected by updating the state, without touching the variable values.

The set of variables X' contains $|R|(4|X_0||R|)$ type-0 variables, and $|R|(4|X_1||R|)$ type-1 variables. The set of parameters Π of S' is $\{\pi_i \mid 0 \leq i \leq 4|X||R|\}$. We will show later how these numbers are obtained.

Variables semantics and invariants. We next describe how we can recover the value of a variable in X from the corresponding shape function $g(x)$. Intuitively, the value of a variable x is recovered by merging together the variables appearing in the shape of x. We call this operation unrolling.

We define the unrolling $u : \mathcal{F}(X') \mapsto E(X', \Sigma)$ of a symbolic variable representation as follows. Given a forest $f = s_0(f_0) \ldots s_n(f_n) \in \mathcal{F}(X')$, the unrolling of f is defined as $u(f) \equiv u_t(s_0, f_0) \ldots u_t(s_n, f_n)$, where for every $s \in X$, and $g_0 \ldots g_m \in X^*$, $u_t(s, g_0 \ldots g_m) \equiv s[\pi_0 \mapsto u(g_0), \ldots, \pi_m \mapsto u(g_m)]$.

After reading i-th symbol a_i of an input word w, S' is in a configuration $((l, f, g), \alpha)$ iff for every look-ahead state $r \in R$, and variable $x \in X$, if $\delta^*(q_0, \textsc{rev}((a_1 \ldots a_i)_r)) = (q, \alpha_1)$, and $u(g(r, x)) = s$, then $\alpha_1(x) = \alpha(s)$.

Counting argument invariants. Next, we describe how we keep the shape function g compact, allowing us to use a finite number of variables while updating them in a copyless manner. The shape function g maintains the following invariants.

Single-Use: each shape $g(r,x)$ is repetition-free: no variable $x' \in X'$ appears twice in $g(r,x)$.

Sharing Bound: for all states $r, r' \in R$, $\sum_{x,y \in X} |\mathcal{M}(g(r,x), g(r',y))| \leq |X|$.

Hole Placement: for every type-1 variable $x \in X_1$, and state $r \in R$, there exists exactly one occurrence of ? in $g(r,x)$, and it does not have any children.

Horizontal compression: for every $ff' \in \mathcal{SF}(g(r,x))$, such that $? \notin ff'$, then there must be a shape $g(r',x')$, with $(r',x') \neq (r,x)$, such that either $f \in \mathcal{SF}(g(r',x'))$ and $ff' \notin \mathcal{SF}(g(r',x'))$, or $f' \in \mathcal{SF}(g(r',x'))$ and $ff' \notin \mathcal{SF}(g(r',x'))$.

Vertical compression: for every $s(f) \in \mathcal{SF}(g(r,x))$, such that $? \notin s(f)$, then there must be a shape $g(r',x')$, with $(r',x') \neq (r,x)$, such that either $s() \in \mathcal{SF}(g(r',x'))$ and $s(f) \notin \mathcal{SF}(g(r',x'))$, or $f \in \mathcal{SF}(g(r',x'))$ and $s(f) \notin \mathcal{SF}(g(r',x'))$.

The first invariant ensures the bounded size of shapes. The second invariant limits the amount of sharing between shapes and it implies that, for each r, and for each $x \neq y$, $g(r,x)$ and $g(r,y)$ are disjoint. The second invariant also implies that, for every state r, the tree $g(r,x)$, for all x cumulatively, can have a total of $|X||R|$ maximal shared subforests, with respect to all other strings. The third invariant says that the tree of each type-1 variable contains exactly one hole and this hole appears as a leaf. This helps us dealing with variable substitution. The fourth and fifth compression invariants guarantee that the shapes use only the minimum necessary amount of variables. Together they imply that the sum $\sum_{x \in X} |g(r,x)|$ is bounded by $4|X||R|$. This is due to the fact that a shape can be updated only in three ways, 1) on the left, 2) on the right, and 3) below the ?. As a result it suffices to have $|R|(4|X||R|)$ variables in Z.

Variable and state updates. Next we show how g and the variables in X' are updated and initialized.

The initial value g_0 in the initial state, and each variable in X' are defined as follows: let $X'_0 = \{z_1, \ldots, z_k\}$, $X'_1 = \{z'_1, \ldots, z'_k\}$, $X_0 = \{x_1, \ldots, x_i\}$, $X_1 = \{x'_1, \ldots, x'_j\}$. For each type-0 variable $x_i \in X_0$, for each type-1 variable $x'_i \in X_1$, and look-ahead state $r \in R$, we have that $g_0(r, x_i) = z_i$, $z_i = \varepsilon$, $g_0(r, x'_j) = z'_j(?)$, and $z'_j = \pi_0$.

We assume S' to be in state (l, f, g), and to be reading the input symbol $a \in \Sigma$. We denote with g' the new value of g. We assume we are given a look-ahead state $r \in R$, such that $\delta_A(r,a)$ to be equal to r', and x and y are the variables to be the updated.

$\{x := w\}$: where without loss of generality $w = \langle a?b \rangle$. We first assume there exists an unused variable, and then show that such a variable must exist. Let z_f be an unused variable. We update $g'(r,x) = z_f$, and set $z_f := \langle a?b \rangle$. Since the counting invariants are preserved, there must have existed an unused variable z_f.

$\{x := xy, y := \varepsilon\}$: we perform the following update: $g'(r,x) = g(r',x)g(r',y)$. Let $g(r',x) = s_1(f_1)\dots s_n(f_n)$ and $g(r',y) = s'_1(f'_1)\dots s'_m(f'_m)$. We now have two possibilities:

- there exists $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s_n or s'_1, but not $s_n s'_1$; or
- there does not exist $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s_n or s'_1, but not $s_n s'_1$. In this case we can compress: if $f_n = t_1 \dots t_i$ and $f'_1 = t'_1 \dots t'_k$, then $s_n := s_n s'_1[\pi_0 \mapsto \pi_i, \dots, \pi_k \mapsto \pi_{k+i}]$ and $g'(r,x) = s_1(f_1)\dots s_n(f_n f'_1)s'_2(f'_2)\dots s'_m(f'_m)$. In this new assignment to s_n, the parameters in s'_1 have been shifted by i to reflect the concatenation with s_n.

In both cases, due to the preserved counting invariant, we can take an unused variable z_f and use it to update $g(r,y)$: $z_f := \varepsilon$, and $g'(r,y) = z_f$.

$\{x := x[y], y :=?)\}$: without loss of generality let x and y be type-1 variables. Let $s(?)$ be the subtree of $g(r',x)$ containing ?. We perform the following update: $g'(r,x) = g(r',x)\{?/g(r',y)\}$, where $a\{b/c\}$ replaces the node b of a with c. Let $g(r',y) = s'_1(f'_1)\dots s'_m(f'_m)$. For every s_l, we now have two possibilities:

- there exists $p \leq m'$, and $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s or s'_p, but not $s(s'_p)$; or
- there does not exist $p \leq m'$, and $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s or s'_p; in this case we can compress: assume $f'_p = t_1 \dots t_k$, then $s := s[\pi_i \mapsto s'_p[\pi_0 \mapsto \pi_i, \dots, \pi_k \mapsto \pi_{k+i}], \pi_{i+1} \mapsto \pi_{i+1+k}, \dots, \pi_{n+k+1}]$, and $g'(r,x) = g'(r,x)\{s(s'_p)/s\}$.

In both cases, due to the preserved counting invariant, we can take an unused variable z_f and use it to update $g(r,y)$: $z_f := \pi_0$, and $g'(r,y) = z_f(?)$. Figure 5.3 shows an example of such an update.

$\{x := y, y := x\}$: we symbolically reflect the swap. $g'(r,x) = g(r',y)$, and $g'(r,y) = g(r',x)$. Similarly to before, we compress if necessary.

By inspection of the variable assignments, it is easy to see that S' is copyless.

Figure 5.3 shows an example of an update of the form $x = x[y]$. In the figure, each variable z_i belongs to X'. The depicted update represents the case in which we are

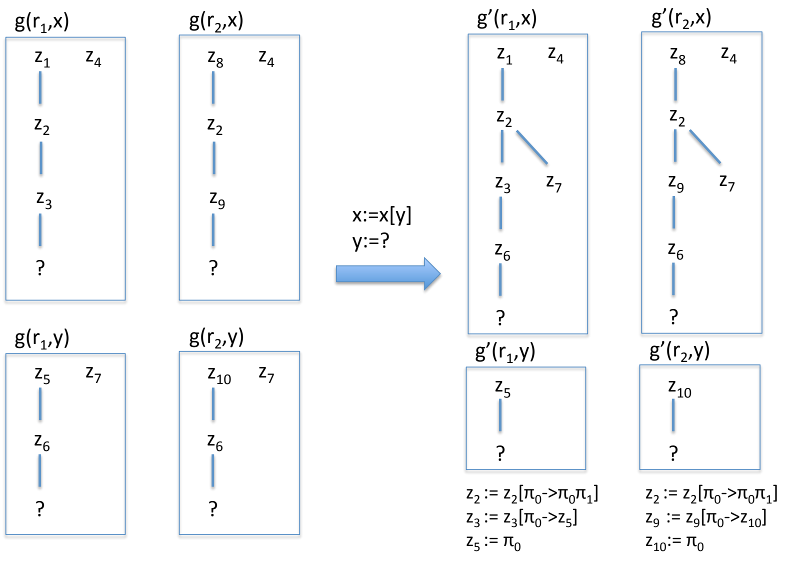


FIGURE 5.3: Example of symbolic variable assignment.

reading the symbol a such that $\delta_A(r_1, a) = r_1$ and $\delta_A(r_2, a) = r_1$. Before reading a (on the left), the variables z_2, z_4, z_6, and z_7 are shared between the two representations of the variables at r_1 and r_2. In the new shape g' the hole ? in $g(r_1, x)$ (respectively $g(r_2, x)$) is replaced by $g(r_1, y)$ (respectively $g(r_2, y)$). However, since the sequence $z_3 z_5$ (respectively $z_9 z_{10}$) is not shared, we can compress it into a single variable z_3 (respectively z_9), and reflect such a compression in the variable update $z_3 := z_3[\pi_0 \mapsto z_5]$ (respectively $z_9 := z_9[\pi_0 \mapsto z_{10}]$). Now the variable z_5 (respectively z_{10}) is unused and we can therefore use it to update $g'(r_1, y)$ (respectively $g'(r_2, y)$).

The output function F, of S' simply applies the unrolling function. For example, let's assume S' ends in state $(l, f, g) \in Q'$, with $l(r_0) = r$, $f(r) = q$, and $F(q) = xy$. We have that $F(l, f, g) = u(g(r_0, x)g(r_0, y))$. This concludes the proof. $\square$

5.6.2 Mapping ranked trees

In a ranked tree, each symbol a has a fixed arity k, and an a-labeled node has exactly k children. Ranked trees can encode terms, and existing literature on tree transducers focuses primarily on ranked trees. Ranked trees can be encoded as nested words of a special form, and the definition of an STT can be simplified to use this structure. For simplicity of notation, we assume that there is a single 0-ary symbol $\mathbf{0} \notin \Sigma$, and every symbol in Σ is binary. The set $BT(\Sigma)$ of *binary trees* over the alphabet Σ is then a subset of nested words defined by the grammar $T := \mathbf{0} \mid \langle a\, T\, T\, a \rangle$, for $a \in \Sigma$. We will use the more familiar tree notation $a\langle t_l, t_r \rangle$, instead of $\langle a\, t_l\, t_r\, a \rangle$, to denote a binary tree with a-labeled root and subtrees t_l and t_r as children. The definition of an STT can be simplified in the following way if we know that the input is a binary tree. First, we do not need to worry about processing of internal symbols. Second, we restrict to bottom-up STTs due to their similarity to bottom-up tree transducers, where the transducer returns, along with the state, values for variables ranging over output nested words, as a result of processing a subtree. Finally, at a call, we know that there are exactly two subtrees, and hence, the propagation of information across matching calls and returns using a stack can be combined into a unified combinator: the transition function computes the result corresponding to a tree $a\langle t_l, t_r\rangle$ based on the symbol a, and the results of processing the subtrees t_l and t_r.

A *bottom-up ranked-tree transducer* (BRTT) § from binary trees over Σ to nested words over Γ consists of a finite set of states Q; an initial state $q_0 \in Q$; a finite set of typed variables X equipped with a conflict relation η; a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that for each state q, the expression $F(q)$ is consistent with η; a state-combinator function $\delta : Q \times Q \times \Sigma \mapsto Q$; and a variable-combinator function $\rho : Q \times Q \times \Sigma \mapsto \mathcal{A}(X, X_l \cup X_r, \eta, \Gamma)$, where X_l denotes the set of variables $\{x_l \mid x \in X\}$, X_r denotes the set of variables $\{x_r \mid x \in X\}$, and the conflict relation η extends to these sets naturally: for every $x, y \in X$, if $\eta(x, y)$, then $\eta(x_l, y_l)$, and $\eta(x_r, y_r)$. The state-combinator extends to trees in $BT(\Sigma)$: $\delta^*(\mathbf{0}) = q_0$ and $\delta^*(a\langle t_l, t_r\rangle) = \delta(\delta^*(t_l), \delta^*(t_r), a)$. The variable-combinator is used to map trees to valuations for X: $\alpha^*(\mathbf{0}) = \alpha_0$, where α_0 maps each type-0 variable to ε and each type-1 variable to ?, and $\alpha^*(a\langle t_l, t_r\rangle) = \rho(\delta^*(t_l), \delta^*(t_r), a)[X_l \mapsto \alpha^*(t_l)][X_r \mapsto \alpha^*(t_r)]$. That is, to obtain the result of processing the tree t with a-labeled root and subtrees t_l and t_r, consider the states $q_l = \delta^*(t_l)$ and $q_r = \delta^*(t_r)$, and valuations $\alpha_l = \alpha^*(t_l)$ and $\alpha_r = \alpha^*(t_r)$, obtained by processing the subtrees t_l and t_r. The state corresponding to t is given by the state-combinator $\delta(q_l, q_r, a)$. The value $\alpha^*(x)$ of a variable x corresponding to t is obtained from the right-hand side $\rho(q_l, q_r, a)(x)$ by setting variables in X_l to values given by α_l and setting variables in X_r to values given by α_r. Note that the consistency with the conflict relation ensures that each value gets used only once. Given a tree $t \in BT(\Sigma)$, let $\delta^*(t)$ be q and let $\alpha^*(t)$ be α. Then, if $F(q)$ is undefined then $[\![S]\!](t)$ is undefined, else $[\![S]\!](t)$ equals $\alpha(F(q))$ obtained by evaluating the expression $F(q)$ according to valuation α.

Theorem 5.17 (Expressiveness of ranked tree transducers)**.** *A partial function from $BT(\Sigma)$ to $W_0(\Gamma)$ is STT-definable iff it is BRTT-definable.*

Proof. From STT to BRTT $\Rightarrow$. Using Theorem 5.7, let $S = (Q, P, q_0, X, F, \eta, \delta, \rho)$ be a binary bottom-up STT. We construct a BRTT $S' = (Q', q_0', X', F', \eta', \delta', \rho')$ equivalent to S. We can assume w.l.o.g. that the set of variables X is partitioned into two disjoint sets $X_l = \{x_1^L, \ldots, x_n^L\}$ and $X_r = \{x_1^R, \ldots, x_n^R\}$ such that, given a tree $t = a\langle t_l, t_r\rangle$, 1) after processing the left child t_l, all the variables in X_l depend on the run over t_l, while all the variables in X_r are reset to their initial values, and 2) after processing the second child, the values of X_r only depend on t_r (do not use any variable $x^p \in X^p$), and every variable $x \in X_l$ is exactly assigned the corresponding value x^p stored on the stack. In summary, every variable in X_r only depends on the second child, and every variable in X_L only depends on the first child. Such variables can only be combined at the return $a\rangle$ at the end of t. An STT of this form can be obtained by delaying the variable update at the return at the end t_r to the next step in which $a\rangle$ is read.

Now that we are given a bottom-up STT, the construction of S' is similar to the one showed in Theorem 5.7. Each state $q \in Q'$ contains a function $f : Q \mapsto Q$, that keeps track of the executions of S for every possible starting state. After processing a tree t, S' is in state f, such that for every $q \in Q$, $f(q) = q'$ iff when S reads the tree t starting in state q, it will end up in state q'. Similarly, the BRTT S' uses multiple variables to keep track of all the possible states with which a symbol could have been processed. The BRTT S' has set of variables $X' = \{x_q \mid x \in X, q \in Q\}$. After processing a tree t, for every $x \in X$, and $q \in Q$, if, after S processes t starting in state q, x contains the value s, then x_q also contains the value s.

Next we describe how these components are updated. Let f_l and f_r be the states of S' after processing the children t_l and t_r of a tree $a\langle t_l, t_r\rangle$. We denote with f' the new state after reading a. For every state $q, \in Q$, and variable $x \in X$, if $\delta_c(q, \langle a) = (q_1, p)$, $f_l(q_1) = q_2$, $f_r(q_2) = q_3$, and $\delta_r(q_3, p, \langle a) = q_4$, then $f'(q) = q_4$, and x_q is assigned the value $\rho(q_3, p, \langle a, x)$, in which every variable in X_l and X_r is replaced with the corresponding variables in X'_l and X'_R. For every state f, the output function of F' of S' is then defined as $F'(f) = F(f(q_0))$. The conflict relation η' of S' has the following rules:

1. for every $x, y \in X$, and $q \neq q' \in Q$, $\eta'(x_q, y_{q'})$;
2. for every $q \in Q$, and $x, y \in X$, if $\eta(x, y)$, then $\eta'(x_q, y_q)$.

The proof of consistency is the same as for Theorem 5.7.

From BRTT to STT $\Leftarrow$. Given a BRTT $S = (Q, q_0, X, F, \eta, \delta, \rho)$, we construct an STT $S' = (Q', P', q'_0, X', F', \eta', \delta', \rho')$ equivalent to S. The STT S' simulates the execution of S while reading the input nested word. The states and stack states of S' are used to keep track of whether the current child is a first or second child. The STT S' has set of states $Q' = Q \times (\{l, r\} \cup Q)$, initial state $q'_0 = (q_0, l)$, and set of stack state $P' = Q \times \{l, r\}$. Given a tree $\langle a\ t_l, t_r\ a\rangle$, the STT S' maintains the following invariant:

- right before processing the first child t_l, S' is in state (q_0, l);
- right before processing the second child t_r, S' is in state (q_1, r);

- right after processing the second child t_r, S' is in state (q_1, q_2).

We now define the transition relation δ' that preserves the invariant defined above. Let's assume S' is in state $q = (q_1, d)$, and it is processing the symbol a.

a is a call $\langle b$: the STT S' resets the control to the initial state and pushes q_1 on the stack: $\delta'(q_1, b) = (q'_0, q_1)$.

a is a return $b\rangle$: we first of all observe that, since the input is a binary tree, d cannot have value r when reading a return symbol. If the state popped from the stack is $p = (q_p, d_p)$ we have the following cases.

End of First Child: if $d_p = l$ we have two cases:

Leaf: if $d = l$, and $\delta(q_0, q_0, b) = q_2$ then $\delta'(q, p, b) = (q_2, r)$;

Not a Leaf: if $d = q_2 \in Q$, and $\delta(q_1, q_2, b) = q_3$, then $\delta'(q, p, b) = (q_3, r)$.

End of Second Child: if $d_p = r$ we have two cases:

Leaf: if $d = l$, and $\delta(q_0, q_0, b) = q_2$, then $\delta'(q, p, b) = (q_p, q_2)$;

Not a Leaf: if $d = q_2 \in Q$, and $\delta(q_1, q_2, b) = q_3$, then $\delta'(q, p, b) = (q_p, q_3)$.

The STT S' has set of variables $X' = X'_l \cup X'_r$, where $X'_d = \{x_d \mid x \in X\}$. We use one set of variables X_l for the variable values after processing the left child, and X_r for the variable values after processing the right child. The variables in X_l and X_r are combined when processing the parent. The STT S' maintains the following invariant: given a tree $\langle a\ t_l, t_r\ a\rangle$, and $d \in \{l, r\}$, after processing the child t_d, every variable $x_d \in X'_d$ will contain the value of x computed by S after processing the tree t_d.

We can now describe the variable update function ρ' of S'. Given a variable $x \in X'$, let the function $\textsc{ini}(x)$ be the function that returns the initial value of a variable x, that is defined as: 1) if x is a type-0 variable, then $\textsc{ini}(x) = \varepsilon$, and 2) if x is a type-1 variable, then $\textsc{ini}(x) =?$. Let's assume S' is in state $q = (q_1, d)$, it is processing the symbol a, and updating the variable $x \in X'$:

a is a call $\langle b$: every variable is copied on the stack $\rho'(q, b, x) = x$;

a is a return $b\rangle$: if the state popped from the stack is $p = (q_p, d_p)$, we have the following cases.

End of First Child: if $d_p = l$ we have two cases:

- if $x \in X_L$ we have two more cases

  Leaf: if $d = l$, then $\rho'(q, p, b, x) = \rho(q_0, q_0, b, x)$;

Not a Leaf: if $d = q_2 \in Q$, then $\rho'(q, p, b, x) = \rho(q_1, q_2, b, x)$.

- if $x \in X_R$, then $\rho'(q, p, b, x) = \textsc{ini}(x)$.

End of Second Child: if $d_p = r$ we have two cases

- if $x \in X_L$, then $\rho'(q, p, b, x) = x^p$;
- if $x \in X_R$ we have two more cases:

  Leaf: if $d = l$, then $\rho'(q, p, b, x) = \rho(q_0, q_0, b, x)$;

  Not a Leaf: if $d = q_2 \in Q$, then $\rho'(q, p, b, x) = \rho(q_1, q_2, b, x)$.

The conflict relation η' of S' is defined as: for every $x, y \in X$, and $d \in \{L, R\}$, if $\eta(x, y)$, then $\eta'(x_d, y_d)$. The consistency of η is trivial. Finally, the output function F' is defined as follows. For every state $(q, d) \in Q'$, if $d = r$, $F'(q, d) = F(q)$, otherwise F' is undefined. □

5.7 Restricted outputs

Let us now consider how the transducer model can be simplified when the output is restricted to the special cases of strings and ranked trees. The desired restrictions correspond to limiting the set of allowed operations in expressions used for updating variables.

5.7.1 Mapping nested words to strings

Each variable of an STT stores a potential output fragment. These fragments get updated by addition of outputs symbols, concatenation, and insertion of a nested word in place of the hole. If we disallow the substitution operation, then the STT cannot manipulate the hierarchical structure in the output. More specifically, if all variables of an STT are type-0 variables, then the STT produces outputs that are strings over Γ. The set of expressions used in the right-hand sides can be simplified to $E_0 := \varepsilon \mid a \mid x_0 \mid E_0 E_0$. That is, each right-hand side is a string over $\Gamma \cup X$. Such a restricted form of STT is called a *streaming tree-to-string transducer* (STST). While less expressive than STTs, this class is adequate to compute all tree-to-string transformations, that is, if the final output of an STT is a string over Γ, then it does not need to use holes and substitution.

Theorem 5.18 (STST expressiveness). *A partial function from $W_0(\Sigma)$ to Γ^* is STT-definable iff it is STST-definable.*

Proof. Since STSTs are also STTs, the $\Leftarrow$ direction of the proof is immediate.

We now prove the $\Rightarrow$ direction. Given an STT S that only outputs strings over Γ^*, we can build an equivalent STST S' as follows. The goal of the construction is eliminating all the type-1 variables from S. This can be done by replacing each type-1 variable x in S with two type-0 variables x_l, x_r in S' representing the values to the left and to the right of the ? in x. If, after reading a nested word w, the type-1 variable x of S contains the value $w_l?w_r$, then the type-0 variables x_l and x_r of S' respectively contain the values w_l and w_r. Notice that this cannot be done in general for SSTs, because w_l and w_r might not be well-matched nested words. The states, and state transition function of S' are the same as for S, and the variable update function and the output function can be easily derived. If η is the conflict relation of S, then the conflict relation η' of S' is defined as follows: for every variable x, $\eta'(x,x)$, and given two variables $x \neq y$ of S, if $\eta(x,y)$, and $d, d' \in \{l, r\}$, then $\eta'(x_d, y_{d'})$. $\square$

If we want to compute string-to-string transformations, then the STT does not need a stack and does not need type-1 variables. Such a transducer is both an SSTT and an STST, and this restricted class coincides with the definition of streaming string transducers (SST) [AC11].

5.7.2 Mapping nested words to ranked trees

Suppose we are interested in outputs that are binary trees in $BT(\Gamma)$. Then, variables of the transducer can take values that range over such binary trees, possibly with a hole. The internal symbols, and the concatenation operation, are no longer needed in the set of expressions. More specifically, the grammar for the type-0 and type-1 expressions can be modified as

$$\begin{aligned} E_0 &:= \mathbf{0} \mid x_0 \mid a\langle\, E_0\, E_0 \,\rangle \mid E_1[E_0] \\ E_1 &:= \; ? \mid x_1 \mid a\langle\, E_0\, E_1 \,\rangle \mid a\langle\, E_1\, E_0 \,\rangle \mid E_1[E_1] \end{aligned}$$

where $a \in \Gamma$, $x_0 \in X_0$ and $x_1 \in X_1$. To define transformations from ranked trees to ranked trees, we can use the model of bottom-up ranked-tree transducers with the above grammar.

5.8 Expressiveness

The goal of this section is to prove that the class of nested-word transductions definable by STTs coincides with the class of transductions definable using Monadic Second Order logic (MSO).

We first show that any STT can be simulated by an MSO definable transduction. The other direction of the proof relies on the known equivalence between MSO and Macro Tree Transducers over ranked trees. Our first step is to lift STTs to operate over ranked trees. Next, to simulate a given Macro Tree Transducer, we show construct a sequence of STTs over ranked trees. Finally, using closure under composition, this sequence of STTs is composed into a single STT.

5.8.1 MSO for nested-word transductions

Formulas in monadic second-order logic (MSO) can be used to define functions from (labeled) graphs to graphs [Cou94]. We adapt this general definition for our purpose of defining transductions over nested words. A nested word $w = a_1 \dots a_k$ over Σ is viewed as an edge-labeled graph G_w with $k+1$ nodes $v_0 \dots v_k$ such that (1) there is an edge from each v_{j-1} to v_j, for $1 \leq j \leq k$, labeled with the symbol $a_j \in \Sigma$, and (2) for every pair of matching call-return positions i and j, there is an unlabeled (nesting) edge from v_{i-1} to v_{j-1}. The monadic second-order logic of nested words is given by the syntax:

$$\phi := a(x,y) \mid X(x) \mid x \rightsquigarrow y \mid \phi \vee \phi \mid \neg\phi \mid \exists x.\phi \mid \exists X.\phi$$

where $a \in \Sigma$, x, y are first-order variables, and X is a second-order variable. The semantics is defined over nested words in a natural way. The first-order variables are interpreted over nodes in G_w, while set variables are interpreted over sets of nodes. The formula $a(x,y)$ holds if there an a-labeled edge from the node x to node y (this can happen only when y is interpreted as the linear successor position of x), and $x \rightsquigarrow y$ holds if the nodes x and y are connected by a nesting edge.

An *MSO nested-word transducer* Φ from input alphabet Σ to output alphabet Γ consists of a finite set C called the *copy set*, node formulas ϕ^c for each $c \in C$, each of which is an MSO formula over nested words over Σ with one free first-order variable x, and edge formulas $\phi^{c,d}$ and $\phi_a^{c,d}$ for each $a \in \Gamma$ and $c, d \in C$, each of which is an MSO formula over nested words over Σ with two free first-order variables x and y. Intuitively, given an input nested word, Φ builds an output graph corresponding to the output nested word; this graph can contain up to C times as many nodes (the copies) as the ones appearing in the input nested word; for each node x and $c \in C$, x^c denotes the c-th copy of the node x. Each unary formula ϕ^c is used to decide whether the c-th copy of a node x should appear in the graph, the binary formula $\phi^{c,d}$ is used to decide whether there exists a nesting edge between the c-th copy of x and the d-th copy of y, and the binary formula $\phi_a^{c,d}$ is used to decide whether there exists an a-labeled edge between the c-th copy of x and the d-th copy of y. Given an input nested word w, consider the following output graph: for each node x in G_w and $c \in C$, there is a node x^c in the output if the formula ϕ^c holds over G_w, and for all such nodes x^c and y^d, there is an a-labeled edge from x^c to y^d if the formula $\phi_a^{c,d}$ holds over G_w, and there is a nesting edge from x^c to y^d if the formula $\phi^{c,d}$ holds over G_w. If this graph is the graph corresponding to the nested word u over Γ then $[\![\Phi]\!](w) = u$, and otherwise $[\![\Phi]\!](w)$ is undefined. A nested-word transduction f from input alphabet Σ to output alphabet Γ is *MSO-definable* if there exists an MSO nested-word transducer Φ such that $[\![\Phi]\!] = f$.

By adapting the simulation of string transducers by MSO [EH01, AC10], we show that the computation of an STT can be encoded by MSO, and thus, every transduction computable by an STT is MSO definable.

Theorem 5.19 (STT-to-MSO)**.** *Every STT-definable nested-word transduction is MSO-definable.*

Proof. Consider a copyless STT S with regular look-ahead automaton A. The labeling of positions of the input nested word with states of the regular look-ahead automaton can be expressed in MSO. The unique sequence of states and stack symbols at every step of the execution of the transducer S over a given input nested word w can be captured in MSO using second order existential quantification. Thus, we assume that each node in the input graph is labeled with the corresponding state of the STT while processing the next symbol. The positions corresponding to calls and returns are additionally labeled with the corresponding stack symbol pushed/popped.

We explain the encoding using the example shown in Figure 5.4. Suppose the STT uses one variable x of type-1. The corresponding MSO transducer has eight copies in the copy set, four for the current value of the variable and four for the current value of the variable on the top of the stack. The current value of the variable x is represented by 4 copies in the copy set: x_i, x_o, $x_{i?}$ and $x_{o?}$. At every step i (see top of Figure 5.4) the value of x corresponds to the sequence of symbols labeling the unique path starting at x_i and ending at $x_{i?}$, followed by the hole ? and by the sequence labeling the unique path starting at $x_{o?}$ and ending at x_o. At step i the value of x on top of the stack (x_p in this example) is captured similarly.

We now explain how the STT variable updates are captured by the MSO transducer. At step 0, x is instantiated to ? by adding an ε-labeled edge from the x_i node to the $x_{i?}$ node

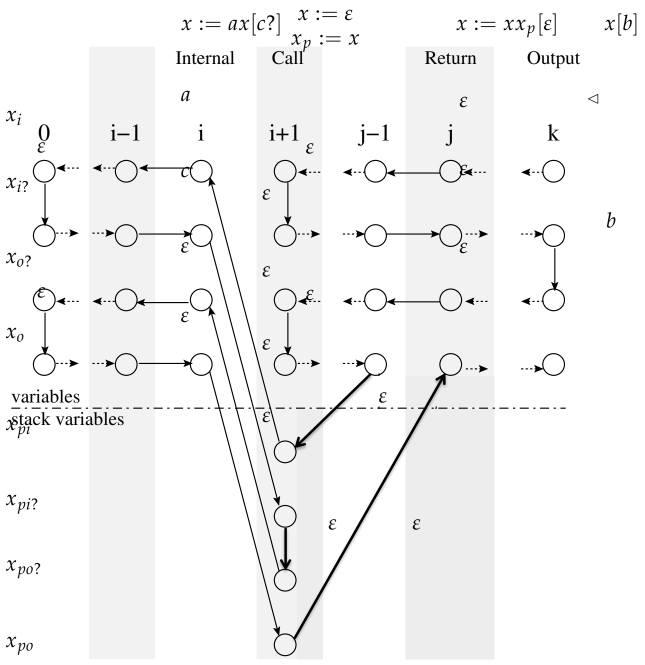


FIGURE 5.4: Encoding of an STT's computation in MSO.

and from the $x_{o?}$ node to the x_o node. Consider an internal position i and the variable assignment $x := ax[c?]$. This means that the value of x for column i is the value of x in column $i-1$, preceded by the symbol a, where we add a c before the parameter position in $i-1$. To reflect this assignment, we insert an a-labeled edge from the x_i node in column i to the x_i node in column $i-1$, a c-labeled edge from the $x_{i?}$ node in column $i-1$ to the $x_{i?}$ node in column i (this reflects adding to the left of the ?), an ε-labeled edge from the $x_{o?}$ node in column i to the $x_{o?}$ node in column $i-1$, and an ε-labeled edge from the x_o node in column $i-1$ to the x_o node in column i.

We again use Figure 5.4 to show how variables are stored on the stack. At the call step $i+1$, the assignment $x_p := x$ is reflected by the ε-labeled edges between the x_p nodes in column $i+1$ and the x nodes in column i. The edges are added in a similar way as for the previous assignment. The value of x_p in column $i+1$ is preserved unchanged until the corresponding matching return position is found. At the return step j, the value of x can depend on the values of x in column $j-1$, and the value of x_p on the top stack, which is captured by the input/output nodes for x_p in column $j-1$. The $j-1$ position can be identified uniquely using the matching relation $\rightsquigarrow$. Even though it is not shown in figure, at position j we have to add ε edges from the x_p nodes at position j to the x_p nodes at position i to represent the value of x_p that now is on the top of the stack. In Figure 5.4 the bold edges connect the current variable values to the values on top of the stack and they are added when processing the return position j.

To represent the final output, we use an additional column k. In the example, the output expression is $x[b]$. We mark the first edge from x_i by a special symbol $\lhd$ to indicate where the output string starts, a b-labeled edge from the $x_{i?}$ node to the $x_{o?}$ node of x. In the same way as before we connect each component to the corresponding $k-1$ component using ε edges.

Notice that in this exposition, we have assumed that in the MSO transducer, edges can be labeled with strings over the output alphabet (including ε) instead of single symbols. It is easy to show that allowing strings to label the edges of the output graph does not increase the expressiveness of MSO transducers. Also notice that not every node will appear in the final output string. An MSO transducer able to remove the useless edges and nodes can be defined. Using closure under composition we can then build the final transducer. Some extra attention must be paid to add the matching edges in the output, but since the output at every point is always a well-matched nested word, the matching relation over the output nested word can be induced by inspection of each assignment (i.e. the matching edges are always between nodes in the same column). $\square$

Nested Words as Binary Trees

Nested words can be encoded as binary trees. This encoding is analogous to the encoding of *unranked* trees as binary trees. Such an encoding increases the depth of the tree by imposing unnecessary hierarchical structure, and thus, is not suitable for processing of inputs. However, it is useful to simplify proofs of subsequent results about expressiveness. The desired transduction *nw_bt* from $W_0(\Sigma)$ to $BT(\Sigma)$ is defined by:

$$\begin{aligned}
nw\_bt(\varepsilon) &= \mathbf{0};\\
nw\_bt(aw) &= a\langle\, nw\_bt(w), \mathbf{0}\,\rangle;\\
nw\_bt(\langle a\, w_1\, b\rangle\, w_2) &= a\langle\, nw\_bt(w_1), b\langle\, nw\_bt(w_2), \mathbf{0}\,\rangle\,\rangle.
\end{aligned}$$

Notice that the tree corresponding to a nested word w has exactly one internal node for each position in w. Observe that *nw_bt* is a one-to-one function, and in particular, the encodings of the two nested words aaa and $\langle a\, a\, a\rangle$ differ:

- $nw\_bt(aaa) = a\langle\, a\langle\, a\langle\, \mathbf{0}, \mathbf{0}\,\rangle, \mathbf{0}\,\rangle, \mathbf{0}\,\rangle$;
- $nw\_bt(\langle aaa\rangle) = a\langle\, a\langle\, \mathbf{0}, \mathbf{0}\,\rangle, a\langle\, \mathbf{0}, \mathbf{0}\,\rangle\,\rangle$.

We can define the inverse partial function *bt_nw* from binary trees to nested words as follows: given $t \in BT(\Sigma)$, if t equals $nw\_bt(w)$, for some $w \in W_0(\Sigma)$ (and if so, the choice of w is unique), then $bt\_nw(t) = w$, and otherwise $bt\_nw(t)$ is undefined. The next proposition shows that both these mappings can be implemented as STTs.

Proposition 5.20 (Nested words to binary trees correspondence). *$nw\_bt : W_0(\Sigma) \mapsto BT(\Sigma)$, and $bt\_nw : BT(\Sigma) \mapsto W_0(\Sigma)$ are both STT-definable transductions.*

Proof. We prove the two statements in the order.

STT for *nw_bt***.** The transduction *nw_bt* can be performed by an STT S that basically simulates its inductive definition and only needs one type-1 variable x. When processing the input nested word $w = a_1 \dots a_n$, after reading the symbol a_i, $x[\mathbf{0}]$ contains the value of $nw\_bt(\textsc{wms}(w,i))$ (see Theorem 5.7 for the definition of $\textsc{wms}(w,i)$). The STT S has one state q, which is also initial. The set of stack states is the same as the input alphabet $P = \Sigma$, and the output function is $F(q) = x[\mathbf{0}]$. Next we define the update functions. For every $a, b \in \Sigma$,

- $\delta_i(q,a) = q$, $\delta_c(q,a) = (q,a)$, and $\delta_r(q,a,b) = q$;
- $\rho_i(q,a,x) = x[a\langle ?, \mathbf{0}\rangle]$, $\rho_c(q,a,x) = x$, and $\rho_r(q,a,b) = x^p[a\langle x[\mathbf{0}], b\langle ?, \mathbf{0}\rangle\rangle]$.

STT for *bt_nw***.** The translation *bt_nw* can be implemented by the following BRTT S'. The BRTT S' has three type-0 variables x_i, x_c, x_r such that after processing the tree t, 1) x_i contains the value of $bt\_nw(t)$, assuming t's root was representing an internal symbol, 2) x_c contains the value of $bt\_nw(t)$, assuming t's root was representing a call, and 3) x_r contains the value of $bt\_nw(t)$, assuming t's root was representing a return. The BRTT S' has set of states $Q' = \{q_0\} \cup (\{q_a \mid a \in \Sigma\} \times \{C, IR\})$, and initial state q_0. The $\{C, IR\}$ component is used to remember whether the last symbol S' read was a call, or an internal or return symbol. The BRTT S' is in a state $(q_a, \_)$ after processing a tree rooted with the symbol a. The output function F' is defined as follows: for every $a \in \Sigma$, $F'(q_a, IR) = x_i$, $F'(q_a, C) = x_c$, and F' is undefined otherwise. Next we define the update functions. When a variable is not assigned, we assume it is set to ε. For every symbol $a \in \Sigma$,

Right Child is $\mathbf{0}$*:* for every $q' \in Q$, $\delta(q', q_0, a) = (q_a, IR)$, and

- if $q' = q_0$, then $\rho(q_0, q_0, a, x_i) = a$;
- $q' = (q_b, C)$ for some $b \in \Sigma$, then $\rho((q_b, C), q_0, a, x_i) = a x_c^l$, and $\rho((q_b, IR), q_0, a, x_r) = x_c^l$;

- $q' = (q_b, IR)$ for some $b \in \Sigma$, then $\rho((q_b, IR), q_{\mathbf{0}}, a, x_i) = ax_i^l$, and $\rho((q_b, IR), q_{\mathbf{0}}, a, x_r) = x_i^l$.

Right child is not **0**: for every $q_l, q_r \in Q$, $\delta(q_l, q_r, a) = (q_a, C)$, and q_r must be of the form (q_b, IR) for some $b \in \Sigma$. The variable update is defined as follows:

- if $q_l = q_{\mathbf{0}}$, then $\rho(q_{\mathbf{0}}, (q_b, IR), a, x_c) = \langle ab \rangle x_r^r$;
- $q_l = (q_c, C)$ for some $c \in \Sigma$, then $\rho((q_c, C), (q_b, IR), a, x_c) = \langle ax_c^l b \rangle x_r^r$;
- $q_l = (q_c, IR)$ for some $c \in \Sigma$, then $\rho((q_c, IR), (q_b, IR), a, x_c) = \langle ax_i^l b \rangle x_r^r$.

Finally, the conflict relation of S' only contains $\eta'(x_i, x_r)$ □

For a nested-word transduction f from $W_0(\Sigma)$ to $W_0(\Gamma)$, we can define another transduction $\tilde{f}$ that maps binary trees over Σ to binary trees over Γ: given a binary tree $t \in BT(\Sigma)$, if t equals $nw\_bt(w)$, then $\tilde{f}(t) = nw\_bt(f(w))$, and otherwise $\tilde{f}(t)$ is undefined. The following proposition can be proved easily from the definitions of the encodings.

Proposition 5.21 (Encoding nested-word transductions). *If f is an MSO-definable transduction from $W_0(\Sigma)$ to $W_0(\Gamma)$, then the transduction $\tilde{f} : BT(\Sigma) \mapsto BT(\Gamma)$ is an MSO-definable binary-tree transduction and $f = bt\_nw \cdot \tilde{f} \cdot nw\_bt$.*

Since STT-definable transductions are closed under composition, to establish that every MSO-definable transduction is STT-definable, it suffices to consider MSO-definable transductions from binary trees to binary trees.

5.8.2 Macro tree transducers

A Macro Tree Transducer (MTT) [EV85, EM99] is a tree transducer in which the translation of a tree may depend not only on its subtrees but also on its context. While the subtrees are represented by input variables, the context information is handled by parameters. We refer the reader to Engelfriet et al. [EV85, EM99] for a detailed definition of MTTs, and present here the essential details. We only consider deterministic MTTs with regular look-ahead that map binary trees to binary trees.

A *(deterministic) macro-tree transducer with regular look-ahead* (MTTR) M from $BT(\Sigma)$ to $BT(\Gamma)$ consists of a finite set Q of ranked states, a list $Y = y_1, \ldots y_n$ of parameter symbols, variables $X = \{x_l, x_r\}$ used to refer to input subtrees, an initial state q_0, a finite set R of look-ahead types, an initial look-ahead type r_0, a look-ahead combinator

$\theta : \Sigma \times R \times R \mapsto R$, and the transduction function Δ. For every state q and every look-ahead type r, $\Delta(q, r)$ is a ranked tree over the alphabet $(Q \times X) \cup \Gamma \cup Y$, where the rank of a label (q, x) is the same as the rank of q, the rank of an output symbol $a \in \Gamma$ is 2, and the rank of each parameter symbol is 0 (that is, only leaves can be labeled with parameters).

The look-ahead combinator is used to define look-ahead types for trees: $\theta^*(\mathbf{0}) = r_0$ and $\theta^*(a\langle s_l, s_r \rangle) = \theta(a, \theta^*(s_l), \theta^*(s_r))$. Assume that only the tree $\mathbf{0}$ has the type r_0, and for every state q, $\Delta(q, r_0)$ is a tree over $\Gamma \cup Y$ (the variables X are used to refer to immediate subtrees of the current input tree being processed, and the type r_0 indicates that the input tree has no subtrees).

The MTTR M rewrites the input binary tree s_0, and at every step the output tree is a ranked tree with nodes labeled either with an output symbol, or with a pair consisting of a state of the MTTR along with a subtree of the input tree. Let $\mathcal{T}(s_0)$ denote the set of all subtrees of the input tree s_0. Then, the output t at any step is a ranked tree over $(Q \times \mathcal{T}(s_0)) \cup \Gamma \cup \{\mathbf{0}\}$. The semantics of the MTTR is defined by the derivation relation, denoted by $\Rightarrow$, over such trees. Initially, the output tree is a single node labeled with $[q_0, s_0]$. Consider a subtree of the output of the form $u = [q, s](t_1, \dots t_n)$, that is, the root is labeled with the state q of rank n, with input subtree s, and children of this node are the output subtrees $t_1, \dots t_n$. Suppose the look-ahead type of the input subtree s is r, and let s_l and s_r be the children of the root. Let χ be the tree obtained from the tree $\Delta(q, r)$ by replacing input variables x_l and x_r appearing in a node label with the input subtrees s_l and s_r respectively, and replacing each leaf labeled with a parameter y_l by the output subtree t_l. Then, in one step, the MTTR can replace the subtree u with the tree χ. The rewriting stops when all the nodes in the output tree are labeled only with output symbols. That is, for $s \in BT(\Sigma)$ and $t \in BT(\Gamma)$, $[\![M]\!](s) = t$ iff $[q_0, s] \Rightarrow^* t$.

In general, MTTs are more expressive than MSO. The restrictions needed to limit the expressiveness rely on the notions of *single-use* and *finite copying*, which enforce an MTT to process every subtree in the input a bounded number of times. Let M be an MTTR.

1. The MTTR M is *single use restricted in the parameters* (SURP) if for every state q and every look-ahead type r, each parameter y_j occurs as a node-label at most once in the tree $\Delta(q, r)$.

2. The MTTR M is *finite-copying in the input* (FCI) if there exists a constant K such that, for every tree s over Σ and subtree s' of s, if the (intermediate) tree t is derivable from $[q_0, s]$, then t contains at most K occurrences of the label $[q, s']$ (and thus, each input subtree is processed at most K times during a derivation).

Theorem 5.22 (Regularity for MTTs [EM99]). *A ranked-tree transduction f is MSO-definable iff there exists an MTTR M with SURP/FCI such that $f = [\![M]\!]$.*

5.8.3 MSO equivalence

We first show that bottom-up ranked-tree transducers are as expressive as MTTs with regular-look-ahead and single-use restriction, and then conclude that STTs are equivalent to MSO-definable transducers.

Theorem 5.23 (MTTs to BRTTs). *If a ranked-tree transduction $f : BT(\Sigma) \mapsto BT(\Gamma)$ is definable by an MTTR with SURP/FCI, then it is BRTT-definable.*

Proof. In the same way as we did for STTs, we can extend BRTTs to multi-parameter BRTTs (MBRTT). We omit the definition, since it is straightforward. Using the proof for Theorem 5.17 we can show that these are equivalent to multi-parameter STTs and therefore using Theorem 5.14 to STTs.

We are given a MTTR with SURP/FCI $M = (Q, Y, q_0, R, r_0, \theta, \Delta)$ with FCI constant K computing a transduction f and we construct a BRTT B equivalent to M.

We divide the construction into several steps, each step using one of the properties of the MTT.

1. We construct a BRTT S_1, computing the transduction $f_1 : BT(\Sigma) \mapsto BT(R)$, where each input element is replaced by its regular look-ahead state.
2. We construct an STT S_2, computing the transduction $f_2 : BT(R) \mapsto BT(R')$, where each element of R', contains information on the set of states in which the MTT processes the corresponding node.
3. We construct a multi-parameter BRTT S_3, that computes the function $f_3 : BT(R') \mapsto BT(\Gamma)$. This part relies on the SURP restriction.
4. Finally, we use the fact that STTs are closed under composition (Theorem 5.15), and the equivalence of STTs and BRTTs (Theorem 5.17), to show that $f = f_1 \cdot f_2 \cdot f_3$ is a BRTT definable transduction.

Step 1. The BRTT S_1 simulates the look-ahead automaton of M by following the transition relation θ. The set of states of S_1 is R, with initial state r_0, and it only has one type-0 variable x. For every symbol $a \in \Sigma$, and states $r_1, r_2 \in R$, the transition function δ_1, and the variable update function ρ_1 of S_1 are as follows: if $\theta(r_1, r_2, a) = r$, then $\delta_1(r_1, r_2, a) = r$, and $\rho_1(r_1, r_2, a, x) = r\langle x_l x_r \rangle$. Finally, for every $r \in R$, $F(r) = x$.

Step 2. STTs can also be viewed as a top down machine, and f_2 is in fact a top-down relabeling. Every subtree s can be processed at most K times, and we can therefore use S_2 to label s with the ordered sequence of states that processes it. So given a tree over $BT(R)$, S_2 outputs a tree over $BT(R')$, where $R' = R \times Q^{\leq K}$, and $Q^{\leq K} = \bigcup_{0 \leq k \leq K} Q^k$.

The states and stack states of S_2 are defined by the set $Q_2 = P_2 = Q^{\leq K} \cup (Q^{\leq K} \times Q^{\leq K})$. The initial state $q_0^2 = q_0 \in Q^{\leq K}$ means that the root is only processed by q_0. The construction of S_2 then maintains the following invariants: assume t_l and t_r are the left and right children of a node, $m_l \in Q^{\leq K}$ is the sequence of states that processes t_l in M, and $m_r \in Q^{\leq K}$ is the sequence of states that processes t_r in M;

- before processing t_l, S_2 is in state (m_l, m_r);
- before processing t_r (after processing t_l), S_2 is in state m_r.

The states m_i can be obtained directly from the right hand sides of the rules of the MTT. It's now trivial to do the corresponding labeling using the information stored in the state. Given a state $q \in Q$, and a symbol $r_1 \in R$, let $\mathcal{S}_d(q, r_1)$, with $d \in \{l, r\}$, be the sequence of states processing the child s_d in $\Delta(q, r_1)$, assuming a linearization of such a tree. For every sequence $m = q_1 \dots q_n \in Q^{\leq K}$, and symbol $r_1 \in R$, $\mathcal{S}_d(m, r_1) = \mathcal{S}_d(q_1, r_1) \dots \mathcal{S}_d(q_n, r_1)$. The STT S_2 only has one variable x.

We can now define the transition relation δ_2, and the variable update function ρ_2 of S_2. For every states $m \in Q_2$, $m' \in Q^{\leq K}$, $(m_1, m_2) \in Q^{\leq K} \times Q^{\leq K}$, and for every symbol $r \in R$:

r_1 is a call $\langle r_2$: store the variables on the stack and update the state consistently:

- $\delta_2(m', r_2) = (m_1, m')$, where $m_1 = (\mathcal{S}_l(m', r_2), \mathcal{S}_r(m', r_2))$, and
  $\delta_2((m_1, m_2), r_2) = (m_3, (m_1, m_2))$, where $m_3 = (\mathcal{S}_l(m_1, r_2), \mathcal{S}_r(m_1, r_2))$;
- $\rho_2(m', r_2, x) = x$, and $\rho_2((m_1, m_2), r_2, x) = x$;

r is a return $r_2\rangle$: use the values on the stack to compute the labeling:

- $\delta_2(m, (m_1, m_2), r_2) = m_2$, and $\delta_2(m, m', r_2) = m$;
- $\rho_2(m, (m_1, m_2), r_2, x) = \langle (r, m_1)\ x\ (r, m_1) \rangle$, and
  $\rho_2(m, m', r_2, x) = x^p\ \langle (r_2, m')\ x\ (r_2, m') \rangle$.

Finally, the output function F_2 of S_2 outputs x for every possible $q \in Q_2$.

Step 3. This last step relies on the SURP property of the MTT M. We notice that, when processing bottom-up the MTT parameter updates behave in a different way as when processing top-down: to perform the top-down parameter update $y_1 := a(y_2)$ in bottom-up manner, we need to use the multi-parameter BRTT parameter substitution $x := x[y_1 \mapsto a(y_2)]$, where now, the new visible parameter is y_2. We now formalize this idea.

We construct a multi-parameter BRTT $S_3 = (Q_3, q_{03}, \Pi_3, \eta_3, X_3, F_3, \delta_3, \rho_3)$ from $BT(R') \mapsto BT(\Gamma)$, that implements the transduction f_3. The set of states Q_3 only contains one state q_{03}, which is also initial. The transition function is therefore trivial.

The set of variables $X_3 = \{x_1, \ldots, x_K\}$ contains K variables. After processing a tree t, x_i contains the result of M processing the tree t starting in q_i, with the possible holes given from the parameters. At the beginning all the variable values are set to ε. The parameter set Π_3 is Y.

Next, we define the update functions of S_3. We start from the leaf rules, in which both the **0** children are labeled with the empty sequence. Let's assume the current leaf is (r, m), where $m = q_1 \ldots q_j$. We assume w.l.o.g., that all the states have exactly K parameters. For every $q_i \in m$, if $\Delta(q_i, r) = t_i(y_1, \ldots, y_K)$, we update $x_i := t_i(y_1, \ldots, y_K)$, where $y_1, \ldots, y_K \in \Pi$. Since the MTT is SURP, there is at most one occurrence of each y_i.

We now analyze the binary node rules. Let's assume the node we are processing the input node (r, m), where $m = q_1 \ldots q_j$. For every $q_i \in m$, $\Delta(q_i, r)$ is of the form

$$t_i(Y, (q^i_{l,1}, s_l), \ldots, (q^i_{l,a_i}, s_l), (q^i_{r,1}, s_r), \ldots, (q^i_{r,b_i}, s_r))$$

where $q^i_{l,1} \ldots q^i_{l,a_i}$ is the sequence of states processing the left subtree, and $q^i_{r,1} \ldots q^i_{r,b_i}$ is the sequence of states processing the right subtree.

When picking the concatenation of all the $\Delta(q_i, r)$, we have that by the construction of S_2, the left child (similarly for the right), must have been labeled with the sequence $m_l = q^1_{l,1} \ldots q^1_{l,a_1} \ldots q^j_{l,1} \ldots q^j_{l,a_j}$ such that $|m_l| \leq K$. Moreover, we have that for all $x_i \in X_l$ (similarly for X_r), x_i contains the output of M when processing the left child of the current node starting in state q_i, where q_i is the i-th element of the sequence m_l and assuming the parameter are not instantiated yet

Now we have all the ingredients to complete the rule. The right hand side of a variable x_i contains the update corresponding to the rule in M where we replace every state with the corresponding variable in the linearization stated above and parameters are updated via substitution. Since ρ is copyless η_3 is trivial. The output function F_S simply outputs x_1, i.e. the transformation of the input tree starting in q_0.

Step 4. We use Theorem 5.15, and Theorem 5.17 to compose all the transformations and build the final BRTT equivalent to the MTT M. □

Theorem 5.24 (MSO equivalence). *A nested-word transduction $f : W_0(\Sigma) \mapsto W_0(\Gamma)$ is STT-definable iff it is MSO-definable.*

Proof. From Theorems 5.15, 5.17, 5.19, 5.20, 5.22, and 5.23. □

5.9 Decision problems

In this section, we show that a number of analysis problems are decidable for STTs. Moreover, we show that STTs are the first model that capture MSO-definable transformations while enjoying an elementary upper bound for the problem of checking inequivalence (NEXPTIME).

5.9.1 Output analysis

Given an input nested word w over Σ, and an STT S from Σ to Γ, consider the problem of computing the output $[\![S]\!](w)$. To implement the operations of the STT efficiently, we can store the nested words corresponding to variables in linked lists with reference variables pointing to positions that correspond to holes. Each variable update can be executed by changing only a number of pointers that is proportional to the number of variables.

Proposition 5.25 (Computing output). *Given an input nested word w and an STT S with k variables, the output nested word $[\![S]\!](w)$ can be computed in time $\mathcal{O}(k|w|)$ in a single pass.*

Proof. A naive implementation of the transducer would cause the running time to be $\mathcal{O}(k|w|^2)$ due to possible variable sharing. Let's consider the assignment $(x, y) := (x, x)$. The naive implementation of this assignment would copy the value of the variable x in both x and y, causing the single step to cost $\mathcal{O}(k|w|)$ since every variable might contain a string of length $\mathcal{O}(|w|)$.

We now explain how we achieve the $\mathcal{O}(k|w|)$ bound by changing the representation of the output. Instead of outputting the final string, we output a pointer graph representation of the run. The construction is exactly the same as in Theorem 5.19. In this case the transducer might not be copyless; however, we can construct the graph in the same way. The key of the construction is that due to the definition of sharing, each variable contributes only once to the final output. In this way, starting from the output variable, we can reconstruct the output string by just following the edges in the graph. Notice that the stack variables won't cause a linear blow-up in the size of the graph because at every point in the run the graph only needs to represent the top of the stack. □

The second problem we consider corresponds to *type-checking*: given a regular language L_{pre} of nested words over Σ, a regular language L_{post} of nested words over Γ, and an STT S from Σ to Γ, the type-checking problem is to determine if $[\![S]\!](L_{pre}) \subseteq L_{post}$ (that is, if for every $w \in L_{pre}$, $[\![S]\!](w) \in L_{post}$).

Theorem 5.26 (Type-checking)**.** *Given an STT S from Σ to Γ, an NWA A accepting nested words over Σ, and an NWA B accepting nested words over Γ, checking $[\![S]\!](L(A)) \subseteq L(B)$ is solvable in* EXPTIME, *and more precisely in time $\mathcal{O}(|A|^3 \cdot |S|^3 \cdot n^{kn^2})$ where n is the number of states of B, and k is the number of variables in S.*

Proof. The construction is similar to the one of Theorem 5.15. Therefore we only present a sketch. Given S, A, and B, we construct an NWA P, that accepts a nested word w iff w is accepted by A and $[\![S]\!](w)$ is not accepted by B. This is achieved by summarizing the possible executions of B on the variable values of S. The states of P are triplets (q_A, q_S, f), such that:

- q_A is the current state of A;
- q_S is the current state of S;
- for every variable x of S, and states q_1, q_2 in B, $f(x, q_1, q_2)$ is a pair of states (q_1', q_2') of B, such that if the value of x is $w_1?w_2$:
  - if B reads w_1 starting in state q_1, then it ends in state q_1' and produces some stack Λ;
  - if B reads w_2 starting in state q_2, and with stack Λ, then it ends in state q_2'.

The final states of the machine are those where A is final, and the summary of the output function of the current state of Q_S of S leads to a non accepting state in B. □

As noted in Proposition 5.2, the image of an STT is not necessarily regular. However, the pre-image of a given regular language is regular, and can be computed. Given an STT S from input alphabet Σ to output alphabet Γ, and a language $L \subseteq W_0(\Gamma)$ of output nested words, the set $PreImg(L, S)$ consists of input nested words w such that $[\![S]\!](w) \in L$.

Theorem 5.27 (Computing pre-image). *Given an STT S from Σ to Γ, and an NWA B over Γ, there is an* EXPTIME *algorithm to compute an NWA A over Σ such that $L(A) = PreImg(L(B), S)$. The NWA A has size $\mathcal{O}(|S|^3 \cdot n^{kn^2})$ where n is the number of states of B, and k is the number of variables in S.*

Proof. The proof is the same as for Theorem 5.26, but this time the final states of the machine are those where S is in a final configuration, and the summary of the output function of the current state q_S of S leads to an accepting state in B. □

5.9.2 Functional equivalence

Finally, we consider the problem of checking *functional equivalence* of two STTs: given two STTs S and S', we want to check if they define the same transduction. Given two *streaming string transducers* S and S', one can construct an NFA A over the alphabet $\{0, 1\}$ such that the two transducers are *inequivalent* exactly when A accepts some string w such that w has equal number of 0's and 1's [AC10, AC11]. The idea can be adopted for the case of STTs, but A now is a nondeterministic pushdown automaton. The size of A is polynomial in the number of states of the input STTs, but exponential in the number of variables of the STTs. We then adapt existing results to show that [Esp97, SSMH04, KT10] it is decidable to check whether this pushdown automaton accepts a string with the same number of 0's and 1's.

Theorem 5.28 (Checking equivalence). *Given two STTs S_1 and S_2, it can be decided in* NEXPTIME *whether $[\![S]\!] \neq [\![S']\!]$.*

Proof. Two STTs S_1 and S_2 are inequivalent if one of the following holds:

1. for some input u, only one of $[\![S_1]\!](u)$ and $[\![S_2]\!](u)$ is defined;
2. for some input u, the lengths of $[\![S_1]\!](u)$ and $[\![S_2]\!](u)$ are different;
3. for some input u, there exist two symbols a and b, such that $a \neq b$, $[\![S_1]\!](u) = u_1 a u_2$, $[\![S_2]\!](u) = v_1 b v_2$, and u_1 and v_1 have the same length.

The first case can be solved in PTIME [AM09]. The second case can be reduced to checking an affine relation over pushdown automata and this problem can be solved in PTIME [MOS04]. Informally, let A be a pushdown automaton where each transition computes an affine transformation. Checking whether a particular affine relation holds at every final state is decidable in polynomial time [MOS04]. We can therefore take the product S_p of S_1 and S_2, where S_p updates the variables of S_1 and S_2 as follows. Let X_i

be the set of variables in S_i. For every state (q_1, q_2), symbol $a \in \Sigma$ and every $x \in X_i$, S_p updates the variable x to the sum of the number of constant symbols in the variable update of x in S_i when reading a in state q_i, and the variables appearing in such variable update. For every state (q_1, q_2) for which $F_1(q_1)$ and $F_2(q_2)$ are defined, we impose the affine relation $F_1(q_1) = F_2(q_2)$ which checks whether the two outputs have the same length. We can then use the algorithm by Müller-Olm and Seidl [MOS04] to check if such a relation holds.

Let us focus on the third case, in which the two outputs differ in some position. Given the STT S_1 and a symbol a, we construct a nondeterministic visibly pushdown transducer (VPT) V_1 (a visibly pushdown automaton with output), from Σ to $\{0\}$ such that V_1 produces 0^n on input u, if $[\![S_1]\!](u) = u_1 a u_2$ and $|u_1| = n$. Similarly, given S_2, and a symbol $b \neq a$, we construct a VPT V_2 from Σ to $\{1\}$ such that V_2 produces 1^n on input u, if $[\![S_2]\!](u) = u_1 b u_2$ and $|u_1| = n$.

Using V_1 and V_2 we then build the product $V = V_1 \times V_2$ which interleaves the outputs of V_1 and V_2. If V outputs a string s containing the same number of 0s and 1s, then there exists an input w on which S_1 outputs a string $\alpha_1 a \beta_1$, S_2 outputs $\alpha_2 b \beta_2$, and $|\alpha_1| = |\alpha_2|$. Since in V the input labels are synchronized, they are no longer relevant. Therefore, we can view such machine as a pushdown automaton that generates/accepts strings over $\{0, 1\}^*$. If a string s with the same number of 0s and 1s is accepted by V, it can be found by constructing the semi-linear set that characterizes the Parikh image of the context-free language of V [Esp97, SSMH04]. A solution to such semi-linear set can be found in NP (in the number of states of V) [KT10]. However, as we will see, the number of states of V is polynomial in the number of states of S_1 and S_2, and exponential in their number of variables of S_1 and S_2. This allows us to conclude that checking whether two STTs are inequivalent can be solved in NExpTime.

The rest of the proof shows how to construct V_1, V_2, and $V = V_1 \times V_2$. We modify $S_1 = (Q, q_0, P, X, \eta, F, \delta, \rho)$, so that we do not have to worry about the output function. We add to S_1 a new variable x_f, and a new state q_f, obtaining a new STT S'_1. We can then add a special end-of-input symbol #, such that, for every state $q \in Q$ for which $F(q)$ is defined, S'_1 goes from q to q_f on input #, and updates x_f to $F(q)$. This new STT S'_1 has the following property: for every input u, $[\![S_1]\!](w) = [\![S'_1]\!](w\#)$.

Each state of V_1 is a pair (q, f), where q is a state of S'_1, and f is a partition of the variables X of S'_1 into 6 categories $\{l, m1, m?, m2, r, n\}$, such that a variable x is in the set:

l: if x contributes to the final output occurring on the left of a symbol a where a is the symbol we have guessed the two transducers differ in the final output;

m1*:* if x contributes to the final output and the symbol a appears in this variable on the left of the ?;

m?*:* if x contributes to the final output and the symbol a appears in this variable in the ? (a future substitution will add a to the ?);

m2*:* if x contributes to the final output and the symbol a appears in this variable on the right of the ?;

r*:* if x contributes to the final output occurring on the right of a symbol a;

n*:* if x does not contribute to the final output.

At every step, V_1 nondeterministically chooses which of the previous categories each of the variables of S_1 belongs to. In the following, we use f_p to denote a particular partition: for example f_{m1} is the set of variables mapped by f to $m1$. A state (q, f) of V_1 is initial, if $q = q_0$ is the initial state of S_1, and $f_{m1} \cup f_{m2} = \emptyset$. A state (q, f) of V_1 is final, if $q = q_f$, $f_{m1} = \{x_f\}$ (the only variable contributing to the output is x_f), and $f_l \cup f_r \cup f_{m?} \cup f_{m2} = \emptyset$.

We now define the transition relation of V_1. Given a state s, and a symbol b, a transition updates the state to some s', and outputs a sequence in 0^*. We assume V_1 is in state (q, f), and it is processing the input symbol b. Given an expression $\alpha \in E(X, \Sigma)$ (i.e. an assignment right hand side), we use $x \in_a \alpha$ to say that a variable $x \in X$ appears in α.

b is internal: the VPT V_1 goes to state (q', f'), where $\delta_i(q, s) = q'$. For computing f' we have three different possibilities.

1. The VPT V_1 guesses that in this transition, some variable x is going to contain the position on which the outputs differ. We show the construction with an example. Assume $\rho_i(q, b, x) = \alpha_1 c \alpha_2 ? \alpha_3$, and c is the position on which we guess the output differs. The transition must satisfy the following properties, which ensures that the current labeling is consistent:
    - $\forall y \in_a \alpha_1 . y \in f_l$, $\forall y \in_a \alpha_2 \alpha_3 . y \in f_r$, $f'_{m1} = \{x\}$, and $f_m = \emptyset$ (the only variable that contributes in the middle now is x);
    - for every $y \neq x$, all the variables appearing to the left of the ? in $\rho_i(q, b, y)$ belong to the same partition p_l;
    - for every $y \neq x$ all the variables appearing to the right of the ? in $\rho_i(q, b, y)$ belong to the same partition p_r;
    - if $p_l = p_r$, then y belongs to f'_{p_l};
    - if $p_l \neq p_r$, then y belongs to $f'_{m?}$.

If a variable is assigned a constant, we nondeterministically choose which category it belongs to in f'. The output of the transition is 0^k, where k is the sum of the number of output symbols in α and in $\{\rho_i(q,b,y)|y \in f'_l\}$.

2. The transition is just maintaining the consistency of the partition, and the position on which the output differs has not been guessed yet. Similar to case (1).
3. The transition is just maintaining the consistency of the partition, and the position on which the output differs has already been guessed. Similar to case (1).

b is a call: in this case the updates are similar to the internal case. The updated partition is stored on the stack, and a new partition is non-deterministically chosen for the variables that are reset.

b is a return $b'\rangle$: the VPT V_1 uses the partition in the state for the variables in X, and the partition on the stack state for the variables in X_p. We assume (p, f') is the state on top of the stack. The VPT V_1 steps to (q'', f''), such that $\delta_r(q, p, b) = q'$. The computation of f' is similar to case in which a is internal.

For every transition we also impose the following conditions: the cardinality of $f_{m1} \cup f_{m2}$ is always less or equal than 1, and if a variable x does not appear in the right hand side of any assignment then $x \in f_n$. The first condition ensures that at most one variable contains the position on which the outputs differ. Due to the partitioning of the variables into the 6 categories V_1 will have exponentially many states. The VPT V_2 corresponding to S_2 and a symbol b such that $b \neq a$ can be built in a similar manner.

We finally show how to build the product VPT $V = V_1 \times V_2$ that reads nested words over the alphabet Σ and outputs strings over the alphabet $\{0,1\}$. The VPT V is a simple product construction where:

- each state of V is a pair (q_1, q_2), where q_i is a state of V_i;
- each stack state of V is a pair (p_1, p_2), where p_i is a stack state of V_i;
- the initial state is (q_1^0, q_2^0), where q_i^0 is the initial state of V_i;
- a state (q_1, q_2) is final iff q_i is a final state of V_i;
- if δ^1 and δ^2 are the transition functions of V_1 and V_2 respectively, then given a symbol $a \in \Sigma$, a state (q_1, q_2), and a stack state (p_1, p_2), the transition relation δ of V is defined as
  - if $\delta_i^1(q_1, a) = q_1', 0^n$ and $\delta_i^2(q_2, a) = q_2', 1^m$, then $\delta_i((q_1, q_2), a) = (q_1', q_2'), 0^n 1^m$;

- if $\delta_c^1(q_1, a) = q_1', p_1', 0^n$ and $\delta_c^2(q_2, a) = q_2', p_2', 1^m$, then $\delta_c((q_1, q_2), a) = (q_1', q_2'), (p_1', p_2'), 0^n 1^m$;
- if $\delta_r^1(q_1, p_1, a) = q_1', 0^n$ and $\delta_r^2(q_2, p_2, a) = q_2', 1^m$, then $\delta_r((q_1, q_2), (p_1, p_2), a) = (q_1', q_2'), 0^n 1^m$.

This concludes the proof. □

If the number of variables is bounded, then the size of V is polynomial, and this gives an upper bound of NP. For the transducers that map strings to nested words, that is, for streaming string-to-tree transducers (SSTT), the above construction yields a PSPACE bound [AC11].

Corollary 5.29 (Equivalence of string-to-tree transducers)**.** *Given two SSTTs S and S' that map strings to nested words, the problem of checking whether $[\![S]\!] = [\![S']\!]$ is solvable in* PSPACE.

Improving these bounds remains a challenging open problem.

5.10 Related work

We have proposed the model of streaming tree transducers to implement MSO-definable tree transformations by processing the linear encoding of the input tree in a single left-to-right pass in linear time. Below we discuss the relationship of our model to the rich variety of existing transducer models, and directions for future work.

5.10.1 Executable models

A streaming tree transducer is an executable model, just like a deterministic automaton or a sequential transducer, meaning that the operational semantics of the machine processing the input coincides with the algorithm to compute the output from the input and the machine description. Earlier executable models for tree transducers include bottom-up tree transducers, visibly pushdown transducers (a VPT is a sequential transducer with a visibly pushdown store: it reads the input nested word left to right producing output symbols at each step) [RS09], and multi bottom-up tree transducers (such a transducer computes a bounded number of transformations at each node by combining the transformations of subtrees) [ELM08]. Each of these models computes the output in a single left-to-right pass in linear time. However, none of these models can compute all MSO-definable transductions.

5.10.2 Regular look-ahead

Finite copying Macro Tree Transducers (MTTs) with regular look-ahead [EM99] can compute all MSO-definable ranked-tree-to-ranked-tree transductions. The "finite copying" restriction, namely, each input node is processed only a bounded number of times, can be equivalently replaced by the syntactic "single use restriction" which restricts how the variables and parameters are used in the right-hand sides of rewriting rules in MTTs. In all these models, regular look-ahead cannot be eliminated without sacrificing expressiveness: all of these process the input tree in a top-down manner, and it is well-known that deterministic top-down tree automata cannot specify all regular tree languages. A more liberal model, that uses a "weak finite copying" restriction, achieves closure under regular look-ahead and MSO-equivalence by allowing each input node to be processed an unbounded number of times, provided only a bounded subset of these contribute to the final output. It should be noted, however, that a linear time algorithm exists to compute the output [Man02]. This algorithm essentially uses additional look-ahead passes to label the input with the information needed to restrict attention to only those copies that contribute to the final output (in fact, Maneth [Man02] shows how relabeling of the input can be effectively used to compute the output of every MTT in time linear in the size of the input and the output). Finally, to compute tree-to-string transductions using regular look-ahead, MTTs need just one parameter (alternatively, top-down tree transducers suffice). In absence of regular look-ahead, even if the final output is a string, the MTT needs multiple parameters, and thus, intermediate results must be trees (that is, one parameter MTTs are not closed under regular look-ahead). Thus, closure under regular look-ahead is a key distinguishing feature of STTs.

5.10.3 From SSTs to STTs

The STT model generalizes earlier work on streaming string transducers (SST). An SST is a copyless STT without a stack [AC10, AC11]. While results in Section 5 follow by a natural generalization of the corresponding results for SSTs, the results in Section 3 and 4 require new approaches. In particular, equivalence of SSTs with MSO-definable string-to-string transductions is proved by simulating a two-way deterministic sequential transducer, a well-studied model known to be MSO-equivalent [EH01], by an SST. The MSO-equivalence proof in this chapter first establishes closure under regular look ahead, and then simulates finite copying MTTs with regular look-ahead. The natural analog of two-way deterministic string transducers would be the two-way version of visibly pushdown transducers [RS09]: while such a model has not been studied, it is easy to show that it would violate the "linear-bounded output" property of Proposition 1, and thus, won't be MSO-equivalent. While in the string case the copyless restriction does not reduce the expressiveness, in Section 5.4 we argue that the example conditional swap cannot be expressed by a copyless STT. Proving this result formally is a challenging open problem.

5.10.4 Succinctness

To highlight the differences in how MTTs and STTs compute, we consider two examples. Let f_1 and f_2 be two MSO-definable transductions, and consider the transformation $f(w) = f_1(w)f_2(w)$. An MTT at every node can send multiple copies to children and this enables a form of parallelism. Therefore, an MTT can compute f by having one copy compute f_1, and one copy compute f_2, and the size of the resulting MTT will be the sum of the sizes of MTTs computing f_1 and f_2. STTs are sequential and, to compute f, one needs the product of the STTs computing f_1 and f_2. This can be generalized to show that MTTs (or top-down tree transducers) can be exponentially more succinct than STTs. If we were to restrict MTT rules so that multiple states processing the same subtree must coincide, then this gap disappears. In the other direction, consider the transformation f' that maps input $u\#v\#a$ to uv if $a = 0$ and vu otherwise. The transduction f' can be easily implemented by an STT using two variables, one of which stores u and one which stores v. The ability of an STT to concatenate variables in any order allows it to output either uv or vu depending on the last symbol. In absence of look-ahead, an MTT for f' must use two parameters, compute (the tree encodings of) uv and vu separately in parallel, and make a choice at the end. This is because, while an MTT rule can swap or discard output subtrees corresponding to parameters, it cannot combine subtrees corresponding to parameters. This example can be generalized to show that an MTT must use exponentially many parameters as well as states, compared to an STT.

5.10.5 Input/output encoding

Most models of tree transducers process ranked trees (exceptions include visibly pushdown transducers [RS09] and Macro forest transducers [PS04]). While an unranked tree can be encoded as a ranked tree (for example, a string of length n can be viewed as a unary tree of depth n), this is not a good encoding choice for processing the input, since the stack height is related to depth (in particular, processing a string does not need a stack at all). We have chosen to encode unranked trees by nested words; formalization restricted to tree words (that are isomorphic to unranked trees) would lead to a slight simplification of the STT model and the proofs.

5.10.6 Streaming algorithms

Consistent with the notion of a streaming algorithm, an STT processes each input symbol in constant time. However, it stores the output in multiple chunks in different variables, rearranging them without examining them, making decisions based on finite-state control. Unlike a typical streaming algorithm, or a sequential transducer, the output of an STT is available only after reading the entire input. This is unavoidable if we want to compute a function that maps an input to its reverse. We would like to explore if the STT model can be modified so that it commits to output symbols as early as possible. A related direction of future work concerns minimization of resources (states and variables). Another streamable model is that of Visibly Pushdown Transducers (VPT) [FGRS11], which is however less expressive than STT. In particular VPTs cannot guarantee the output to be a well-matched nested word.

We have argued that SSTs correspond to a natural model with executable interpretation, adequate expressiveness, and decidable analysis problems, and in future work, we plan to explore its application to querying and transforming XML documents [Hos11, HMNI14] (see also `http://www.w3.org/TR/xslt20/`). Our analysis techniques typically have complexity that is exponential in the number of variables, but we do not expect the number of variables to be the bottleneck.

5.10.7 Complexity of checking equivalence

The problem of checking functional equivalence of MSO tree transducers is decidable with non-elementary complexity [EM06]. Decidability follows for MSO-equivalent models such as MTTs with finite copying, but no complexity bounds have been established. Polynomial-time algorithms for equivalence checking exist for top-down tree transducers (without regular look-ahead) and visibly pushdown transducers [CDG$^+$07, RS09, EMS09]. For STTs, we have established an upper bound of NEXPTIME, while the upper bound for SSTs is PSPACE [AC11]. Improving these bounds, or establishing lower bounds, remains a challenging open problem. If we extend the SST/STT model by removing the single-use restriction on variable updates, we get a model more expressive than MSO-definable transductions; it remains open whether the equivalence problem for such a model is decidable.

5.10.8 Alphabet limitations

STTs only operate over finite alphabets, and this can be a limitation in practical domains like XML processing. Integrating data values (that is, tags ranging over a potentially unbounded domain) in our model is an interesting research direction. A particularly suitable implementation platform for this purpose seems to be the framework of *symbolic automata* and *symbolic transducers* we discussed in Chapter 3. These models integrate automata-theoretic decision procedures on top of an SMT solver, enabling manipulation of formulas specifying input/output values from a large or unbounded alphabet in a symbolic and succinct manner [VB12b, DVLM14]. A symbolic version of visibly pushdown automata is discussed in Chapter 4.

Part III

Conclusion

Chapter 6

Future work

"If I had to do it all over again? Why not, I would do it a little bit differently."
— Freddie Mercury, *Life in His Own Words*

This dissertation demonstrates that automata and transducers are effective tools for reasoning about real-world programs that manipulate lists and trees. We first presented two novel transducer-based languages, BEX and FAST, used the former to verify efficient implementations of complex string coders, the latter to prove properties of HTML sanitizers, functional programs, and augmented reality taggers. We then introduced two new executable single-pass models for efficiently processing hierarchical data: symbolic visibly pushdown automata for monitoring properties involving infinite alphabets, and streaming tree transducers for expressing complex tree-to-tree transformations.

These results do not, however, close the book on automata- and transducer-based programming. In fact, our work can be seen as a new trampoline for designing languages and techniques that enable analysis and efficient execution of restricted classes of programs. This final chapter identifies specific avenues for future research.

6.1 Declarative languages

The languages described in Chapters 2 and 3 mainly acted as a frontend for the underlying transducer model: the language syntax and the transducer definition were nearly isomorphic. Another option is that of separating the two and designing declarative languages that can be compiled into transducers. For example, regular expressions are a declarative language that can be compiled into finite automata. Since declarative languages typically improve programmability and enable succinctness, one might wonder whether it is possible to design declarative programming languages that capture more classes of languages and transformations.

We and others have already made progress in this area. Alur, Freilich, and Raghothaman identified a set of combinators that captures exactly the class of regular string-to-string transformations [AFR14]. Based on these combinators, together with Alur and Raghothaman we designed DReX, a declarative language for string transformations [ADR15]. Programs written in DReX can be analysed statically and executed efficiently.

6.2 Algorithms

We described several novel models and decision procedures. For many of the models, in particular those supporting infinite alphabets, the theory is still not complete and many algorithms can be improved. For example, together with Veanes we developed new algorithms for minimizing symbolic automata [DV14]. Other problems remain open, despite a very active community. For the problem of checking equivalence of streaming tree transducers we provided a NExpTime upper bound, but we have no exponential lower bound. A similar result holds for streaming string transducers. Improving these bounds is a challenging and active research direction [FR14].

6.3 New models

We introduce new models that target particular application domains. There are countless possible extensions that could help capture richer domains. How does one handle infinite strings and trees? Can these models be extended with counters? Together with Alur, Deshmukh, Raghothaman, and Yuan, we investigated numerical extensions of streaming transducers [ADD$^+$13]. Understanding the properties of these models has been an active research direction [AR13].

6.4 Data-parallelism

In this dissertation we often emphasized how many automata and transducer models allow the efficient processing of inputs in a single pass. For very large inputs this is not enough and it is necessary to process the inputs in a parallel fashion. Veanes, Mytkowicz, Molnar, and Livshits have investigated how programs expressed as transducers can be used to generate efficient data-parallel code [VMML15]. This builds on the idea that the transition relation of a transducer can be represented as a matrix, and applying the transducer to the input corresponds to performing a matrix multiplication. In this representation, it can be shown that the state component of the transducer is not necessary. The input can therefore be divided into chunks that can be processed in parallel and then merged. This insight opens a new avenue in the world of automata-based programming. In particular, it would be interesting to create data-parallel models in the context of XML processing.

6.5 New applications

In this dissertation we only discussed a few applications of automata- and transducer-based languages. However, many more applications can benefit from these models.

6.5.1 Direct memory access in device drivers

Together with Albarghouthi, Cerny, and Ryzhyk, we are currently applying symbolic transducers to the verification of direct memory access (DMA) in device drivers.[1] Although there exist many tools for verifying device drivers [BLR11], verifying the DMA part remains challenging. When performing DMA the state space accessed by the driver becomes the whole memory and explicit techniques do not scale. Our goal is to model DMA operations, in particular those operating over packet queues, using transducer models. This requires the extension of our models to allow counters and data values, features that typically cause most procedures to become undecidable. We are currently in the process of identifying a model that can capture typical DMA operations without compromising decidability.

6.5.2 Networking

Recently Anderson, Foster, Guha, Jeannin, Kozen, Schlesinger, and Walker proposed a novel declarative language for verifying network configurations, NetKAT, that is strongly founded in automata theory [AFG$^+$14]. In NetKAT a network is abstracted

[1] `https://msdn.microsoft.com/en-us/library/windows/hardware/ff544074%28v=vs.85%29.aspx`

by an automaton that moves packets from node to node along the links in its topology. Actions in the network are then modeled using Kleene algebra with tests (KAT), an extension of regular expressions. Using KAT's closure properties and decision procedures the network can be then statically analyzed. Many of the ideas presented in this dissertation could be applied to NetKAT and in general to networking. Currently NetKAT explicitly queries each packet using multiple disjoint predicates, as KAT operates over finite alphabets. Symbolic automata techniques could be beneficial in reducing the complexity of these operations by representing predicates in a more succinct way, and by clearly separating the packet querying from the network operations.

6.5.3 Deep packet inspection

Fast identification of network traffic patterns is of vital importance in network routing, firewall filtering, and intrusion detection. This task is addressed with the name "deep packet inspection" (DPI) [SEJK08]. Due to performance constraints, DPI must be performed in a streaming fashion, and many automata models have been proposed to perform this task. Since packets contain large headers and payload, symbolic automata could be again beneficial in achieving succinctness. In particular, combining symbolic automata with BDDs could help leverage the bit-vector structure of the packets.

6.5.4 Binary assemblers and disassemblers

An assembler converts code written in assembly language into binary machine code. A disassembler inverts this operation. These programs are ubiquitous and hard to verify as they operate over large bit patterns. Transducers provide a potential solution to this problem. In particular, the models we devised in Chapter 2 to verify string coders could be adapted to work in this setting. Moreover, modeling disassemblers as transducers could provide new insights on deobfuscation, the task of extracting readable code from binaries.

6.5.5 Binary code similarity analysis

Identifying whether two binaries were compiled from similar pieces of code is necessary in identifying malware fingerprints – whether two programs contained the same malicious code. This task is challenging due to the many compiler transformations that slightly affect the structure of the compiled binary. Dalla Preda, Giacobazzi, Lakhotia, and Mastroieni recently showed how to treat symbolic automata as an abstract domain [DPGLM15] and used this notion to perform binary similarity analysis. The core idea is to represent programs as automata, and then use abstract interpreting operations on such automata. Although at first the programs would not be identical, iterated abstraction operations can remove unnecessary information which can make the two automata closer to each other. This application of symbolic automata is novel and many possible extensions are possible. For example, symbolic visibly pushdown automata could be used to extend the techniques presented by Dalla Preda et al. to operate over recursive programs.

Bibliography

[AC10] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In *FSTTCS*, pages 1–12, 2010.

[AC11] Rajeev Alur and Pavol Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11*, pages 599–610. ACM, 2011.

[AD12] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 42–53. Springer Berlin Heidelberg, 2012.

[ADD$^+$13] R. Alur, L. D'Antoni, J. Deshmukh, M. Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 13–22, June 2013.

[ADR15] Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. Drex: A declarative language for efficiently evaluating regular string transformations. *SIGPLAN Not.*, 50(1):125–137, January 2015.

[AFG$^+$14] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 113–126, 2014.

[AFR14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (Forthcoming)*, 2014.

[AM09] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, May 2009.

[AR13] Rajeev Alur and Mukund Raghothaman. Decision problems for additive regular functions. In *Automata, Languages, and Programming*, volume 7966 of *Lecture Notes in Computer Science*, pages 37–48. Springer Berlin Heidelberg, 2013.

[Bak79] Brenda S. Baker. Composition of top-down and bottom-up tree transductions. *Inform. and Control*, 41:186–213, 1979.

[BB13] Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. *SIGPLAN Not.*, 48(1):443–456, January 2013.

[BDM$^+$06] Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and xml reasoning. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 10–19, New York, NY, USA, 2006. ACM.

[BDM$^+$11] Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Logic*, 12(4):27:1–27:26, July 2011.

[Bec03] Oliver Becker. Streaming transformations for XML-STX. In Rainer Eckstein and Robert Tolksdorf, editors, *XMIDX 2003*, volume 24 of *LNI*, pages 83–88. GI, 2003.

[BHH$^+$08] Ahmed Bouajjani, Peter Habermehl, Lukas Holik, Tayssir Touili, and Tomas Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA'08*, volume 5148 of *Lecture Notes in Computer Science*, pages 57–67. Springer Berlin Heidelberg, 2008.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[CA07] Swarat Chaudhuri and Rajeev Alur. Instrumenting C programs with nested word monitors. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2007.

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

[CHMO15] Conrad Cotton-Barratt, David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. Fragments of ML decidable by nested data class memory automata. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 249–263, 2015.

[CK86] Karel Culik, II and Juhani Karhumäki. The equivalence of finite valued transducers (on hdt0l languages) is decidable. *Theor. Comput. Sci.*, 47(1):71–84, November 1986.

[Cou94] Bruno Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53 – 75, 1994.

[DA14] Loris D'Antoni and Rajeev Alur. Symbolic visibly pushdown automata. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer International Publishing, 2014.

[DGN$^+$13] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, and Mohamed Zergaoui. Early nested word automata for XPath query answering on XML streams. In *CIAA*, pages 292–305, 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DPGLM15] Mila Dalla Preda, Roberto Giacobazzi, Arun Lakhotia, and Isabella Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. *SIGPLAN Not.*, 50(1):329–341, January 2015.

[DTR12] Evan Driscoll, Aditya Thakur, and Thomas Reps. Opennwa: A nested-word automaton library. In P. Madhusudan and SanjitA. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 665–671. Springer Berlin Heidelberg, 2012.

[DV13a] Loris D'Antoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 624–639, Berlin, Heidelberg, 2013. Springer-Verlag.

[DV13b] Loris D'Antoni and Margus Veanes. Static analysis of string encoders and decoders. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI 2013*, volume 7737 of *LNCS*, pages 209–228. Springer, 2013.

[DV14] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 541–553, New York, NY, USA, 2014. ACM.

[DV15] Loris D'Antoni and Margus Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, pages 1–27, 2015.

[DVLM14] Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. Fast: A transducer-based language for tree manipulation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 384–394, New York, NY, USA, 2014. ACM.

[DZL03] Silvano Dal Zilio and Denis Lugiez. Xml schema, tree logic and sheaves automata. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263. Springer Berlin Heidelberg, 2003.

[EH01] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001.

[ELM08] Joost Engelfriet, Eric Lilin, and Andreas Maletti. Extended multi bottom-up tree transducers. In *Developments in Language Theory*, LNCS 5257, pages 289–300, 2008.

[EM99] Joost Engelfriet and Sebastian Maneth. Macro tree transducers, attribute grammars, and mso definable tree translations. *Information and Computation*, 154:34–91, 1999.

[EM02] J. Engelfriet and S. Maneth. Output string languages of compositions of deterministic macro tree transducers. *Journal of Computer and System Sciences*, 64(2):350 – 395, 2002.

[EM06] Joost Engelfriet and Sebastian Maneth. The equivalence problem for deterministic mso tree transducers is decidable. *Inf. Process. Lett.*, 100(5):206–212, December 2006.

[EMS09] Joost Engelfriet, Sebastian Maneth, and Helmut Seidl. Deciding equivalence of top-down {XML} transformations in polynomial time. *Journal of Computer and System Sciences*, 75(5):271 – 286, 2009.

[Eng75] Joost Engelfriet. Bottom-up and top-down tree transformations – a comparison. *Math. Systems Theory*, 9:198–231, 1975.

[Eng77] Joost Engelfriet. Top-down tree transducers with regular look-ahead. *Math. Systems Theory*, 10:289–303, 1977.

[Eng80] Joost Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R. V. Book, editor, *Formal Language Theory*, pages 241–286. Academic Press, 1980.

[Ési80] Zoltán Ésik. Decidability results concerning tree transducers. *Acta Cybernetica*, 5(1):1–20, 1980.

[Esp97] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundam. Inform.*, 31(1):13–25, 1997.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71 – 146, 1985.

[FGRS11] Emmanuel Filiot, Olivier Gauwin, Pierre-Alain Reynier, and Frédéric Servais. Streamability of nested word transductions. In Supratik Chakraborty and Amit Kumar, editors, *FSTTCS*, volume 13 of *LIPIcs*, pages 312–324. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[FR14] Emmanuel Filiot and Pierre-Alain Reynier. On streaming string transducers and HDT0L systems. *CoRR*, abs/1412.0537, 2014.

[FSVY91] Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science, 1991. LICS '91., Proceedings of Sixth Annual IEEE Symposium on*, pages 300–309, July 1991.

[FV88] Zoltán Fülöp and Sándor Vágvölgyi. Variants of top-down tree transducers with look-ahead. *Mathematical systems theory*, 21(1):125–145, 1988.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1998.

[FV14] Zoltán Fülöp and Heiko Vogler. Forward and backward application of symbolic tree transducers. *Acta Informatica*, 51(5):297–325, 2014.

[GLQ12] Pierre Geneves, Nabil Layaida, and Vincent Quint. On the analysis of cascading style sheets. In *WWW '12*, pages 809–818, New York, NY, USA, 2012. ACM.

[GN11] Olivier Gauwin and Joachim Niehren. Streamable fragments of forward xpath. In Béatrice Bouchou-Markhoff, Pascal Caron, Jean-Marc Champarnaud, and Denis Maurel, editors, *Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 3–15. Springer Berlin Heidelberg, 2011.

[Gri68] Thomas V. Griffiths. The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *J. ACM*, 15(3):409–413, July 1968.

[HLM$^+$11] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[HMNI14] Shizuya Hakuta, Sebastian Maneth, Keisuke Nakano, and Hideya Iwasaki. Xquery streaming by forest transducers. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 952–963, 2014.

[Hos11] H. Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge University Press, 2011.

[HP03] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003.

[HU79] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Iba78] Oscar H. Ibarra. The unsolvability of the equivalence problem for Efree NGSM's with unary input (output) alphabet and applications. *SIAM Journal on Computing*, 4:524–532, 1978.

[IH08] Kazuhiro Inaba and Haruo Hosoya. Multi-return macro tree transducers. In *Proc. 6th ACM SIGPLAN Workshop on Programming Language Technologies for XML*, San Francisco, California, January 2008.

[KCTV07] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS 2007*, pages 155–164. ACM/IEEE, 2007.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329 – 363, 1994.

[KGG$^+$09] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *ISSTA 2009, Proceedings of the 2009 International Symposium on Software Testing and Analysis*, Chicago, IL, USA, July 21–23, 2009.

[KT08] Michael Kaminski and Tony Tan. Tree automata over infinite alphabets. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science*, pages 386–423. Springer-Verlag, Berlin, Heidelberg, 2008.

[KT10] Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on*, pages 80–89, 2010.

[KTU10] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. *SIGPLAN Not.*, 45(1):495–508, January 2010.

[KV01] Armin Kühnemann and Janis Voigtländer. Tree transducer composition as deforestation method for functional programs, 2001.

[LG07] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 218–232, July 2007.

[Man02] Sebastian Maneth. The complexity of compositions of deterministic tree transducers. In *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference*, LNCS 2556, pages 265–276, 2002.

[MBPS05] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML type checking with macro tree transducers. In *PODS'05*, pages 283–294, New York, NY, USA, 2005. ACM.

[MGHK09] Andreas Maletti, Jonathan Graehl, Mark Hopkins, and Kevin Knight. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39(2):410–430, June 2009.

[MK08] Jonathan May and Kevin Knight. A primer on tree automata software for natural language processing, 2008.

[MNS09] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM J. Comput.*, 39(4):1486–1530, 2009.

[Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23(2):269–311, June 1997.

[MOS04] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. *SIGPLAN Not.*, 39(1):330–341, January 2004.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proc. 19th ACM Symposium on Principles of Database Systems (PODS'2000)*, pages 11–22. ACM, 2000.

[MZDZ13] Barzan Mozafari, Kai Zeng, Loris D'antoni, and Carlo Zaniolo. High-performance complex event processing over hierarchical data. *ACM Trans. Database Syst.*, 38(4):21:1–21:39, December 2013.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004.

[PS04] Thomas Perst and Helmut Seidl. Macro forest transducers. *Inf. Process. Lett.*, 89(3):141–149, February 2004.

[PS12] Adam Purtee and Lenhart Schubert. TTT: A tree transduction language for syntactic and semantic processing. In *Proceedings of the Workshop on Application of Tree Automata Techniques in NLP*, 2012.

[Rao92] Jean-Claude Raoult. A survey of tree transductions. In *Tree Automata and Languages*, pages 311–326. sn, 1992.

[RS09] Jean-Fancois Raskin and Fréd'eric Servais. Visibly pushdown transducers. In *Automata, Languages and Programming: Proceedings of the 35th ICALP*, LNCS 5126, pages 386–397, 2009.

[Sch75] Marcel Paul Schützenberger. Sur les relations rationnelles. In *Proceedings of the 2Nd GI Conference on Automata Theory and Formal Languages*, pages 209–213, London, UK, UK, 1975. Springer-Verlag.

[Seg06] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer Berlin Heidelberg, 2006.

[Sei94] Helmut Seidl. Equivalence of finite-valued tree transducers is decidable. *Math. Systems Theory*, 27:285–346, 1994.

[SEJK08] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM '08*, pages 207–218. ACM, 2008.

[SSMH04] Helmu Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. Counting in trees for free. In *Automata, Languages and Programming: 31st International Colloquium*, LNCS 3142, pages 1136–1149, 2004.

[VB12a] Margus Veanes and Nikolaj Bjørner. Symbolic automata: The toolkit. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 472–477, Berlin, Heidelberg, 2012. Springer-Verlag.

[VB12b] Margus Veanes and Nikolaj Bjørner. Symbolic tree transducers. In *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, pages 377–393, Berlin, Heidelberg, 2012. Springer-Verlag.

[VB15] Margus Veanes and Nikolaj Bjørner. Symbolic tree automata. *Information Processing Letters*, 115(3):418 – 424, 2015.

[VBNB14] Margus Veanes, Nikolaj Bjørner, Lev Nachmanson, and Sergey Bereg. Monadic decomposition. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 628–645, 2014.

[vdBHKO02] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, July 2002.

[Vea13] Margus Veanes. Applications of symbolic finite automata. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, volume 7982 of *Lecture Notes in Computer Science*, pages 16–23. Springer Berlin Heidelberg, 2013.

[VHL$^+$12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. Symbolic finite state transducers: Algorithms and applications. *SIGPLAN Not.*, 47(1):137–150, January 2012.

[VMML15] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 139–152, 2015.

[vNG01] Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, January 1988.

[Wal07] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

[Wat99] Bruce W. Watson. Implementing and using finite automata toolkits. In *Extended finite state models of language*, pages 19–36, New York, NY, USA, 1999. Cambridge University Press.

[Web93] Andreas Weber. Decomposing finite-valued transducers and deciding their equivalence. *SIAM J. Comput.*, 22(1):175–202, February 1993.

[Yu97] Sheng Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1997.