Programming Using Automata and Transducers

Loris D'antoni
University of Pennsylvania, lorisdan@seas.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/edissertations

Part of the Computer Sciences Commons

Recommended Citation
http://repository.upenn.edu/edissertations/1677

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/edissertations/1677
For more information, please contact libraryrepository@pobox.upenn.edu.
Programming Using Automata and Transducers

Abstract
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.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Rajeev Alur

Keywords
Automata, Programming languages, String transformations, Symbolic automata, Transducers, Tree transformations

Subject Categories
Computer Sciences

This dissertation is available at ScholarlyCommons: http://repository.upenn.edu/edissertations/1677
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

Supervisor of Dissertation

Rajeev Alur, Zisman Family Professor of Computer and Information Science

Graduate Group Chairperson

Lyle H. Ungar, Professor of Computer and Information Science

Dissertation Committee

Chaired by Sampath Kannan, Henry Salvatori Professor of Computer and Information Science

Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science

Val Tannen, Professor of Computer and Information Science

Margus Veanes (External), Senior Researcher at Microsoft Research
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

Acknowledgements iii
Abstract iv
List of tables xi
List of figures xiii

1 Introduction 1
   1.1 Preamble ............................................. 1
   1.2 Automata, languages, and program properties ............... 2
   1.3 Transducers, transformations, and programs ................ 3
   1.4 Limitations of existing models .......................... 5
   1.5 Contributions ........................................ 6
       1.5.1 Foundational results ............................. 6
       1.5.2 Language design and implementation ................. 7
       1.5.3 Applications ...................................... 7
   1.6 Acknowledgements ...................................... 8
## Transducer-based programming languages

### 2 BEX: a language for verifying string coders

#### 2.1 Introduction

- **2.1.1 Challenges in verifying real-world coders**
- **2.1.2 Contributions**

#### 2.2 Verifying BASE64 in BEX

#### 2.3 Symbolic extended finite automata and transducers

- **2.3.1 Preliminaries**
- **2.3.2 Model definitions**
- **2.3.3 From BEX to S-EFTs**
- **2.3.4 Cartesian S-EFAs and S-EFTs**
- **2.3.5 Monadic S-EFAs and S-EFTs**

#### 2.4 Properties of symbolic extended finite automata

#### 2.5 Equivalence of symbolic extended finite transducers

- **2.5.1 Equivalence of S-EFTs is undecidable**
- **2.5.2 Equivalence of Cartesian S-EFTs is Decidable**

#### 2.6 Composition of symbolic extended finite transducers

- **2.6.1 S-EFTs are not closed under composition**
- **2.6.2 A practical algorithm for composing S-EFTs**

#### 2.7 Experiments and applications

- **2.7.1 Analysis of string encoders**
- **2.7.2 Deep packet inspection**
- **2.7.3 Verification of list manipulating programs**

#### 2.8 Related work

- **2.8.1 From S-FA/S-FT to S-EFA/S-EFT**
3 Fast: a transducer-based language for manipulating trees

3.1 Introduction

3.1.1 Limitations of current models

3.1.2 Contributions

3.2 Overview of Fast

3.3 Symbolic tree transducers and Fast

3.3.1 Background

3.3.2 Alternating symbolic tree automata

3.3.3 Symbolic tree transducers with regular look-ahead

3.3.4 The role of regular look-ahead

3.3.5 Operations on automata and transducers

3.4 Composition of S-TTRs

3.4.1 Composition algorithm

3.4.2 Proof of correctness

3.5 Evaluation

3.5.1 HTML sanitization

3.5.2 Conflicting augmented reality applications

3.5.3 Deforestation

3.5.4 Analysis of functional programs

3.5.5 CSS analysis

3.6 Related work

3.6.1 Tree transducers

3.6.2 Symbolic transducers
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6.3</td>
<td>DSLs for tree manipulation</td>
<td>80</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Applications</td>
<td>81</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Open problems</td>
<td>82</td>
</tr>
<tr>
<td>II</td>
<td>Executable models for hierarchical data</td>
<td>83</td>
</tr>
<tr>
<td>4</td>
<td>Symbolic visibly pushdown automata</td>
<td>84</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Existing models</td>
<td>84</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Contributions</td>
<td>85</td>
</tr>
<tr>
<td>4.2</td>
<td>Motivating example</td>
<td>86</td>
</tr>
<tr>
<td>4.3</td>
<td>Model definition</td>
<td>88</td>
</tr>
<tr>
<td>4.4</td>
<td>Closure Properties and Decision Procedures</td>
<td>90</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Closure Properties</td>
<td>91</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Decision Procedures</td>
<td>95</td>
</tr>
<tr>
<td>4.5</td>
<td>Applications and Evaluation</td>
<td>96</td>
</tr>
<tr>
<td>4.5.1</td>
<td>XML Validation</td>
<td>97</td>
</tr>
<tr>
<td>4.5.2</td>
<td>HTML Filters</td>
<td>98</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Runtime Program Monitors</td>
<td>100</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Experimental results</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Streaming tree transducers</td>
<td>104</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>104</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Existing models of tree transducers</td>
<td>104</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Contributions</td>
<td>105</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Organization</td>
<td>107</td>
</tr>
<tr>
<td>5.2</td>
<td>The features of streaming tree transducers</td>
<td>108</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5.3</td>
<td>Model definition</td>
<td>109</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Nested words</td>
<td>109</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Nested words with holes</td>
<td>110</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Nested word expressions</td>
<td>110</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Single use restriction</td>
<td>111</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Transducer definition</td>
<td>112</td>
</tr>
<tr>
<td>5.3.6</td>
<td>STT syntax</td>
<td>112</td>
</tr>
<tr>
<td>5.3.7</td>
<td>STT semantics</td>
<td>113</td>
</tr>
<tr>
<td>5.4</td>
<td>Examples</td>
<td>114</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Reverse</td>
<td>115</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Tag-based sorting</td>
<td>115</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Conditional swap</td>
<td>115</td>
</tr>
<tr>
<td>5.5</td>
<td>Properties and variants</td>
<td>116</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Regular nested-word languages</td>
<td>119</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Copyless STTs</td>
<td>120</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Bottom-up transducers</td>
<td>122</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Regular look-ahead</td>
<td>126</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Multi-parameter STTs</td>
<td>134</td>
</tr>
<tr>
<td>5.5.6</td>
<td>Closure under composition</td>
<td>141</td>
</tr>
<tr>
<td>5.6</td>
<td>Restricted inputs</td>
<td>151</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Mapping strings</td>
<td>152</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Mapping ranked trees</td>
<td>157</td>
</tr>
<tr>
<td>5.7</td>
<td>Restricted outputs</td>
<td>161</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Mapping nested words to strings</td>
<td>161</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Mapping nested words to ranked trees</td>
<td>162</td>
</tr>
<tr>
<td>5.8</td>
<td>Expressiveness</td>
<td>163</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.8.1</td>
<td>MSO for nested-word transductions</td>
<td>163</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Macro tree transducers</td>
<td>168</td>
</tr>
<tr>
<td>5.8.3</td>
<td>MSO equivalence</td>
<td>170</td>
</tr>
<tr>
<td>5.9</td>
<td>Decision problems</td>
<td>173</td>
</tr>
<tr>
<td>5.9.1</td>
<td>Output analysis</td>
<td>173</td>
</tr>
<tr>
<td>5.9.2</td>
<td>Functional equivalence</td>
<td>175</td>
</tr>
<tr>
<td>5.10</td>
<td>Related work</td>
<td>179</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Executable models</td>
<td>179</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Regular look-ahead</td>
<td>180</td>
</tr>
<tr>
<td>5.10.3</td>
<td>From SSTs to STTs</td>
<td>180</td>
</tr>
<tr>
<td>5.10.4</td>
<td>Succinctness</td>
<td>181</td>
</tr>
<tr>
<td>5.10.5</td>
<td>Input/output encoding</td>
<td>181</td>
</tr>
<tr>
<td>5.10.6</td>
<td>Streaming algorithms</td>
<td>182</td>
</tr>
<tr>
<td>5.10.7</td>
<td>Complexity of checking equivalence</td>
<td>182</td>
</tr>
<tr>
<td>5.10.8</td>
<td>Alphabet limitations</td>
<td>183</td>
</tr>
</tbody>
</table>

III Conclusion 184

6 Future work 185

6.1 Declarative languages 185

6.2 Algorithms 186

6.3 New models 186

6.4 Data-parallelism 186

6.5 New applications 187

6.5.1 Direct memory access in device drivers 187

6.5.2 Networking 187
List of Tables

2.1 Analysed coders with corresponding look-aheads and running times. . 42
3.1 Summary of domain specific languages for manipulating trees. . . . . 81
4.1 Properties of different automata models. . . . . . . . . . . . . . . 86
List of Figures

1.1 Examples of finite automata. ........................................ 2
1.2 Functional programs that can be modeled using transducers. .... 4
1.3 Examples of finite state transducers. .............................. 5

2.1 Role of string encoders and decoders. ............................. 11
2.2 BASE64 encoding ................................................... 12
2.3 BASE64 analysis in BEX ........................................... 14
2.4 Running times for equivalence/inequality checking ................. 43

3.1 Implementation and analysis of an HTML sanitizer in FAST. .... 50
3.2 Encoding of &lt;div id='e"&gt;&lt;script&gt;a/&lt;/script&gt;&lt;/div&gt;&lt;br /&gt; .................................. 51
3.3 Result of applying sani to the tree in Figure 3.2 ................. 52
3.4 Syntax of FAST. ...................................................... 53
3.5 Tree languages in FAST ........................................... 55
3.6 Depiction of an S-TT linear rule of rank 3 ......................... 60
3.7 Conditional flip. ..................................................... 61
3.8 S-TTs for which the composition is naturally expressible by an S-TTR ... 62
3.9 Non-composable S-TTR-definable transformations .................. 64
3.10 Augmented reality: running times for operations on transducers .... 75
3.11 Deforestation advantage for a list of 4,096 integers .................. 77
3.12 Analysis of functional programs in FAST ......................... 78

4.1 Example of S-VPA over the theory of integers ...................... 90
4.2 Example XML Schema with an equivalent S-VPA .................... 97
4.3 S-VPAs for HTML filtering ....................................... 98
4.4 S-VPA W accepting well-formed HTML documents ................... 99
4.5 Time to filter an HTML file ....................................... 101
4.6 Running times for S-VPA equivalence and determinization ........ 103

5.1 Depiction of the transformation tree swap ......................... 108
5.2 Example of parameter tree update ................................ 140
5.3 Example of symbolic variable assignment .......................... 157
5.4 Encoding of an STT’s computation in MSO ......................... 165
“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. 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].

1.2 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...
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 $T$, and two automata $I$ and $O$, it is decidable whether for every string accepted by the automaton $I$, the transducer $T$ 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 $\text{is\_empty}$ (i.e. the empty list).
Introduction

1 (* Types for DNA bases and sequences *)
2 type base = A | T | G | C
3 type dnaseq = list base
4 (* True for any DNA sequence l *)
5 let is_dna (l: dnaseq) : bool = true
6 (* True for empty sequence *)
7 let is_empty (l: dnaseq) : bool =
8     match l with
9     [] -> true
10    | _ -> false
11 (* True if all the elements of the list l are T or A *)
12 let rec all_TG (l: dnaseq) : bool =
13     match l with
14     [] -> true
15    | h::t -> (h = T || h = G) && (all_TG t)
16 (* True if all the elements of the list l are even *)
17 let rec all_AC (l: dnaseq) : bool =
18     match l with
19     [] -> true
20    | h::t -> (h = A || h = C) && (all_AC t)
21 (* Replaces each base with the matching one *)
22 let rec map_base (l: dnaseq) : dnaseq =
23     match l with
24     [] -> []
25    | A::t -> T::(map_base t)
26    | T::t -> A::(map_base t)
27    | G::t -> C::(map_base t)
28    | C::t -> G::(map_base t)
29 (* Removes all the T and G elements from a list l *)
30 let rec filter.AC (l: dnaseq) : dnaseq =
31     match l with
32     [] -> []
33    | A::t -> A::(filter.AC t)
34    | C::t -> C::(filter.AC t)
35    | T::t -> filter.AC t
36    | G::t -> filter.AC t
37 (* Applies map_base then filter.AC to l *)
38 let m_f_DNA (l: dnaseq) : dnaseq =
39     filter.AC (map_base l)
40 (* Applies filter.AC then map_base then filter.AC to l *)
41 let f_m_f_DNA (l: dnaseq) : dnaseq =
42     m_f_DNA (filter.AC l)
43 (* Reverses a list *)
44 let rec reverse (l: dnaseq): dnaseq =
45     match l with
46     [] -> []
47    | h::t -> (reverse t) @ [h]

Figure 1.2: Functional programs that can be modeled using 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, KP94], 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
Introduction

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 \textit{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, \textit{BEX} and \textit{FAST}, that act as frontend for automata and transducer models. \textit{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 \textit{UTF8} and \textit{BASE64} encoders and decoders using less than 50 lines of code. \textit{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. \textit{BEX} is also available as a runnable web interface at \url{http://rise4fun.com/Bex/}. \textit{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 \cite{DMB08}. These include theories over bit-vectors, integers, reals, and data-types. \textit{FAST} also supports Boolean operations over languages, language equivalence, transducer composition, and regular type-checking. \textit{FAST} is also available as a runnable interface at \url{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 \url{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 \textit{BEX} to program efficient versions of \textit{BASE64} and \textit{UTF8} encoders and decoders that make heavy use of bit-level manipulations. Using \textit{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 Mollner [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. For example, the invalid sequence \[C0_{16}, AF_{16}\] (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 exist in Apache Tomcat (≤ 6.0.18), where “%C0%AE” has been used for encoding ‘.’. Further attacks have used double-encoding, where the input string is encoded twice to violate some of the decoder assumptions.

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:
\[
E \circ D \equiv I \land D \circ E \equiv I
\]

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.

<table>
<thead>
<tr>
<th>Text content</th>
<th>ASCII</th>
<th>M</th>
<th>a</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit pattern</td>
<td>77 (4D_{16})</td>
<td>97 (61_{16})</td>
<td>110 (6E_{16})</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>19</td>
<td>22</td>
<td>5</td>
<td>46</td>
</tr>
<tr>
<td>BASE64</td>
<td>F</td>
<td>W</td>
<td>F</td>
<td>u</td>
</tr>
</tbody>
</table>

**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;
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 \[a–zA–Z0–9+/\]. 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, s1, s2)’ stands for ‘if \(b\) then \(s1\) else \(s2\)’. 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
// Character maps
function em(x) := ite(x <= 25, x + 65, ite(x <= 51, x + 71,
    ite(x <= 61, x - 4, ite(x == 62, ‘+’, ‘/’))));
function dm(x) := ite(x == ‘/’, 63, ite(x == ‘+’, 62,
    ite(x == ‘9’, x + 4, ite(x == ‘Z’, x - 65, x - 71))));

// Base64 encoder
program base64encode := repeat {
    \[0–\times F F\] \{3\}  \Rightarrow \{ em(bits(7, 2, #0)),
    \{ em((bits(1, 0, #0) \& \& 4)\&\& 74, #1)),
    em((bits(3, 0, #1) \& \& 2)\&\& 76, #2)),
    em(bits(5, 0, #2))\};
    \[0–\times F F\] \{2\}$  \Rightarrow \{ em(bits(7, 2, #0)),
    em((bits(1, 0, #0) \& \& 4)\&\& 74, #1)),
    em(bits(3, 0, #1) \& \& 2, ‘=’); \}
    \[0–\times F F\] \{1\}$  \Rightarrow \{ em((bits(1, 0, #0) \& \& 4)\&\& 74, #1)),
    em((bits(1, 0, #0) \& \& 4)\&\& 76, #2),
    em((bits(1, 0, #0) \& \& 4)\&\& 76, #2),
    \}

// Base64 decoder
program base64decode := repeat {
    [a–zA–Z0–9+/] \{4\}  \Rightarrow \{ dm(#0) \& \& 2\&\& 70, #2),
    (bits(3, 0, dm(#1)) \& \& 4)\&\& 70, #2),
    (bits(1, 0, dm(#2)) \& \& 6)\&\& 72, #2); \}
    [a–zA–Z0–9+/] \{3\}$  \Rightarrow \{ dm(#0) \& \& 2\&\& 70, #2),
    (bits(3, 0, dm(#1)) \& \& 4)\&\& 70, #2),
    (bits(3, 0, dm(#1)) \& \& 4)\&\& 70, #2); \}
    [a–zA–Z0–9+/] \{2\}$  \Rightarrow \{ dm(#0) \& \& 2\&\& 70, #2),
    \}

// Identity
program identity \{ \[0–\times F F\] \Rightarrow [\#0]; \}
// Check correctness
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 \( \lambda \)-terms to represent anonymous functions. A Boolean \( \lambda \)-term \( \lambda x.\varphi(x) \), where \( x \) is a variable of type \( \sigma \) is called a \( \sigma \)-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 \( \tau \). We write \( \Sigma \) for \( U_{\sigma} \) and \( \Gamma \) for \( U_{\gamma} \).

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

For \( \sigma \)-predicates \( \varphi \), we assume an effective \emph{witness} function \( \mathcal{W} \) such that, if \( \text{IsSat}(\varphi) \) then \( \mathcal{W}(\varphi) \in \llbracket \varphi \rrbracket \), where \( \llbracket \varphi \rrbracket \subseteq \mathcal{U}^\sigma \) is the set of all values that satisfy \( \varphi \); \( \varphi \) is \emph{valid}, \( \text{IsValid}(\varphi) \), when \( \llbracket \varphi \rrbracket = \mathcal{U}^\sigma \).

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 \emph{extended} S-FTs or S-EFTs.

\textbf{Definition 2.1.} A Symbolic Extended Finite Transducer (S-EFT) with input type \( \sigma \) and output type \( \gamma \) 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
  - \( \Delta \) is a set of transitions \( r = (p, \ell, \varphi, f, q) \), denoted \( p \xrightarrow{\varphi/f} \ell q \), where
    - \( p \in Q \) is the start state of \( r \);
    - \( \ell \geq 1 \) is the look-ahead of \( r \);
    - \( \varphi \), the guard of \( r \), is a \( \sigma^{\ell} \)-predicate;
    - \( f \), the output of \( r \), is a \( \lambda \)-terms from \( \sigma^{\ell} \) to sequences of \( \gamma \);
    - \( q \in Q \) is the continuation state of \( r \);
  - \( F \) is a set of finalizers \( r = (p, \ell, \varphi, f) \), denoted \( p \xrightarrow{\varphi/f} \ell \bullet \), with components as above and where \( \ell \) 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 \( e \) is a Symbolic Extended Finite Automaton (S-EFA).

A finalizer is a rule without a continuation state. A finalizer with look-ahead \( \ell \) is used when the end of the input sequence has been reached with exactly \( \ell \) 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 \( \text{Int} \) then that symbol would have to be outside \( \text{Int} \).

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

\[
[p \xrightarrow{\phi/f} q] \overset{\text{def}}{=} \{ p \mid [a_0, \ldots, a_{\ell-1}]_{\ell} \to q \mid (a_0, \ldots, a_{\ell-1}) \in [\phi] \}.
\]

The notation \( p \xrightarrow{[a_0, \ldots, a_{\ell-1}]_{\ell}} q \) 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 \( \ell \) reads \( \ell \) 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 \( [R] \overset{\text{def}}{=} \bigcup_{r \in R} [r] \). We write \( s_1 \cdot s_2 \) for the concatenation of sequences \( s_1 \) and \( s_2 \).

**Definition 2.2.** For \( u \in \Sigma^*, v \in \Gamma*, q \in Q, q' \in Q \cup \{ \cdot \} \), define \( q \xrightarrow{u/v}_A q' \) as follows: there exists \( n \geq 0 \) and \( \{ p_i, u_i/v_i \mid p_{i+1} \mid p_i \mid i \leq n \} \subseteq [R] \) 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 \( q \xrightarrow{\epsilon/\cdot}_A q \) for all \( q \in Q \).

**Definition 2.3.** The transduction relation of \( A \) is defined as \( \mathcal{J}_A(u) \overset{\text{def}}{=} \{ v \mid q^0 \xrightarrow{u/v}_A \cdot \} \).

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 \( \lambda \)'s. We write

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

where \( \phi \) 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-EFT’s `Base64encode` and `Base64decode` corresponding to the BEX programs `base64encode` and `base64decode` presented in Figure 2.3.
Definition 2.5. A function \( f : X \to 2^Y \) is single-valued if \( |f(x)| \leq 1 \) for all \( x \in X \). An S-EFT \( A \) is single-valued if \( \mathcal{J}_A \) is single-valued.

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

\(^4\)For more information see http://www.rise4fun.com/Bek/tutorial/base64.
\[ \lambda(x_1, \ldots, x_{\text{max}(m,n)}) \cdot \varphi(x_1, \ldots, x_m) \land \psi(x_1, \ldots, x_n). \] We define equivalence of \( f \) and \( g \) with respect to \( \varphi \), \( f \equiv g \), as: 
\[ \text{IsValid}(\lambda \bar{x}. (\varphi(\bar{x}) \Rightarrow f(\bar{x}) = g(\bar{x}))). \]

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

(a) Assume \( q, q' \in Q \). If \( \text{IsSat}(\varphi \land \varphi') \) then \( q = q', \ell = \ell' \) and \( f \equiv \varphi \land \varphi' \).

(b) Assume \( q = q' = \bullet \). If \( \text{IsSat}(\varphi \land \varphi') \) and \( \ell = \ell' \) then \( f \equiv \varphi \land \varphi' \).

(c) Assume \( q \in Q \) and \( q' = \bullet \). If \( \text{IsSat}(\varphi \land \varphi') \) 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 \( f : X \to 2^Y \) is \( \mathcal{D}(f) \overset{\text{def}}{=} \{ x \in X \mid f(x) \neq \emptyset \} \) and for an S-EFT \( A \), \( \mathcal{D}(A) \overset{\text{def}}{=} \mathcal{D}(\mathcal{T}_A) \). When \( A \) is single-valued and \( u \in \mathcal{D}(A) \), we treat \( A \) as a partial function from \( \Sigma^* \) to \( \Gamma^* \) and write \( A(u) \) for the value \( v \) such that \( \mathcal{T}_A(u) = \{ v \} \). For example, \( \text{Base64encode}("Foo") = "Rm9v" \) and \( \text{Base64decode}("QmFy") = "Bar" \).

### 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 \( v_0 \).

```
program name := [q = v_0] repeat {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

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

where \( \text{guard} \) 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 \( \text{update} \) 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

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

where \( L(\text{guard}) \) is the set of strings accepted by the regular expression \( \text{guard} \). The programs in Figure 2.2 only have one state and the state component is therefore implicit.
The second form is
\[ \text{guard} \in, q = v_1 \Rightarrow \text{update} \]
where \textit{guard} is a regular expression that accepts only strings with some finite length \( l \) and does not contain the symbol \( \in \), and \textit{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
\[ v_1 \lambda x_1, \ldots, x_l : (x_1 \cdots x_l) \in L(\text{guard})/\lambda \#_0, \ldots, \#_{l-1}. \text{update} \]

### 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 \( \sigma^\infty \)-predicates for \( n \geq 2 \) in the obvious way. In order to decide if a satisfiable \( \sigma^\infty \)-predicate \( \varphi \) is Cartesian over \( \sigma \), let \( (a_0, \ldots, a_{n-1}) = \mathcal{W}(\varphi) \) and perform the following validity check:

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

In other words, a \( \sigma^\infty \)-predicate \( \varphi \) is Cartesian over \( \sigma \) if \( \varphi \) can be rewritten equivalently as a conjunction of \( n \) independent \( \sigma \)-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 \( \text{IsCartesian}(\varphi) \) is decidable by using the decision procedure of the label theory. Namely, decide unsatisfiability of \( \neg \text{IsCartesian}(\varphi) \).

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 \( \Delta \) has look-ahead 2 and every finalizer has look-ahead 0. For every rule \( r = p \xrightarrow{\varphi(x_1) \land \varphi(x_2)} q \), the S-FA \( A' \) has a 3 states \( q, p, q_r \) and two rules
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 $x < y \mod 2$ has an MNF $(x < 0 \land y \mod 2 = 0) \lor (x < 1 \land y \mod 2 = 1)$ that is also a DNF. Another MNF of $x < y \mod 2$ is $x < 0 \lor (x < 1 \land y \mod 2 = 1)$, 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 $\phi$ 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 $p \xrightarrow{\phi/f} q$ by the rules $p \xrightarrow{\phi_i/f} q$, for $i < n$, where all $\phi_i$ 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: $\text{INC}_i$ (increment $r_i$ and continue with the next instruction); $\text{DEC}_i$ (decrement $r_i$ if $r_i > 0$ and continue with the next instruction); $\text{JZ}_i(j)$ (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 ($\text{program counter}, r_1, r_2$) 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 $\sigma$ as follows. Let $\varphi^{\text{ini}}$ be the $\sigma$-predicate $\lambda x.(x = (0,0,0))$ stating that the program counter and both registers are 0. Let $\varphi^{\text{fin}}$ be the final $\sigma$-predicate $\lambda x.(\pi_0(x) = |P| \land \pi_1(x) \neq 0)$.

Let $\varphi^{\text{step}}$ be the $\sigma^2$-predicate $\lambda (x, x'). \forall i < |P| \varphi^{\text{step}}_i$ where $\varphi^{\text{step}}_i$ is the formula for the $i$th instruction. If the $i$th instruction is $\text{INC}_1$ then $\varphi^{\text{step}}_i$ is

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

If the $i$th instruction is $\text{JZ}_1(j)$ then $\varphi^{\text{step}}_i$ is

$$\pi_0(x) = i \land \pi_0(x') = \text{Ite}(\pi_1(x) = 0, j, i + 1) \land \pi_1(x') = \pi_1(x) \land \pi_2(x') = \pi_2(x).$$

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

$$A = (\{p_0\}, p_0, \{p_0 \xrightarrow{\varphi^{\text{step}}} p_0, \quad p_0 \xrightarrow{\text{true}} 0 \}) \quad \text{and} \quad B = (\{q_0, q_1\}, q_0, \{q_0 \xrightarrow{\varphi^{\text{ini}}} q_1, \quad q_1 \xrightarrow{\varphi^{\text{step}}} q_1, \quad q_1 \xrightarrow{\varphi^{\text{fin}}} 0\}).$$

So $\alpha \in \mathcal{D}(A) \cap \mathcal{D}(B)$ iff $\alpha$ 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 $\mathcal{D}(A) \cap \mathcal{D}(B) \neq \emptyset$ 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. \qed
**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 $A_1 = (Q_1, q_1^0, R_1)$ and $A_2 = (Q_2, q_2^0, R_2)$ over a sort $\sigma$ we construct an S-EFA $B$ over $\sigma$ such that $\mathcal{D}(C) = \mathcal{D}(A) \cup \mathcal{D}(B)$. $C$ 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:

$$R = R_1 \cup R_2 \cup \{q_0 \xrightarrow{\varphi} q \mid q_0^1 \xrightarrow{\varphi} q \in R_1 \lor q_0^2 \xrightarrow{\varphi} q \in R_2\}.$$  

As a consequence of Theorems 2.11 and 2.12 we have that S-EFAs 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. \hfill $\square$

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 $\sigma$ it is undecidable to check whether $A$ accepts all the sequences in $\sigma^*$, 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 (program counter, $r_1$, $r_2$) of $M$. Let $\varphi^{\text{ini}}, \varphi^{\text{fin}}$ and $\varphi^{\text{step}}$ be as in Theorem 2.11.

We construct an S-EFA $A_M$ that does not accept all the strings in $\sigma^*$ 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

\[ A = \{(p_0, p_1), p_0, \{p_0 \xrightarrow{\text{true}} p_0, p_0 \xrightarrow{\neg \phi_{\text{step}}/2} p_1, p_1 \xrightarrow{\text{true}} p_1, p_1 \xrightarrow{\text{true}} \ast}\}\), \]

\[ B = \{(q_0, q_1), q_0, \{q_0 \xrightarrow{\neg \phi_{\text{ini}}/1} q_1, q_1 \xrightarrow{\text{true}} q_1, q_1 \xrightarrow{\text{true}} \ast}\}\), \]

\[ C = \{(r_0), r_0, \{r_0 \xrightarrow{\text{true}} r_0, r_0 \xrightarrow{\neg \phi_{\text{fin}}/1} \ast}\}\), \]

\[ D = \{(s_0), s_0, \{s_0 \xrightarrow{\text{true}} \ast}\}\). \]

\(A\) 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 \(\sigma^*\) 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 \(A_{\text{all}}\) that accepts all strings we have that being able to check the equivalence of \(A_{\text{all}}\) 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 \(\varphi_1(x_1, \ldots, x_i)\) and \(\varphi_2(x_{i+1}, \ldots, x_{k+1})\). We now must have that \(\psi(x_1, \ldots, x_{k+1}) \equiv \forall \varphi_1(x_1, \ldots, x_i) \land \varphi_2(x_{i+1}, \ldots, x_{k+1})\). However the predicate \(\psi\) does not admit such a representation as the variables \(x_i\) and \(x_{i+1}\) do not belong to any binary relation and cannot be compared for equality. \(\square\)

### 2.5 Equivalence of symbolic extended finite transducers

While the general equivalence problem of \(\mathcal{T}_A = \mathcal{T}_B\) is already undecidable for very restricted classes of finite state transducers \([\text{Gri68}]\), 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 \( f, g : X \to 2^Y \) are one-equal, \( f \overset{1}{=} g \), if for all \( x \in X \), if \( x \in D(f) \cap D(g) \) then \( |f(x) \cup g(x)| = 1 \). Let

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

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

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

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

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 \( B \) (\( \mathcal{T}_A = \mathcal{T}_B \)).

### 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:

\[
 A'_i = (Q_{A_i}, q^0_{A_i}, \Delta_{A_i} \cup \{ p \frac{\varphi/\overline{i}}{k} \cdot \mid p \frac{\varphi}{k} \cdot \in F_{A_i} \}).
\]

So \( \mathcal{T}_{A'_i}(t) = \{ [i] \} \) if \( t \in \mathcal{D}(A_i) \) and \( \mathcal{T}_{A'_i}(t) = \emptyset \) otherwise. Let \( f = \mathcal{T}_{A'_1} \uplus \mathcal{T}_{A'_2} \). So

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

It follows that \( A'_1 \overset{1}{=} A'_2 \) iff (by Proposition 2.17) \( f \) is single-valued iff \( \mathcal{D}(A_1) \cap \mathcal{D}(A_2) = \emptyset \). Now use Theorem 2.11.
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) \cite{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. \cite{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 $\phi^{\text{step}}$ 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 $\epsilon$). The SLT $A$ outputs $\epsilon$ 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 $\text{true}$ that always outputs $\epsilon$. 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. \cite{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. \cite{VHL+12, Theorem 1}. We use the following definitions. A transition, $p \xrightarrow{\phi/f \sim} q$ where $\ell > 1$, $\phi$ is Cartesian and $\mathcal{W}(\phi) = (a_1, \ldots, a_\ell)$, is represented, given $\phi_i = \lambda x.\phi(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_\ell)$, by the following path of split transitions,

$$p \xrightarrow{\phi_1/f \sim} p_1 \xrightarrow{\phi_2/\bot \sim} p_2 \cdots p_{\ell-1} \xrightarrow{\phi_{\ell}/\bot \sim} q$$

where $p_i$ for $1 \leq i < \ell$ are new temporary states. Let $\Delta^*_{A}$ denote such split view of $\Delta_A$. 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 • with a new state \( p_1 \) and add the new finalizer \( p_1 \rightarrow_{true/\varepsilon} \bullet \).

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 \( q^0_A \times q^0_B \) of \( A \times B \) is \( \langle q^0_A, q^0_B \rangle \). The states and transitions of \( A \times B \) are obtained as the least fixed point of

\[
\begin{align*}
(p,q) &\in Q_{A \times B} \\
p \xrightarrow{\phi/f} p' \in \Delta_A^s &\quad \text{IsSat}(\phi \land \psi) \quad (p',q') \in Q_{A \times B}, \quad (p,q) \xrightarrow{\psi/g/f} (p',q') \in \Delta_{A \times B}.
\end{align*}
\]

Let \( F_{A \times B} \) be the set of all rules \( (p,q) \xrightarrow{true/(v,w)} \bullet \) such that \( p \xrightarrow{true/v} \bullet \in F_A, q \xrightarrow{true/w} \bullet \in F_B, \) 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).

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 \( \xRightarrow{A \times B} \) is defined analogously to S-EFTs.

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

\[
(p,q) \xrightarrow{u/(v,w)}_{A \times B} \bullet \quad \iff \quad p \xrightarrow{u/v}_{A} \bullet \land q \xrightarrow{u/w}_{B} \bullet
\]

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 \( \varepsilon \), 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 \( p',q',u',v',w' \),

\[
(p',q') \xrightarrow{u'/(v',w')}_{A \times B} \bullet \quad \iff \quad p' \xrightarrow{u'/v'}_{A} \bullet \land q' \xrightarrow{u'/w'}_{B} \bullet
\]

If they exists, let \( p \xrightarrow{\phi/f} p_1 \) and \( q \xrightarrow{\psi/g} q_1 \) 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 \( (p,q) \xrightarrow{\phi \land \psi} (p_1,q_1) \). Since \( a_1 \ldots a_l \) is a model of both \( \phi \) and \( \psi \), it is also a model of \( \phi \land \psi \). 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. \( \square \)

\(^{5}\) In our constructions we always remove unnecessary states to improve the efficiency of our algorithms.
We define also, for all $u \in \Sigma^*$, $\mathcal{F}_{A \times B}(u) \overset{\text{def}}{=} \{(v, w) \mid q_0^0 \xrightarrow{u/(v,w)} \bullet\}$ and $\mathcal{D}(A \times B) \overset{\text{def}}{=} \mathcal{D}(\mathcal{F}_{A \times B})$. Lemma 2.22 implies that $\mathcal{D}(A \times B) = \mathcal{D}(A) \cap \mathcal{D}(B)$ and $A \not\sim B$ iff there exists $u$ and $v \neq w$ such that $(v, w) \in \mathcal{F}_{A \times B}(u)$.

Next we prove an alignment lemma that allows us either to effectively eliminate all non-aligned pair-states from $A \times B$ without affecting $\mathcal{F}_{A \times B}$ or to establish that $A \not\sim B$. A product S-EFT is aligned if all pair-states in it are aligned.

**Lemma 2.23** (Alignment). If $A \not\sim B$ 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 $A \not\sim B$, 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_0^0\}. 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 $\ell = 2$),

   $$(p, q) \xrightarrow{\phi/(f, g)} (p_1, q_1) \xrightarrow{\psi/(\bot, \bot)} (p_2, q_2),$$

   replace the path with the following combined transition with look-ahead 2

   $$(p, q) \xrightarrow{\lambda(x_0, x_1) \phi(x_0) \wedge \psi(x_1)/(f, g)} (p_2, q_2)$$

   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 $\sigma^k/\gamma$-sequence and the $B$-output $g$ is a $\sigma^\ell/\gamma$-sequence ($k \neq \ell$, say $k = 2$ and $\ell = 1$):

   $$(p, q) \xrightarrow{\phi/(f, g)} (p_1, q_1) \xrightarrow{\psi/(\bot, \bot)} (p_2, q_2).$$

   So $p_1$ is temporary while $q_1$ is not.

   Decide if $f$ can be split into two independent $\sigma/\gamma$-sequences $f_1$ and $f_2$ such that for all $a_1 \in \llbracket \phi \rrbracket$ and $a_2 \in \llbracket \psi \rrbracket$, $\llbracket f \rrbracket(a_1, a_2) = \llbracket f_1 \rrbracket(a_1) \cdot \llbracket f_2 \rrbracket(a_2)$. 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 $f_1 =$
\( \lambda x. h_1(x, \mathcal{H}(\psi)), f_2 = \lambda x. h_2(\mathcal{H}(\phi), x) \) and check validity of the split predicate

\[ \forall x y \left((\varphi(x) \land \psi(y)) \Rightarrow f(x, y) = f_1(x) \cdot f_2(y) \right). \]

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

\[ (p, q) \xrightarrow{\varphi/(f_1, q_1)} (p_1', q_1') \xrightarrow{\psi/(f_2, q_1)} (p_2, q_2) \]

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

Suppose that splitting fails. We show that \( A \nRightarrow B \), by contradiction. Assume \( A \nRightarrow B \).

Since splitting fails, the following dependency predicates are satisfiable:

\[ D_1 = \lambda (x, x', y). (\varphi(x) \land \varphi(x') \land \psi(y) \land f(x, y) \neq f(x', y)), \]
\[ D_2 = \lambda (x, y, y'). (\varphi(x) \land \psi(y) \land \psi(y') \land f(x, y) \neq f(x, y')). \]

Let \( \langle a_1, a_1', a_2 \rangle = \mathcal{H}(D1) \) and \( \langle e_1, e_2, e_2' \rangle = \mathcal{H}(D2) \). 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

\[ [b] = \llbracket f \rrbracket(a_1, a_2), \quad [b'] = \llbracket f \rrbracket(a_1', a_2), \quad [d] = \llbracket f \rrbracket(e_1, e_2), \quad [d'] = \llbracket f \rrbracket(e_1, e_2'). \]

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 \( IsSat(D1) \),

\[
\begin{align*}
    p_0 &\xrightarrow{a/u_1} p \xrightarrow{[a_1, a_2]/[b]} p_2 \xrightarrow{\beta/u_2} A \bullet & \implies & u_1 \cdot [b] \cdot u_2 = & v_1 \cdot \llbracket \gamma \rrbracket(a_1) \cdot v_2, \\
    q_0 &\xrightarrow{a/v_1} q \xrightarrow{[a_1]/[\gamma](a_1)} q_1 \xrightarrow{[a_2]/\gamma v_2} & & & & \implies & u_1 \cdot [b'] \cdot u_2 = & v_1 \cdot \llbracket \gamma \rrbracket(a_1') \cdot v_2, \\
    p_0 &\xrightarrow{a/u_1} p \xrightarrow{[a_1', a_2]/[b']} p_2 \xrightarrow{\beta/u_2} A \bullet & \implies & u_1 \cdot [b'] \cdot u_2 = & v_1 \cdot \llbracket \gamma \rrbracket(a_1') \cdot v_2, \\
    q_0 &\xrightarrow{a/v_1} q \xrightarrow{[a_1']/[\gamma](a_1')} q_1 \xrightarrow{[a_2']/\gamma v_2} & & & & \implies & u_1 \cdot [d'] \cdot u_2 = & v_1 \cdot \llbracket \gamma \rrbracket(e_1) \cdot v_4. \\
\end{align*}
\]

By \( b \neq b', |v_1| \leq |u_1| < |v_1| \cdot \llbracket \gamma \rrbracket(a_1) = |v_1| + |\gamma| \). Also, by \( IsSat(D2) \),

\[
\begin{align*}
    p_0 &\xrightarrow{a/u_1} p \xrightarrow{[e_1, e_2]/[d]} p_2 \xrightarrow{\beta/u_2} A \bullet & \implies & u_1 \cdot [d] \cdot u_2 = & v_1 \cdot \llbracket \gamma \rrbracket(e_1) \cdot v_3, \\
    q_0 &\xrightarrow{a/v_1} q \xrightarrow{[e_1]/[\gamma](e_1)} q_1 \xrightarrow{[e_2]/\gamma v_3} & & & & \implies & u_1 \cdot [d'] \cdot u_2 = & v_1 \cdot \llbracket \gamma \rrbracket(e_1) \cdot v_4. \\
\end{align*}
\]
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 establish that \(A\) by contradiction.

\[b \in [f_1] (e_1, e_2), b' = [f_1] (e_1, e_2), b_2 = [f_2] (a_1, a_2), b'_2 = [f_2] (a_1', a_2)\]

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

\[
\begin{align*}
p_0 & \xrightarrow{a/u_1} p \xrightarrow{[a_1,a_2]/[b_1]} p_2 \xrightarrow{\beta/u_2} A \bullet \quad \Longrightarrow \quad u_1 \cdot [\cdot, b_2] \cdot u_2 = v_1 \cdot \|g\| (a_1) \cdot v_2 \\
q_0 & \xrightarrow{a'/u_1} q \xrightarrow{[a_1]/\|g\| (a_1)} q_1 \xrightarrow{[a_2]/\beta/v_2} B \bullet \quad \Longrightarrow \quad u_1 \cdot [\cdot, b'_2] \cdot u_2 = v_1 \cdot \|g\| (a_1') \cdot v_2 \\
p_0 & \xrightarrow{a'/u_1} p \xrightarrow{[a_1,a_2]/[b_1]} p_2 \xrightarrow{\beta/u_2} A \bullet \quad \Longrightarrow \quad u_1 \cdot [\cdot, b_2] \cdot u_2 = v_1 \cdot \|g\| (a_1) \cdot v_2 \\
q_0 & \xrightarrow{a'_1/u_1} q \xrightarrow{[a_1]/\|g\| (a_1)} q_1 \xrightarrow{[a_2]/\beta/v_2} B \bullet \quad \Longrightarrow \quad u_1 \cdot [\cdot, b'_2] \cdot u_2 = v_1 \cdot \|g\| (a_1') \cdot v_2 \\
\end{align*}
\]

Since \(b_2 \neq b'_2\) it must be that \(|u_1| + 1 < |v_1 \cdot \|g\| (a_1)| = |v_1| + |g|\) Also,

\[
\begin{align*}
p_0 & \xrightarrow{a'/u_1} p \xrightarrow{[c_1,c_2]/[b_1]} p_2 \xrightarrow{\beta/u_2} A \bullet \quad \Longrightarrow \quad u_1 \cdot [b_1, \_] \cdot u_2 = v_1 \cdot \|g\| (e_1) \cdot v_3 \\
q_0 & \xrightarrow{a_1/u'_1} q \xrightarrow{[c_1]/\|g\| (e_1)} q_1 \xrightarrow{[c_2]/\beta/v_3} B \bullet \quad \Longrightarrow \quad u_1 \cdot [b'_1, \_] \cdot u_2 = v_1 \cdot \|g\| (e_1) \cdot v_4 \\
p_0 & \xrightarrow{a'/u_1} p \xrightarrow{[c_1,c_2]/[b_1]} p_2 \xrightarrow{\beta/u_2} A \bullet \quad \Longrightarrow \quad u_1 \cdot [b_1, \_] \cdot u_2 = v_1 \cdot \|g\| (e_1) \cdot v_3 \\
q_0 & \xrightarrow{a_1/u'_1} q \xrightarrow{[c_1]/\|g\| (e_1)} q_1 \xrightarrow{[c_2]/\beta/v_3} B \bullet \quad \Longrightarrow \quad u_1 \cdot [b'_1, \_] \cdot u_2 = v_1 \cdot \|g\| (e_1) \cdot v_4 \\
\end{align*}
\]

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 \(A \neq B\). \(\square\)
Assume \( A \times B \) is aligned and let \([A \times B]\) be the following product S-FT (product S-EFT all of whose transitions have look-ahead 1) over the input type \( \sigma^* \). For each

\[
\frac{\lambda x, \varphi(x_0, x_1, \ldots, x_{\ell-1})/(f, g)}{t} \quad q \text{ in } \Delta_{A \times B}
\]

let \( y \) be a variable of sort \( \sigma^* \) and let \( \varphi_1 \) be the \( \sigma^* \)-predicate

\[
\lambda y. \varphi(y[0], y[1], \ldots, y[\ell - 1]) \land \text{tail}^f(y) = \varepsilon \land \text{tail}^g(y) \neq \varepsilon
\]

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

**Lemma 2.24** (Grouping). Assume \( A \times B \) is aligned. For all \( u \in \Sigma^* \) and \( v, w \in \Gamma^* \): \((v, w) \in \mathcal{T}_{A \times B}(u)\) iff \( \exists z(u = [z] \land (v, w) \in \mathcal{T}_{A \times B}(z)) \).

**Proof.** The type lifting does not affect the semantics of the label-theory specific transformations. \( \square \)

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 \((v, w) \in \mathcal{T}_{A \times B}(u)\). 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 \((v, w) \in \mathcal{T}_C(u)\) 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 \( \mathcal{T}(A \times B) = \mathcal{T}(A) \cap \mathcal{T}(B) \) and \( A \neq B \) iff there exist \( u \) and \( v \neq w \) such that \((v, w) \in \mathcal{T}_{A \times B}(u)\). By using the Alignment lemma 2.23 construct aligned product S-FT \( C \) such that \( \mathcal{T}_C = \mathcal{T}_{A \times B} \) or else determine that \( A \neq B \). Now lift \( C \) to \([C]\), and by using the Grouping lemma 2.24 \( A \neq B \) iff there exist \( u \) and \( v \neq w \) such that \((v, w) \in \mathcal{T}_{[C]}(u)\).
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 $f : X \rightarrow 2^Y$ and a set of inputs $x \subseteq X$, the lifting of function $f$ to a set of inputs $x$ is defined as $f(x) \overset{\text{def}}{=} \bigcup_{x \in x} f(x)$. Given two lifted functions $f : X \rightarrow 2^Y$ and $g : Y \rightarrow 2^Z$, we define their composition $f \circ g(x) \overset{\text{def}}{=} g(f(x))$. This definition follows the convention from Fülöp et al. [FV98], i.e., $\circ$ applies first $f$, then $g$, contrary to how $\circ$ is used for standard function composition. The intuition is that $f$ corresponds to the relation $R_f : X \times Y$, $R_f \overset{\text{def}}{=} \{(x, y) \mid y \in f(x)\}$, so that $f \circ g$ corresponds to the binary relation composition $R_f \circ R_g \overset{\text{def}}{=} \{(x, z) \mid \exists y (R_f(x, y) \land R_g(y, z))\}$.

**Definition 2.28.** A class of transducers $C$ is closed under composition iff for every $\mathcal{R}_1$ and $\mathcal{R}_2$ that are $C$-definable $\mathcal{R}_1 \circ \mathcal{R}_2$ 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 $\text{Int} \rightarrow \text{Int}$

$$A = (\{q\}, q, \{q \xrightarrow{\text{true}/[x_1, x_0]} \frac{x_1 + x_0}{2}, q \xrightarrow{\text{true}/0} \bullet\})$$
and $B$ be following S-EFT over $\mathbb{Int} \rightarrow \mathbb{Int}$

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

The two transformations behave as in the following examples:

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

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

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

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

Intuitively, looking at $\mathcal{T}_{A \circ B}$ we can see that no finite look-ahead seems to suffice for this function. Formally, for each $a_i$ such that $i \geq 0$, $\mathcal{T}_{A \circ B}$ 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 $\mathcal{T}_{A \circ B}$. We consider the S-EFT $C$ with minimal look-ahead (let’s say $n$) that computes $\mathcal{T}_{A \circ B}$.

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 $T_{A \circ B}$ is not S-EFT-definable. □

**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 $f = T_{A \circ B}$ 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

$\begin{align*}
A &= \{ (p_0, p_0, \frac{\text{step}}{[x_0, x_1]} 2, p_0, \frac{\text{true}}{0} ) \}, \\
B &= \{ (q_0, q_1), q_0, \{ q_0, \frac{\text{step}}{[x_0]} q_1, q_1, \frac{\text{step}}{[x_0, x_1]} q_1, q_1, \frac{\text{fin}}{[x_0]} \} \}.
\end{align*}$

This construction ensures that $\alpha \in D(A \circ B)$ iff $\alpha$ 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 $D(A \circ B) = \{ \alpha \}$ iff there exists $a$, such that $M$ halts on $a$ or $D(A \circ B) = \emptyset$ otherwise. In the first case we will have $T_{A \circ B}(\alpha) = \alpha$ and undefined on any input different from $a$. In the second case $T_{A \circ B}$ is always undefined. In both cases $T_{A \circ B}$ is S-EFT definable. Let’s call $C$ the S-EFT that implements $T_{A \circ B}$. 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. □
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. Botičan 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 $a_1 \ldots a_n$ 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 $a_1 \ldots a_n$ 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 \[\text{VHL}^{+12}\].

**Definition 2.32.** A Symbolic Transducer or S-T over \(\sigma \rightarrow \gamma\) and register type \(\tau\) 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;
- \(\rho^0 \in U^T\) is the initial register value;
- \(R\) is a finite set of rules \(R = \Delta \cup F\);
- \(\Delta\) 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 \(r\);
  - \(\varphi\), 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 \(r\);
  - \(\varphi\), the guard of \(r\), is a \(\tau\)-predicate;
  - \(o\), the output of \(r\), is a finite sequence of \(\tau / \gamma\)-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 \(\tau\) be \(\sigma^*\) that records previously seen characters, where initially \(\rho^0 = []\), indicating that no input characters have been seen yet.

An S-EFT transition

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

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

\[
p \xrightarrow{(\lambda(x,y).true) / \epsilon; \lambda(x,y).cons(x,[])} p_1 \\
p_1 \xrightarrow{(\lambda(x,y).true) / \epsilon; \lambda(x,y).cons(x,y)} p_2 \\
p_2 \xrightarrow{(\lambda(x,y).\varphi(y[1],y[0],x)) / \lambda(x,y).o(y[1],y[0],x); \lambda(x,y).[]} q
\]
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 \((p, q, o, u, q) \in R\) denotes the following set of concrete transitions:

\[
\{(p, s) \overset{a/[o]((a,s))}{\rightarrow} (q, [u]) (a,s)) \mid (a,s) \in \llbracket q \rrbracket \}.
\]

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

\[
\{(p, s) \overset{\epsilon/[o](s)}{\rightarrow} \bullet \mid s \in \llbracket q \rrbracket \}.
\]

The reachability relation \( p \overset{a/b}{\rightarrow}_A q \) for \( a \in \Sigma^*, b \in \Gamma^* \), \( p \in (Q \times U^T) \), \( q \in (Q \times U^T) \cup \{\bullet\} \) is defined analogously to S-EFTs and \( \mathcal{F}_A(a) \overset{\text{def}}{=} \{b \mid (q^0, \rho^0) \overset{a/b}{\rightarrow} \bullet\} \).

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 \( p \overset{\lambda x.q(x)}/\lambda x.o(x)}{\rightarrow}_k q \), where \( |x| = k \), by \( p \overset{q(x)}/o(x)}{\rightarrow}_k q \), and an S-T rule \( p \overset{\lambda x.y.q(x,y)}/\lambda x.y.o(x,y)\lambda x.y.u(x,y)}{\rightarrow} q \) by \( p \overset{q(x,y)}/o(x,y)u(x,y)}{\rightarrow} q \).

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

\[
q \overset{\text{true}/[x_1,x_0,x_0]}{\rightarrow}_2 q, \quad q \overset{\text{true}/[]}{}{\rightarrow}_0 \bullet
\]

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

\[
q \overset{\text{true}/[]}{}{\rightarrow}_x p, \quad p \overset{\text{true}/[x,y,y]}{\rightarrow}_0 q, \quad q \overset{\text{true}/[]}{}{\rightarrow}_x \bullet
\]

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 \( A^{st} \circ A^{st} \). The register of \( A^{st} \circ A^{st} \) 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

\[
q_0 \overset{\text{true}/[x,y_1]}{\rightarrow} q_1, \quad q_1 \overset{\text{true}/[y_0,x,y_0]}{\rightarrow} q_2, \quad q_2 \overset{\text{true}/[x,y_1]}{\rightarrow} q_3, \quad q_3 \overset{\text{true}/[x,y_1,y_0,y_0,y_0]}{\rightarrow} q_0, \quad q_0 \overset{\text{true}/[]}{}{\rightarrow}_x \bullet
\]
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 \( p \xrightarrow{\lambda(x,y) \cdot o(x,y) \cdot u(x,y)} q \) by \( p \xrightarrow{\phi(x,y); o(x,y); u(x,y)} q \).

#### Input: S-T \( A^{\sigma/\gamma: \tau} \).

#### Output: \( \bot \) or an S-EFT over \( \sigma \rightarrow \gamma \) that is equivalent to \( A \).

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

\[
p' \xrightarrow{\mathbf{[]} \land \phi(\text{head}(x),y) \cdot o(\text{head}(x),y) \cdot (\text{head}(x),y)} q.
\]

Intuitively, \( x \) must be a non-empty list, i.e., input \( \mathbf{[]} \) 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 \( p_1 \xrightarrow{\phi_1; o_1; u_1} (k) \xrightarrow{(\tau)} p_2 \) in \( R \):

- let \( \phi = \lambda(x,y) . \phi_1(x,y) \land \phi_2(\text{tail}^k(x), u_1(x,y)) \),
- let \( o = \lambda(x,y) . o_1(x,y) \cdot o_2(\text{tail}^k(x), u_1(x,y)) \),
- let \( u = \lambda(x,y) . u_2(\text{tail}^k(x), u_1(x,y)) \),
- if IsSat(\( \phi \)) then add \( p_1 \xrightarrow{\phi; o; u} (k+\tau) \xrightarrow{} p_2 \) 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 ⊥.

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 \([\text{cons}(a_0, _), \text{cons}(a_1, _), \text{cons}(a_2, _)]\) and produces output \(v\), where the tails _ are unconstrained and irrelevant. Step 2 further groups the inputs characters, e.g., to \([\text{cons}(a_0, \text{cons}(a_1, _)), \text{cons}(a_2, _)]\), 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 \(\sigma\), 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 \(\varphi\) 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 \(\sigma^*\) so that a lifted character is a list (sequence of type \(\sigma^*\)). We let \(|x| > k\) abbreviate the formula \(\bigwedge_{i=0}^{k} \text{tail}^k(x) \neq []\). 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:

\[
\begin{align*}
q_0 & \xrightarrow{x \neq [[]]\,(x_0,y_1)} (1) q_1, \\
q_1 & \xrightarrow{x \neq [[]]\,(y_0,x_0,y_0;0)} (1) q_2, \\
q_2 & \xrightarrow{x \neq [[]]\,(x_0,y_1)} (1) q_3, \\
q_3 & \xrightarrow{x \neq [[]]\,(x_0,y_1,y_0,d_0,y_0;0)} (1) q_0, \\
q_0 & \xrightarrow{\text{true}/[]} (1) \bullet,
\end{align*}
\]

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: 

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

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:

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

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:

\[ q_0 \xrightarrow{|x|>3/\{x_0,x_1,x_2\};\{x_2,x_0\};\{0,0\}} q_0. \]

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

\[ q_0 \xrightarrow{\text{true}/\{x_0,x_1,x_2\};\{x_2,x_0\};\{0,0\}} q_0, \quad q_0 \xrightarrow{0} \bot. \]

where the lifting has been undone and here each variable \( x_i \) is of type \( \sigma \). 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 \( B^{st} \circ A^{st} \), after lifting the input type, is then the following S-T (recall that \( B^{st} \circ A^{st}(w) = A^{st}(B^{st}(w)) \)):

If we apply the register elimination algorithm to this S-T, it may first eliminate the state \( q_2 \), by creating the transition \( q_1 \xrightarrow{|x|>1/\{x_0\};\{0,x_0\}} q_1 \). After this step the algorithm stops and returns \( \bot \).
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 $B^{st} \circ A^{st}$, after lifting the input type, is then the following S-T, which is very similar to the one in Example 2.35:

If we apply the register elimination algorithm to this S-T, it may, again, first eliminate the state $q_2$, by creating the transition $q_1 \xrightarrow{(x \neq \text{nil} \land x_0 = 0)} q_1$. 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 $\bot$.

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 $B$: 

\[
\begin{align*}
(x \neq \text{nil} \land x_0 \neq 0) / [ ]; (x_0, y_1) & \quad (x \neq \text{nil}) / [x_0, y_1]; (0, y_0) \\
(0, 0) \xrightarrow{(x \neq \text{nil}) / [ ]; (0, x_0)} q_1 (x \neq \text{nil} \land x_0 = 0) / [x_0, y_1]; (0, 0) & \quad q_3
\end{align*}
\]
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: $E \circ D \downarrow I$ and $D \circ E \downarrow I$ (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.
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 $1 \leq i \leq 9$, Figure 2.4(a);
- **Inequivalence for Enc/Dec**: $E^{i+1} \circ D^i \not\equiv I$ for $1 \leq i \leq 9$, Figure 2.4(a);
- **Equivalence for Dec/Enc**: $D^i \circ E^i \overset{1}{=} I$ for $1 \leq i \leq 3$, Figure 2.4(b);
- **Inequivalence for Dec/Enc**: $D^i \circ E^{i+1} \not\equiv I$ for $1 \leq i \leq 3$, 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 $E^{i+1} \circ D^i \not\equiv I$.

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.
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.

**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

\[
x1::x2::xs \rightarrow (x1+x2)::(f2 \; xs) \quad \text{and} \quad x1::x2::x3::xs \rightarrow (x1+x2+x3)::(f3 \; xs)
\]

can be naturally expressed as S-EFT transitions. Let’s consider two functions \(f_2\) and \(f_3\), both of type \(\text{list int} \rightarrow \text{list int}\), 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

\[
\forall l. f_3(f_2 l) = f_2(f_3 l)
\]

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 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 Botičan 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) \cite{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 \cite{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 \cite{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 \cite{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 \cite{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. 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 \texttt{sani} (Figure 3.1, line 30) traverses the input DOM and outputs a copy where

\footnote{Some sanitizers process the input HTML as a string, often causing the output not to be standards compliant.}
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.

```plaintext
// Datatype definition for HTML encoding
type HtmlE[tag : String] = {nil(), val(1), attr(2), node(3)}

// Language of well-formed HTML trees
lang nodeTree : HtmlE {
  node(x_1, x_2, x_3) given (attrTree x_1) (nodeTree x_2) (nodeTree x_3)
  | nil() where (tag = "") }

lang attrTree : HtmlE {
  attr(x_1, x_2) given (valTree x_1) (attrTree x_2)
  | nil() where (tag = "") }

lang valTree : HtmlE {
  val(x_1) where (tag ≠ "") given (valTree x_1)
  | nil() where (tag = "") }

// Sanitization functions
trans remScript : HtmlE → HtmlE {
  node(x_1, x_2, x_3) where (tag ≠ "script") to
    (node [tag] x_1 (remScript x_2) (remScript x_3))
  | node(x_1, x_2, x_3) where (tag = "script") to x_3
  | nil() to (nil [tag]) }

trans esc : HtmlE → HtmlE {
  node(x_1, x_2, x_3) to (node [tag] (esc x_1) (esc x_2) (esc x_3))
  | attr(x_1, x_2) to (attr [tag] (esc x_1) (esc x_2))
  | val(x_1) where (tag = "" ∨ tag = "").) to (val ["" ] (val [tag] (esc x_1)))
  | val(x_1) where (tag ≠ "" ∧ tag ≠ "") to (val [tag] (esc x_1))
  | nil() to (nil [tag]) }

// Compose remScript and esc, restrict to well-formed trees
def rem_esc : HtmlE → HtmlE := (compose remScript esc)
def sani : HtmlE → HtmlE := (restrict rem_esc nodeTree)

// Language of bad outputs that contain a "script" node
lang badOutput : HtmlE {
  node(x_1, x_2, x_3) where (tag = "script")
  | node(x_1, x_2, x_3) given (badOutput x_2)
  | node(x_1, x_2, x_3) given (badOutput x_3) }

// Check that no input produces a bad output
def bad_inputs : HtmlE := (pre-image sani badOutput)
assert-true (is-empty bad_inputs)
```

Figure 3.1: Implementation and analysis of an HTML sanitizer in Fast.
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 \( \text{node}(x_1, x_2, x_3) \) 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) \( \text{tag} \) contains the node type of \( n \) (div, etc.). Each HTML attribute \( a \) with value \( s \) is encoded as an HtmlE element \( \text{attr}(x_1, x_2) \) 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) \( \text{tag} \) contains the name of \( a \) (id, etc.). Each non-empty string \( w = s_1 \ldots s_n \) is encoded as an HtmlE element \( \text{val}(x_1) \) where \( \text{tag} \) contains the string “\( s_1 \)”, and \( x_1 \) encodes the suffix \( s_2 \ldots s_n \). 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 \( \text{node}(x_1, x_2, x_3) \) 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 \( t = \text{node}(x_1, x_2, x_3) \) and its tag is different from "script", \( \text{remScript} \) outputs a copy of \( t \) in which \( x_2 \) and \( x_3 \) are replaced by the results of invoking \( \text{remScript} \) on \( x_2 \) and \( x_3 \) respectively; 2) if \( t = \text{node}(x_1, x_2, x_3) \) and its tag is equal to "script", \( \text{remScript} \) outputs a copy of \( x_3 \), 3) if \( t = \text{nil} \), \( \text{remScript} \) 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 \( \text{val} \) with tag "' " or "" " is pre-pended a node \( \text{val} \) with tag "\"". The transformations \( \text{remScript} \) and \( \text{esc} \) are then composed into a single transformation \( \text{rem_esc} \) (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 \( \text{rem_esc} \) also accepts input trees that are not in the language nodeTree and do not correspond to correct encodings. Therefore, we compute the transformation sani
Sanitizing the tree of Figure 3.2 with the function \textit{sani} yields the \textit{HtmlE} tree corresponding to \texttt{<div id='e\"'></div><br />} (see Figure 3.3).

We can now use \textsc{Fast} to analyze the program \textit{sani}. First, we define the language \textit{bad_output} (lines 29-32), which accepts all the trees containing at least one node with tag "script"\footnote{This definition illustrates the nondeterministic semantics of \textsc{Fast}: a tree \( t \) belongs to \textit{bad_output} if at least one of the three rules applies.}. Next, using pre-image computation, we compute the language \textit{bad_inputs} (line 34) of inputs that produce a bad output. Finally, if \textit{bad_inputs} is the empty language, \textit{sani} never produces bad outputs. When running this program in \textsc{Fast} this checking (line 35) fails, and \textsc{Fast} provides the following counterexample:

\begin{verbatim}
node ["script"] nil nil (node ["script"] nil nil nil)
\end{verbatim}

where we omit the attribute for the \textit{nil} nodes. This is due to a bug in line 17, where the rule does not recursively invoke the transformation \textit{remScript} on \( x_3 \). After fixing this bug the assertion becomes valid. In this example, we showed how in \textsc{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 \textsc{Fast}

The concrete syntax of \textsc{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 \cite{HWG03}. Additional well-formedness conditions (such as well-typed terms) are assumed to hold. \textsc{Fast} is designed for describing trees, tree languages, and functions from trees to trees. These are supported using symbolic
Fast: a transducer-based language for manipulating trees

Indicators

| Identifier ID | \(a..z|A..Z|_.|0..9^\ast\) |
| Basic types \(\sigma: \text{String} | \text{Int} | \mathbb{R} | \mathbb{B} \ldots\) |
| Built-in operators \(\text{op}<|>||=|+|\text{and}|\text{or}|\ldots\) |
| Constructors \(c: \text{ID}\) Natural numbers \(k: \mathbb{N}\) |
| Tree types \(\tau: \text{ID}\) Language states \(p: \text{ID}\) |
| Transformation states \(q: \text{ID}\) Attribute fields \(x: \text{ID}\) |
| Subtree variables \(y: \text{ID}\) |

**Main definitions:**

\[
\text{Fast ::= type } \tau \quad \{ (x: \sigma)^\ast \} \quad \{ (c (k))^\ast \} \quad \{ \text{tree } t : \tau := \text{TR} \}
\]

\[
\text{lang } p : \tau \quad \{ \text{Lrule}^+ \} \quad \text{trans } q : \tau \rightarrow \tau \quad \{ \text{Trule}^+ \}
\]

\[
\text{def } p : \tau := \text{L} \quad \text{def } q : \tau \rightarrow \tau := \text{T}
\]

\[
\text{assert-true } A \quad \text{assert-false } A
\]

\[
\text{Lrule ::= } c(y_1,\ldots,y_n) \quad (\text{where } \text{Aexp})? \quad (\text{given } (p \quad y)) +?\?
\]

\[
\text{True ::= } \text{Lrule to Tout}
\]

\[
\text{Tout ::= } y \quad (q \quad y) \quad (c [Aexp^+] \quad \text{Tout}^+)
\]

\[
\text{Aexp ::= } \text{ID} \quad \text{Const} \quad (\text{op } \text{Aexp}^+)
\]

**Operations over languages, transductions, and trees:**

\[
\text{L ::= (intersect } L \quad \text{L}) \quad (\text{union } L \quad \text{L}) \quad (\text{complement } L) \quad (\text{difference } L \quad L) \quad (\text{L}) \quad (\text{domain } T)
\]

\[
\text{p} \quad (\text{pre-image } T \quad \text{L}) \quad (\text{p})
\]

\[
\text{T ::= (compose } T \quad T) \quad (\text{restrict } T \quad L) \quad (\text{restrict-out } T \quad L) \quad q
\]

\[
\text{TR ::= } t \quad (c [Aexp^+] \quad \text{TR}^+ \quad (\text{apply } T \quad \text{TR}) \quad (\text{get-witness } L)
\]

\[
\text{A ::= } \text{L} \quad \text{== } \text{L} \quad (\text{is-empty } L) \quad (\text{is-empty } T) \quad (\text{TR } \text{in } T)
\]

\[
\text{Aexp ::= } (\text{type-check } L \quad T \quad L)
\]

**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 $\lambda$-expressions for defining anonymous functions called $\lambda$-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. $[\text{VHL}^{+}12]$. We write $f(v)$ for the functional application of the $\lambda$-term $f$ to the term $v$, $\sigma$ for a type, and $\mathcal{U}^\sigma$ for the universe of elements of type...
\(\sigma\). A \(\sigma\)-predicate is a \(\lambda\)-term \(\lambda x. \varphi(x)\) where \(x\) has type \(\sigma\) and \(\varphi\) is a formula for which the free variables \(FV(\varphi)\) are contained in \(\{x\}\). Given a \(\sigma\)-predicate \(\varphi\), \([\varphi]\) denotes the set of all \(a \in U^{\sigma}\) such that \(\varphi(a)\). The set of \(\sigma\)-predicates is denoted by \(\Psi(\sigma)\). Given a type \(\sigma\) (such as \(\text{int}\)), we extend the universe with \(\sigma\)-labeled finite trees as an algebraic datatype \(T^{\sigma}_U\) where \(\Sigma\) is a finite set of tree constructors \(f\) with rank \(R(f) \geq 0\); \(f\) has type \(\sigma \times (T^{\sigma}_U)^{R(f)} \rightarrow T^{\sigma}_U\)\(^3\). Let \(\Sigma(k) \overset{\text{def}}{=} \{f \in \Sigma \mid R(f) = k\}\). We call \(T^{\sigma}_U\) a tree type. We require that \(\Sigma(0)\) is non-empty so that \(U^{T^{\sigma}_U}\) is non-empty. We write \(f[a](\bar{a})\) for \(f(a, \bar{a})\) and abbreviate \(f[a]()\) by \(f[a]\).

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

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 \((e_i)_{i=1}^k\) for \(\bar{e}\). The empty tuple is () and \((e_i)_{i=1}^1 = e_1\). We use the following operations over \(k\)-tuples of sets. If \(\bar{X}\) and \(\bar{Y}\) are \(k\)-tuples of sets then \(\bar{X} \uplus \bar{Y} \overset{\text{def}}{=} (X_i \cup Y_i)_{i=1}^k\). 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 \((i = j \text{ then } X_i \cup Y \text{ else } X_i)_{i=1}^k\).

### 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 Veane et al. \(\text{[VB12b]}\). 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 \((Q, T^{\sigma}_U, \delta)\), where \(Q\) is a finite set of states, \(T^{\sigma}_U\) is a tree type, and \(\delta \subseteq \bigcup_{k \geq 0} (Q \times \Sigma(k) \times \Psi(\sigma) \times (2^{\Sigma(k)})^k)\) is a finite set of rules \((q, f, \varphi, \bar{\ell})\), where \(q\) is the source state, \(f\) the symbol, \(\varphi\) the guard, and \(\bar{\ell}\) the look-ahead.

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

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

Next, we define the semantics of an S-TA \(A = (Q, T^{\sigma}_U, \delta)\).

\(^3\text{If } R(f) = 0 \text{ then } f \text{ has type } \sigma \rightarrow T^{\sigma}_U.\)
FAST: a transducer-based language for manipulating trees

1  type $BT[i : \text{Int}] \{ L(0), N(2) \}$
2  lang $p : BT \{$
3        $L() \text{ where } (i > 0)$
4        $| \ N(x, y) \text{ given } (p x) (p y)$
5        $\}$
6  lang $o: BT \{$
7        $L() \text{ where } (\text{odd } i)$
8        $| \ N(x, y) \text{ given } (o x) (o y)$
9        $\}$
10 lang $q: BT \{ \ N(x, y) \text{ given } (p y) (o y) \}$

**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

$$L^q_A \overset{\text{def}}{=} \{ f[a](\overline{t}) \in UT^q_{\overline{\sigma}} \mid (q, f, \varphi, \overline{\ell}) \in \delta, a \in \llbracket \varphi \rrbracket, \bigwedge_{i=1}^{R(f)} \bigwedge_{p \in \ell_i} t_i \in L^p_A \}.$$ 

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

$$L^q_A \overset{\text{def}}{=} \begin{cases} 
\bigcap_{q \in q} L^q_A, & \text{if } q \neq \emptyset; \\
U T^q_{\overline{\sigma}}, & \text{otherwise.} 
\end{cases}$$

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

\[
\begin{align*}
\{ (p, L, \lambda x. x > 0, ()), (p, N, \lambda x. \text{true}, (\emptyset), (p)), \} , \\
\{ (o, L, \lambda x. \text{odd}(x), ()), (o, N, \lambda x. \text{true}, (\emptyset), (o)) , \} , \\
\{ (q, N, \lambda x. \text{true}, (\emptyset), (p, o)) \} .
\end{align*}
\]

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 $(p, f, \varphi, \overline{\ell}) \in \delta$ and all $i$ with $1 \leq i \leq R(f)$, $\ell_i$ 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 \( A = (Q, T^E_A, \delta) \) be an S-TA. We compute merged rules \((q, f, \varphi, \bar{\rho})\) over merged states \( q \in 2^Q \), where \( \bar{\rho} \in (2^Q)^{R(f)} \). For \( f \in \Sigma \), let \( \delta^f = \bigcup_{p \subseteq Q} \delta^f(p) \), where

\[
\delta^f(\emptyset) = \{(\emptyset, f, \varnothing, (\varnothing)_{i=1}^{R(f)})\}, \\
\delta^f(p \cup q) = \{r \land s \mid r \in \delta^f(p), s \in \delta^f(q)\}, \\
\delta^f(\{p\}) = \{((\{p\}, f, \{\varphi\}, \bar{\rho})) \mid (p, f, \varphi, \bar{\rho}) \in \delta\},
\]

and where merge operation \( \land \) over merged rules is defined as follows:

\[
(p, f, \varphi, \bar{\rho}) \land (q, f, \psi, \bar{\omega}) \overset{\text{def}}{=} (p \cup q, f, \varphi \cup \psi, \bar{\rho} \cup \bar{\omega}).
\]

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

\[
\mathcal{N}(A) \overset{\text{def}}{=} (2^Q, T^E_A, \{(p, f, \bigwedge \varphi, (\{q_i\})_{i=1}^{R(f)}) \mid f \in \Sigma, (p, f, \varphi, \bar{\rho}) \in \delta^f\}).
\]

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 \( \varphi \) 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 \[6\] p. 88–89.

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

**Theorem 3.7.** For all \( q \subseteq Q \), \( L_A^q = L_{\mathcal{N}(A)}^q \).

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

\[
t \in L_A^q \iff t \in L_{\mathcal{N}(A)}^q.
\]
The proof is by induction over the height of \( t \). As the base case assume \( t = f[a] \) where \( R(f) = 0 \). Then

\[
f[a] \in L_A^q \quad \iff \quad \forall q \in \mathbf{q}(f[a] \in L_A^q)
\]

\[
\iff \quad \forall q \in \mathbf{q}(\exists \varphi((q, f, \varphi, ())) \in \delta, a \in \llbracket \varphi \rrbracket)
\]

\[
\iff \quad \forall q \in \mathbf{q}(\exists \varphi((\{q\}, f, \{\varphi\}, ())) \in \delta^f(\{q\}), a \in \llbracket \varphi \rrbracket)
\]

\[
\iff \quad f[a] \in L_{N(A)}^q.
\]

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 \( t = f[a](t_1, t_2) \).

\[
t \in L_A^q \iff \quad \bigwedge_{i=1}^{2} t \in L_A^{q_i}
\]

\[
\iff \quad \bigwedge_{i=1}^{2} \exists \varphi^i, p_1^i, p_2^i : (q_i, f, \varphi^i, (p_1^i, p_2^i)) \in \delta_A, a \in \llbracket \varphi^i \rrbracket, t_1 \in L_A^{p_1^i}, t_2 \in L_A^{p_2^i}
\]

\[
\iff \quad \bigwedge_{i=1}^{2} \exists \varphi^i, p_1^i, p_2^i : (\{q_i\}, f, \{\varphi^i\}, (p_1^i, p_2^i)) \in \delta^f, a \in \llbracket \varphi^i \rrbracket, t_1 \in L_A^{p_1^i}, t_2 \in L_A^{p_2^i}
\]

\[
\iff \quad \exists \varphi, p_1, p_2 : t_1 \in L_A^{p_1}, t_2 \in L_A^{p_2}, (q, f, \varphi, (\{p_1\}, \{p_2\})) \in \delta_{N(A)}, a \in \llbracket \varphi \rrbracket
\]

\[
\iff \quad \exists \varphi, p_1, p_2 : t_1 \in L_{N(A)}^{p_1}, t_2 \in L_{N(A)}^{p_2}, (q, f, \varphi, (\{p_1\}, \{p_2\})) \in \delta_{N(A)}, a \in \llbracket \varphi \rrbracket
\]

\[
\iff \quad t \in L_{N(A)}^q.
\]

The theorem follows by the induction principle.

Checking whether \( L_A^q \neq \emptyset \) 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 \( U^\sigma = \{()\} \).

**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 \textsc{ExpTime} \cite{CDG07} Theorem 7.5.1. An alternating tree automaton is a tuple $A = (Q, \Sigma, \varphi, \Delta)$ such that $Q$ is a finite set of states, $\Sigma$ is ranked alphabet, $\varphi$ is a propositional formula with variables in $Q$ (e.g., $q_1 \lor \neg q_2$), and $\Delta$ is a transition function that maps pairs $(q, f) \in Q \times \Sigma$ of states and symbols to propositional formula with variables in $Q \times \{1, \ldots, R(f)\}$ (e.g., $(q_1, 2) \land (q_2, 1)$). 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 $A$ at state $q \in Q$, $t \in L_q$, if

$$t = f(t_1, \ldots, t_n) \text{ and } \Delta(q, f)[(p, i)/(t_i \in L_p)],$$

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 \textsc{ExpTime}, consider an S-TA $A = (Q, T_\Sigma, \delta)$ and $q \in Q$. Here $U^\varphi = \{()\}$, i.e. there are no attributes. Construct an alternating tree automaton $A = (Q, \Sigma, \{q\}, \Delta)$ over $\Sigma$ with state set $Q$, initial state $q$, and mapping $\Delta$ such that for $(q, f) \in Q \times \Sigma$,

$$\Delta(q, f) \defeq \bigvee_{(q, f, \varphi, \bar{t}) \in \delta(q)} \bigwedge_{p \in \ell_t} \bigwedge_{i \in I} (p, i).$$

Then $L(A)$ is non-empty iff $U^L_\lambda$ is non-empty. For inclusion in \textsc{ExpTime} we use Theorem 7.5.1 from the TATA book \cite{CDG07}.

For \textsc{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 $\Delta$. Instead, let $A_i = (Q_i, T_\Sigma, \delta_i)$ be top-down tree automata with initial states $q_i \in Q_i$ for $1 \leq i \leq n$ \cite{CDG07}. Consider all these automata as S-TAs without attributes and with pairwise disjoint $Q_i$. In particular, all $A_i$ are normalized. Expand $\Sigma$ to $\Sigma' = \Sigma \cup \{f\}$ where $f$ is a fresh symbol of rank 1. Let $A$ be the S-TA $(\{q\} \cup \bigcup_i Q_i, T_{\Sigma'}, \bigcup_i \delta_i \cup \{(q, f, \lambda x. \text{true}, (\{q_i\}_{1 \leq i \leq n}))\})$ where $q$ is a new state. It follows from the definitions that $U^L_\lambda \neq \emptyset$ iff $\bigcap_i U^L_i \neq \emptyset$. \textsc{ExpTime}-hardness follows now from the intersection non-emptiness problem of tree automata \cite{FSVY91} (already restricted to the top-down-deterministic case \cite{Sei94}).

\[\] 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 $T^e_\Sigma$ 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 $T^e_\Sigma$ 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 $T^e_{\Sigma \cup \{\text{State}\}}$ where $\text{State} \notin \Sigma$ is a new fixed symbol of rank 1. A term $\text{State}[q](t)$ is always used with a concrete value $q$ and $\text{State}[q]$ is also written as $\tilde{q}$.

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

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

**Definition 3.11.** A Symbolic Tree Transducer with Regular look-ahead (S-TTR) $T$ is a tuple $(Q, q^0, T^e_\Sigma, \Delta)$, where $Q$ is a finite set of states, $q^0 \in Q$ is the initial state, $T^e_\Sigma$ is the tree type, $\Delta \subseteq \bigcup_{k \geq 0} (Q \times \Sigma(k) \times \Psi_\sigma \times (2^Q)^k \times \Lambda(T^e_\Sigma, Q, k))$ is a finite set of rules $(q, f, q, \tilde{t}, t)$, where $t$ is the output. For $k = 0$ we assume that $(2^Q)^k = \{()\}$, i.e., a rule for $e \in \Sigma(0)$ 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 $\lambda(x, \tilde{y}).u$ where each $y_i$ occurs at most once in $u$. $T$ is linear when all rules of $T$ are linear.

A rule $(q, f, q, \tilde{t}, t)$ is also denoted by $q \xrightarrow{f, \varphi, \tilde{t}} t$. The open view of a rule $q \xrightarrow{f, \varphi, \tilde{t}} t$ is denoted by $\tilde{q}(f[x](\tilde{y})) \xrightarrow{\varphi[x], \tilde{t}} t(x, \tilde{y})$. 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 $\tilde{O}$ by default. Figure 3.6 illustrates an open view of a linear rule over the tree type $T^e_{\Sigma_1}$ over $\Sigma_1 = \{f, g, h\}$, where $R(f) = 2$, $R(g) = 3$, and $R(h) = 0$.

Let $T$ be an S-TTR $(Q, q^0, T^e_\Sigma, \Delta)$. The following construction is used to extract an STA from $T$ that accepts all input trees for which $T$ is defined. Let $t$ be a $k$-rank tree
transducer. For $1 \leq i \leq k$ let $St(i, t)$ denote the set of all states $q$ such that $\tilde{q}(y_i)$ occurs in $t$.

**Definition 3.12.** The domain automaton of $T$, $d(T)$, is the S-TA $(Q, T_\Sigma^c, \{(q, f, \varphi, (\ell_i \cup St(i, t))^R(f)) | q \xrightarrow{f, \varphi} t \in \Delta\})$.

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 $u = \lambda(x, y).v$, the term $u(a, \tilde{s})$ is the function application of $u$ to $(a, \tilde{s})$, where $a$ and $\tilde{s}$ substitute $x$ and $\tilde{y}$ respectively. In the following let $T$ be the S-TTR, and for $\ell \subseteq Q$, let $L^\ell_T \overset{\text{def}}{=} L^\ell_{d(T)}$.

**Definition 3.13.** For all $q \in Q_T$, the transduction of $T$ at $q$ is the function $T^q_T$ from $UT_\Sigma^c$ to $2^{UT_\Sigma^c}$ such that, for all $t = f[a](\tilde{s}) \in UT_\Sigma^c$,

$$T^q_T(t) \overset{\text{def}}{=} \downarrow_T(\tilde{q}(t)),
$$

$$\downarrow_T(\tilde{q}(t)) = \bigcup \{ \downarrow_T(u(a, \tilde{s})) | (q, f, \varphi, \tilde{s}, u) \in \Delta_T, a \in \llbracket \varphi \rrbracket, \bigwedge_{i=1}^{R(f)} s_i \in L^\ell_T \},
$$

$$\downarrow_T(t) = \{ f[a](\tilde{v}) | \bigwedge_{i=1}^{R(f)} v_i \in \downarrow_T(s_i) \}.$$

The transduction of $T$ is $T^\varnothing_T \overset{\text{def}}{=} T^q_T$. The definitions are lifted to sets using set union.

We omit $T$ from $T^q_T$ and $\downarrow_T$ when $T$ is clear from the context.

**Example 3.14.** Recall the transformation remScript in Figure 3.7 These are the corresponding rules. We use $q$ for the state of remScript, and $i$ for a state that outputs the identity transformation. The “safe” case is

$$\tilde{q}(\text{node}[x](y_1, y_2, y_3)) \overset{x \neq \text{"script"}}{\longrightarrow} \text{node}[x](\tilde{i}(y_1), \tilde{q}(y_2), \tilde{q}(y_3)),
$$

the “unsafe” case is $\tilde{q}(\text{node}[x](y_1, y_2, y_3)) \overset{x = \text{"script"}}{\longrightarrow} \tilde{i}(y_3)$, and the “harmless” case is $\tilde{q}(\text{nil}[x]()) \overset{\text{true}}{\longrightarrow} \text{nil}[x]()$. 

---

**Figure 3.6:** Depiction of an S-TT linear rule of rank 3.
In **FAST**, a transformation $T^q$ is defined by the statement

$$\text{trans } q : \tau \rightarrow \tau \{ f(\bar{y}) \text{ where } \varphi(x) \text{ given } \ell(\bar{y}) \to t(x, \bar{y}) \mid \ldots \}$$

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 $\forall (t \in \mathcal{U}^T, q \in Q_T) : |T^q_T(t)| \leq 1$.

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 $q \xrightarrow{f \varphi \bar{x}} t$ and $q \xrightarrow{f \varphi \bar{y}} u$ in $\Delta_T$, if $[\varphi] \cap [\psi] \neq \emptyset$ and, for all $i \in \{1, \ldots, R(f)\}$, $L^t_i \cap L^r_i \neq \emptyset$, 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.
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 \( L[\text{true}] \). 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 \( L[\text{true}] \) 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 \textbf{assert-true} \((\text{is-empty} \ a)\).

- Operations that compute new languages:

  \textbf{intersect} \( A_1 \ A_2 \), \textbf{complement} \( A \), etc.: operations over S-TAs \cite{VB15};

  \textbf{domain} \( T \): computes the domain of the S-TTR \( T \) using the operation from Definition 3.12.

  \textbf{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.**

\[
\text{restrict } T A = \text{compose } I A T \\
\text{restrict-out } T A \equiv \text{compose } T I A \\
\text{pre-image } T A \equiv \text{domain } (\text{restrict-out } T A) \\
\text{type-check } A_1 T A_2 \equiv \text{is-empty } (\text{intersect } A_1 (\text{pre-image } T (\text{complement } A_2)))
\]

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
FAST: a transducer-based language for manipulating trees

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 $T_{S \circ T} = T_S \circ T_T$. The composition $T_{S \circ T}$ is defined as $(T_{S \circ T})(x) = \bigcup_{y \in T_s(x)} T_T(y)$, 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

\[
\begin{align*}
(\text{true}, (\varnothing)_{i=1}^{R(f)}, \tilde{q}(p[f](\tilde{y}))) & \xrightarrow{S} (\varnothing, \tilde{q}(\_)) \xrightarrow{T} (q, \tilde{\ell}, t) \\
\end{align*}
\]

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 $p.q \xrightarrow{f, \phi, \tilde{\ell}} t$.

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

\[
(\text{true}, \tilde{\varnothing}, \tilde{q}(p[f](y_1, y_2))) \xrightarrow{S} (x \geq 0, \tilde{\varnothing}, \tilde{q}(\tilde{p}(y_2)))
\]

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

The rewriting steps are done modulo attribute constraints. To this end, a $k$-configuration is a triple $(\gamma, L, u)$ where $\gamma$ 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 $S = (Q_S, q_0^S, T_E^T, \Delta_S)$ and $T = (Q_T, q_0^T, T_E^S, \Delta_T)$, the composition of $S$ and $T$ is defined as follows

\[
S \circ T \overset{\text{def}}{=} (Q_S \cup \{ p.q \mid p \in Q_S, q \in Q_T \}, q_0^S, q_0^T, T_E^S \cup \bigcup_{p \in Q_S, q \in Q_T, f \in \Sigma} \text{Compose}(p, q, f)).
\]

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.

\[\text{Compose}(p, q, f) \overset{\text{def}}{=} \]

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

2. **choose** $(\psi, \tilde{p}, t)$ from $\text{Reduce}(\varphi, (\varnothing)_{i=1}^{R(f)}, \tilde{q}(u))$;
3. \textbf{return } (p, q, f, \psi, \bar{t} \uplus \bar{P}, t).

The procedure \textbf{Reduce} uses a procedure \textbf{Look}(\varphi, L, q, t) that, given an attribute formula \varphi 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, a singleton set. Notice that since we operate over normalized transducers, \(\text{Look}(\cdot)\) is always defined.

\textbf{Look}(\varphi, L, q, t) \overset{\text{def}}{=} \\
\begin{enumerate}
\item \textbf{if } \(t = \bar{p}(y_i)\) \textbf{where } \(p \in Q_S\) \textbf{then return } (\varphi, L \uplus \{p, q\});
\item \textbf{if } \(t = g[u_0](\bar{u})\) \textbf{where } \(g \in \Sigma\) \textbf{then}
  \hspace{1em} \begin{enumerate}
  \item \textbf{choose } \(q, g, \psi, \bar{t}\) \textbf{from } \(\delta_{d(T)}\) \textbf{where } \text{IsSat}(\varphi \land \psi(u_0));
  \item \(L_0 := L, \varphi_0 := \varphi \land \psi(u_0);\)
  \item \textbf{for } \(i = 1; i \leq R(g); \text{ } i++\)
    \hspace{1em} \begin{enumerate}
    \item \textbf{choose } \(\varphi_i, L_i\) \textbf{from } \text{Look}(\varphi_{i-1}, L_{i-1}, \text{sin}(\ell_i), u_i);
    \end{enumerate}
  \end{enumerate}
  \hspace{1em} \textbf{return } (\varphi_{R(g)}, L_{R(g)}).
\end{enumerate}

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.

\textbf{Example 3.22.} Consider the instance \textbf{Look}(x > 0, \emptyset, q, t) for \(t = g[x+1](g[x-2](\bar{p}_1(y_2)))\) where \(g \in \Sigma(1)\). Suppose there is a rule \(q, g, \lambda x.\text{odd}(x), \{q\}\) \(\in \delta_{d(T)}\) 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 \(\bar{p}(f[x](y_1, y_2)) \rightarrow g[x+1](g[x-2](\bar{p}_1(y_2))))\) 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 \textbf{Look} procedure.

In \textbf{Look}(x > 0, \emptyset, q, t) line 2(c) we have the recursive call \textbf{Look}(x > 0 \land \text{odd}(x+1), \emptyset, q, g[x-2](\bar{p}_1(y_2))). Inside the recursive call we have the failing satisfiability check of \text{IsSat}(x > 0 \land \text{odd}(x+1) \land \text{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, \emptyset, q, t) is empty.
In the following we pretend, without loss of generality, that for each rule \( \tau = (q, f, \varphi, \bar{t}, t) \) there is a state \( q_\tau \) that uniquely identifies the rule \( (q_\tau, f, \varphi, \bar{t}, t) \); \( q_\tau \) is used to refer to the guard and the look-ahead of \( \tau \) chosen in line 2(a) in the call to \text{Look} \) in 2(b) below, \( q_\tau \) is not used elsewhere.

\text{Reduce}(\gamma, L, v) \overset{\text{def}}{=} \begin{align*}
1. & \text{ if } v = \bar{q}(\bar{p}(y_i)) \text{ where } q \in Q_T \text{ and } p \in Q_S \text{ then return } \langle \gamma, L, \bar{p}\bar{q}(y_i) \rangle \\
2. & \text{ if } v = \bar{q}(g[u_0](\bar{u})) \text{ where } q \in Q_T \text{ and } g \in \Sigma \text{ then} \\
& \hspace{1em} (a) \text{ choose } \tau = (q, g, \omega, \omega, t) \text{ from } \Delta_T; \\
& \hspace{1em} (b) \text{ choose } (\gamma_1, L_1) \text{ from } \text{Look}(\gamma, L, q_\tau, g[u_0](\bar{u})); \\
& \hspace{1em} (c) \text{ choose } \chi \text{ from } \text{Reduce}(\gamma_1, L_1, t(u_0, \bar{u})) \text{ return } \chi; \\
3. & \text{ if } v = g[t_0](\bar{t}) \text{ where } g \in \Sigma \text{ then} \\
& \hspace{1em} (a) \gamma_0 := \gamma, L_0 := L; \\
& \hspace{1em} (b) \text{ for } (i = 1; i \leq R(g); i++) \\
& \hspace{2em} \text{ choose } (\gamma_i, L_i, u_i) \text{ from } \text{Reduce}(\gamma_{i-1}, L_{i-1}, t_i); \\
& \hspace{1em} (c) \text{ return } (\gamma_{R(\Sigma)}, L_{R(\Sigma)}, g[t_0](\bar{u})).
\end{align*}

There is a close relationship between \text{Reduce} and Definition 3.13. We include the case

\[ T^q_T(\bar{p}(t)) \overset{\text{def}}{=} T^q_T(T^q_S(t)) \text{ for } p \in Q_S \text{ and } t \in \mathcal{U}^T, \] (3.2)

that allows states of \( S \) to occur in the input trees to \( T^q_T \) 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 \( t \in \Lambda(T^q_S, Q_S, k), a \in \mathcal{U}, \text{ and } u_i \in \mathcal{U}^T \),

\[ T^q_T(\downarrow_S(t(a, \bar{u}))) \leq T^q_T(t(a, \bar{u})), \text{ and} \]

\[ T^q_T(\downarrow_S(t(a, \bar{u}))) = T^q_T(t(a, \bar{u})) \text{ when } S \text{ is single-valued or } T \text{ is linear.} \]
Proof. We prove statements 1 and 2 by induction over \( t \). The base case is 
\[ t = \lambda(x,y).p_i \] for some \( p \in Q_S \) and some \( i, 1 \leq i \leq k \). We have 
\[
\mathcal{F}_T^g(\psi_S(\bar{p}(u_i))) = \mathcal{F}_T^g(\mathcal{F}_S^p(u_i)) = \mathcal{F}_T^g(\bar{p}(u_i))
\]
where the last equality holds by using equation (3.2). The induction case is as follows. 
Let \( t = \lambda(x,y).f_0(t_0(x))(t_i(x,y))_{i=1}^R(f) \). Suppose \( R(f) = 1 \), the proof of the general case is analogous.

\[
\begin{align*}
\mathcal{F}_T^g(\psi_S(f[t_0(a)](t_1(a,\bar{u})))) & \overset{\text{Def}}{=} \mathcal{F}_T^g(\psi_S(f[t_0(a)](v) \mid v \in \psi_S(t_1(a,\bar{u})))) \\
\mathcal{F}_T^g & \overset{\text{Def}}{=} \text{IH} \\
\mathcal{F}_T^g & \overset{\text{IH}}{=} \mathcal{F}_T^g(\psi_S(f[t_0(a)](t_1(a,\bar{u}))))
\end{align*}
\]

The step (* \( \leq \) \) when either \( |\psi_S(t_1(a,\bar{u}))| \leq 1 \) or when \( m \leq 1 \). 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.

\[ \square \]

**Example 3.24.** The example shows a case when 

\[ T_T^g(\psi_S(f(t(a,\bar{u})))) \neq T_T^g(t(a,\bar{u})). \]

Suppose \( p \overset{c_T}{\rightarrow} \triangle, p \overset{c_T}{\rightarrow} \triangle, \) and \( q \overset{S_T}{\rightarrow} \lambda xy.f[x](\bar{q}(y),\bar{q}(y)). \) Let \( f = f[0], c = c[0], g = g[0]. \) Then 

\[
\begin{align*}
\bar{q}(g(p(c))) & \overset{\mathcal{S}}{\rightarrow} f(\bar{q}(\bar{p}(c)),\bar{q}(\bar{p}(c))) \\
& \overset{\mathcal{S}}{\rightarrow} \{ f(\bar{q}(\triangle),\bar{q}(\triangle)), f(\bar{q}(\triangle),\bar{q}(\triangle)) \} \cup \{ f(\bar{q}(\triangle),\bar{q}(\triangle)), f(\bar{q}(\triangle),\bar{q}(\triangle)) \}
\end{align*}
\]
\[ \tilde{q}(g(\tilde{p}(c))) \xrightarrow{\tilde{p}} \{ \tilde{q}(g(\triangle)), \tilde{q}(g(\bigtriangledown)) \} \]

where, for example, \( f(\tilde{q}(\bigtriangledown), \tilde{q}(\bigtriangledown)) \) 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 \( \Sigma \times U^c \) 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, \( T \circ S \) is an overapproximation of \( T \circ S \circ T \) and that \( T \circ S \) captures \( T \circ S \circ T \) 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 \( t \in U^c_T \), \( T_{S \circ T}^p(t) \supseteq T_T^q(T_S^p(t)) \), and if \( S \) is single-valued or if \( T \) is linear then \( T_{S \circ T}^p(t) \subseteq T_T^q(T_S^p(t)) \).

**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 \( L^{p,q} \) is the language accepted at the pair state \( p.q \), we have the following relationship that is used below:

\[ L^{p,q} \overset{\text{def}}{=} \{ t \mid T_T^q(T_S^p(t)) \neq \emptyset \} \]
\[ = \{ t \mid \exists u (u \in T_S^p(t) \land T_T^q(u) \neq \emptyset) \} \]
\[ = \{ t \mid \exists u (u \in T_S^p(t) \land u \in L_T^q) \} \]
\[ = \{ t \mid T_S^p(t) \cap L_T^q \neq \emptyset \}. \]

The symbolic (or procedural) semantics of \( \text{Look}(q, \bar{P}, q, t) \) 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 \( \bar{P} \),

\[ L^P \overset{\text{def}}{=} \bigcap_{p,q \in P} L^{p,q}, \]
\[ L^\bar{P} \overset{\text{def}}{=} \{ \bar{u} \mid \bigwedge_{i=1}^{k} u_i \in L^P \}. \]
The concrete semantics of \( \text{Look}(\varphi, \bar{P}, q, t) \) is defined as follows. We assume that \( t \) implicitly stands for \( \lambda (x, y).t(x, y) \) and \( \varphi \) stands for \( \lambda x.\varphi(x) \).

\[
\llbracket \text{Look}(\varphi, \bar{P}, q, t) \rrbracket \overset{\text{def}}{=} \{(a, \bar{u}) | a \in \llbracket \varphi \rrbracket, \bar{u} \in L^P, \downarrow S(t(a, \bar{u})) \cap L^q_t \neq \emptyset\}\tag{3.3}
\]

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

\[
\llbracket (\varphi, \bar{P}) \rrbracket \overset{\text{def}}{=} \{(a, \bar{u}) | a \in \llbracket \varphi \rrbracket, \bar{u} \in L^P\}.
\]

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

\[
\bigcup \{\llbracket \chi \rrbracket | \text{Look}(\varphi, \bar{P}, q, t) \text{ returns } \chi \} = \llbracket \text{Look}(\varphi, \bar{P}, q, t) \rrbracket \tag{3.4}
\]

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

\[
\bigcup \{\llbracket \chi \rrbracket | \text{Look}(\varphi, \bar{P}, q, \bar{p}(y_i)) \text{ returns } \chi \}
\]

\[=
\llbracket (\varphi, \bar{P} \uplus i.p.q) \rrbracket
\]

\[=
\{(a, \bar{u}) | a \in \llbracket \varphi \rrbracket, \bar{u} \in L^P, u_i \in L^p.q\}
\]

\[=
\{(a, \bar{u}) | a \in \llbracket \varphi \rrbracket, \bar{u} \in L^P, \Delta^P_S(u_i) \cap L^q_t \neq \emptyset\}
\]

\[=
\{(a, \bar{u}) | a \in \llbracket \varphi \rrbracket, \bar{u} \in L^P, \downarrow S(\bar{p}(u_i)) \cap L^q_t \neq \emptyset\}
\]

\[=
\llbracket \text{Look}(\varphi, \bar{P}, q, \bar{p}(u_i)) \rrbracket.
\]

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

\[(a, \bar{u}) \in \bigcup \{\llbracket \chi \rrbracket | \text{Look}(\varphi, \bar{P}, q, f[t_0](\bar{f})) \text{ returns } \chi \}
\]

(Def Look) \[\iff\]

\[\exists \psi, q_1, q_2) (q, f, \psi, (\{q_1\}, \{q_2\}) \in \delta_{d(T)}, \text{IsSat}(\varphi \land \psi(t_0)), \text{Look}(\varphi \land \psi(t_0), \bar{P}, q_1, t_1) \text{ returns } (\varphi', \bar{P}'),
\]

(Def Look) \[\iff\]

\[\text{Look}(\varphi', \bar{P}, q_2, t_2) \text{ returns } (\varphi''', \bar{P}'''), (a, \bar{u}) \in \llbracket (\varphi''', \bar{P}''') \rrbracket
\]

(IH) \[\iff\]

\[\exists \psi, q_1, q_2 (q, f, \psi, (\{q_1\}, \{q_2\}) \in \delta_{d(T)}, \text{IsSat}(\varphi \land \psi(t_0)), \text{Look}(\varphi \land \psi(t_0), P, q_1, t_1) \text{ returns } (\varphi', \bar{P}'),
\]

(Eq (3.3)) \[\iff\]

\[\exists \psi, q_1, q_2 (q, f, \psi, (\{q_1\}, \{q_2\}) \in \delta_{d(T)}, \text{IsSat}(\varphi \land \psi(t_0)),
\]
Next, we prove (3.6) under the assumption that $S$.

The base case is $t$ in $\text{proper subterm of } a$. The proof is by induction over $(\text{Eq (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 $\text{Reduce}$ denotes the function $\delta_{(\varphi, \bar{P}, t)}$ such that, for all $a \in \mathcal{U}^\varphi$ and $u_i \in \mathcal{U}^\mathcal{E}$,

$$\delta_{(\varphi, \bar{P}, t)}(a, \bar{u}) \overset{\text{(H)}}{=} \begin{cases} \downarrow_T(t(a, \bar{u})), & \text{if } (a, \bar{u}) \in [\varphi]; \\ \emptyset, & \text{otherwise}. \end{cases}$$

(3.5)

Next, we prove (3.6) under the assumption that $S$ is single-valued or $T$ is linear. For all $a \in \mathcal{U}^\varphi$, $u_i \in \mathcal{U}^\mathcal{E}$ and $v \in \mathcal{U}^\mathcal{E}$,

$$\exists a, v \in \delta_u(a, \bar{u}) \land \text{Reduce}(\varphi, \bar{P}, t) \text{ returns } a \iff v \in \delta_{(\varphi, \bar{P}, t)}(a, \bar{u}).$$

(3.6)

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

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

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

$$\downarrow_T(\bar{q}(f[a](\bar{e}))) \overset{\text{(H)}}{=} \bigcup \{ \downarrow_T(u(a, \bar{e})) \mid (q, f, \varphi, \bar{e}, u) \in \Delta_T, a \in [\varphi], R(f) \overset{\text{R}}{\bigwedge}_{i=1}^\varphi(\downarrow_T(t_i) \cap L_T^i \neq \emptyset) \}. $$

(3.7)
Observe that when \( t_i \) does not contain any states of \( S \) then \( \downarrow_S(t_i) = \{ t_i \} \) and thus the condition \( \downarrow_S(t_i) \cap L^i_T \neq \emptyset \) simplifies to the condition \( t_i \in L^i_T \) used in the original version of Definition 3.13.

There are two induction cases. The first induction case is \( t = \tilde{q}(f[t_0](\bar{I})) \) where \( q \in \mathcal{Q}_T \) and \( f \in \Sigma \). Let \( t' = f[t_0](\bar{I}) \). For all \( a \in \mathcal{U}^\omega, u_i \in \mathcal{U}^T_{\mathcal{E}} \) and \( v \in \mathcal{U}^T_{\mathcal{E}} \),

\[
(\exists \alpha) \; v \in \partial_\alpha(a, \bar{u}), \text{Reduce}(\varphi, \bar{P}, \tilde{q}(t')) \text{ returns } \alpha
\]

\[
\mathsf{Def} \text{ Reduce}
\]

\[
(\exists \beta) \; \text{Reduce}(\varphi, \bar{P}, u(t_0, \bar{I})) \text{ returns } \beta, v \in \partial_\beta(a, \bar{u})
\]

\[
(\exists \beta') \; \text{Reduce}(\varphi, \bar{P}, u(t_0, \bar{I})) \text{ returns } \beta', v \in \partial_\beta(a, \bar{u})
\]

\[
(\exists \beta'' \tau) \; \text{Reduce}(\varphi, \bar{P}, u(t_0, \bar{I})) \text{ returns } \beta'', v \in \partial_\beta(a, \bar{u})
\]

The second induction case is \( t = f[t_0](\bar{I}) \). Assume \( R(f) = 2 \). Generalization to arbitrary ranks is straightforward by repeating IH steps below \( R(f) \) times. For all \( a \in \mathcal{U}^\omega, u_i \in \mathcal{U}^T_{\mathcal{E}} \), and \( v \in \mathcal{U}^T_{\mathcal{E}} \),

\[
(\exists \alpha) \; v \in \partial_\alpha(a, \bar{u}), \text{Reduce}(\varphi, \bar{P}, f[t_0](t_1, t_2)) \text{ returns } \alpha
\]

\[
\mathsf{Def} \text{ Reduce}
\]

\[
(\exists \beta') \; \text{Reduce}(\varphi, \bar{P}, f[t_0](t_1, t_2)) \text{ returns } \beta', v \in \partial_\beta(a, \bar{u})
\]

\[
\mathsf{Def} \text{ Reduce}
\]

\[
(\exists \beta'') \; \text{Reduce}(\varphi, \bar{P}, f[t_0](t_1, t_2)) \text{ returns } \beta'', v \in \partial_\beta(a, \bar{u})
\]

\[
(\exists \beta''' \tau) \; \text{Reduce}(\varphi, \bar{P}, f[t_0](t_1, t_2)) \text{ returns } \beta''', v \in \partial_\beta(a, \bar{u})
\]
Equation (3.6) follows by the induction principle.

Finally, we prove \( \mathcal{S}_{S,T}^{p,q} = \mathcal{S}_S^p \circ \mathcal{T}_T^q \). Let \( p \in Q_S, q \in Q_T \) and \( f[a](\bar{u}), \bar{w} \in U_T^C \) be fixed.

\[
\begin{align*}
\bar{w} &\in \mathcal{S}_{S,T}^{p,q}(f[a](\bar{u})) \\
\text{Def Compose} & \iff (\exists \varphi, \bar{t}, t, (p, f, \varphi, \bar{t}, t) \in \Delta_S, (\exists a, \bar{u}), \text{Reduce}(\varphi, \bar{t}, \bar{q}(t)) \text{ returns } a, \\
& \quad w \in \partial_a(a, \bar{u}), \bar{u} \in \mathcal{I}_S^L \\
\text{Eq 3.6} & \iff (\exists \varphi, \bar{t}, t, (p, f, \varphi, \bar{t}, t) \in \Delta_S, w \in \partial_{(\varphi, \bar{t}, \bar{q}(t))}(a, \bar{u}), \bar{u} \in \mathcal{I}_S^L \\
\text{Def } \mathcal{S}_S^p & \iff (\exists \varphi, \bar{t}, t, (p, f, \varphi, \bar{t}, t) \in \Delta_S, a \in [\varphi], \bar{u} \in \mathcal{I}_S^L, w \in \mathcal{T}_T^q(\bar{t}(a, \bar{u}))), \bar{u} \in \mathcal{I}_S^L \\
\text{Def } \mathcal{T}_T^q & \iff (\exists \varphi, \bar{t}, t, (p, f, \varphi, \bar{t}, t) \in \Delta_S, a \in [\varphi], \bar{u} \in \mathcal{I}_S^L, w \in \mathcal{T}_T^q(\bar{t}(a, \bar{u}))), \bar{u} \in \mathcal{I}_S^L \\
\text{Def } \mathcal{S}_S^p & \iff w \in \mathcal{S}_S^p(\mathcal{T}_T^q(f[a](\bar{u})))) \\
\text{Def } \mathcal{T}_T^q & \iff w \in \mathcal{S}_S^p(\mathcal{T}_T^q(f[a](\bar{u}))).
\end{align*}
\]

Step (\( \ast \)) uses Lemma 3.23. It holds only when \( S \) is single-valued or \( T \) is linear. Otherwise, only \( \iff \) 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.ia/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, htmlLawed, 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.
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
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 \cite{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:

\[
\text{sum (map square (upto 1 n))}
\]

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 \textsc{Fast}. We analyzed the performance gain obtained by deforestation in \textsc{Fast} on these transformations.

Evaluation. We considered the function \texttt{map\_caesar} from Figure 3.12 that replaces each value \( x \) of a integer list with \((x + 5) \% 26\). We composed the function \texttt{map\_caesar} with itself several times to see how the performance changed when using \textsc{Fast}. Let’s call \texttt{map}^n the composition of \texttt{map\_caesar} with itself \( n \) times. We run the experiments on
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 lookahead 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 $\sigma$ indicates whether the language supports finite (ff) or infinite (\(\infty\)) alphabets.
### Table 3.1: Summary of domain specific languages for manipulating trees.

<table>
<thead>
<tr>
<th>Language</th>
<th>$\sigma$</th>
<th>Analysis</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>$\infty$</td>
<td>composition, typechecking, pre-image, language equivalence, determinization, complement, intersection</td>
<td>Tree-manipulating programs</td>
</tr>
<tr>
<td>Tiburon</td>
<td>$ff$</td>
<td>composition; type-checking; training; weights; language equivalence, determinization, complement, intersection</td>
<td>NLP</td>
</tr>
<tr>
<td>TTT</td>
<td>$ff$</td>
<td>-</td>
<td>NLP</td>
</tr>
<tr>
<td>ASF+SDF</td>
<td>$\infty$</td>
<td>-</td>
<td>Parsing</td>
</tr>
<tr>
<td>XPath</td>
<td>$\infty$</td>
<td>emptiness for a fragment</td>
<td>XML query (only selection)</td>
</tr>
<tr>
<td>XDuce</td>
<td>$\infty$</td>
<td>type-checking for navigational part (finite alphabet)</td>
<td>XML query</td>
</tr>
<tr>
<td>XQuery, XSLT, STX</td>
<td>$\infty$</td>
<td>-</td>
<td>XML transformations</td>
</tr>
</tbody>
</table>

#### 3.6.4 Applications

The connection between tree transducers and deforestation was first studied by Wadler [Wad88], and then further investigated by Künnemann 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 Künnemann 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 \cite{CDG07} Thm 1.7.5. Recently, new techniques based on anti-chains have been proposed to check universality and inclusion for nondeterministic tree automata \cite{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 \cite{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 \[\text{NSV04, MSV00, MGHK09}\]. 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 \[\text{DGN+13, GNI11, MZDZ13}\] to verification of recursive programs \[\text{CA07, DTR12}\]. 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;
Symbolic visibly pushdown automata

Table 4.1: Properties of different automata models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Bool. Closure and Decidable Equiv.</th>
<th>Determinizability</th>
<th>Hierarchical Inputs</th>
<th>Infinite Alphabets</th>
<th>Binary Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>—</td>
</tr>
<tr>
<td>S-FA</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>—</td>
</tr>
<tr>
<td>S-EFA</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>Adjacent Positions</td>
</tr>
<tr>
<td>DA, RA</td>
<td>some variants</td>
<td>x</td>
<td>trees</td>
<td>✓</td>
<td>Only Equality</td>
</tr>
<tr>
<td>VPA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>—</td>
</tr>
<tr>
<td>S-VPA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Calls/Returns</td>
</tr>
</tbody>
</table>

- 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 $\varphi_{ev}$ 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 $\text{even}(x)$ and $\neg\text{even}(x)$. However, this solution causes the input alphabet to be different from the original one ($\{\text{even}(x), \neg\text{even}(x)\}$ 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 $\varphi_{ev}$ has one state looping on an edge labeled with the predicate $\text{even}(x)$ 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 $A_{\text{pos}}$ and $A_{\text{ev}}$, 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 $\varphi_{\text{pr}}$, 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) [BDM11] 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 $\text{even}(x)$.

**Procedure calls.** Let $x$ be of type $\text{bool}$ 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 $\text{int}$ 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 \( \Sigma \) of symbols, the tagged alphabet \( \hat{\Sigma} \) consists of the symbols \( \langle a \rangle \), \( a \rangle \), and \( \langle a \rangle \), for each \( a \in \Sigma \). A nested word over \( \Sigma \) 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 \( \langle a \rangle \), 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 \( P_x(\Psi) \) and \( P_{x,y}(\Psi) \) to denote the set of unary and binary predicates in \( \Psi \) respectively. We assume that every unary predicate in \( P_x(\Psi) \) contains \( x \) as the only free variable (similarly \( P_{x,y}(\Psi) \) with \( x \) and \( y \)).

**Definition 4.1** (S-VPA). A (nondeterministic) symbolic visibly pushdown automaton over an alphabet \( \Sigma \) 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;
- \( \delta_i \subseteq Q \times P_x \times Q \) is a finite set of internal transitions;
- \( \delta_c \subseteq Q \times P_x \times Q \times P \) is a finite set of call transitions;
- \( \delta_r \subseteq Q \times P_{x,y} \times P \times Q \) is a finite set of return transitions;
- \( \delta_b \subseteq Q \times P_x \times Q \) is a finite set of empty-stack return transitions;
Symbolic visibly pushdown automata

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

A transition $(q, \varphi, q') \in \delta$, where $\varphi \in \mathcal{P}_x$, when reading a symbol $a$ such that $a \in [\varphi]$, starting in state $q$, updates the state to $q'$. A transition $(q, \varphi, q', p) \in \delta$, where $\varphi \in \mathcal{P}_x$, and $p \in P$, when reading a symbol $\langle a \rangle$ 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 $\varphi \in \mathcal{P}_x,y$, is triggered when reading an input $b$, 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 $\varphi \in \mathcal{P}_x$, is triggered when reading a tagged input $a$ 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 $\Gamma$ the set of all stacks. Given a nested word $w = a_1 \ldots a_k$ in $\Sigma^*$, a run of $M$ on $w$ starting in state $q$ is a sequence $\rho_q(w) = (q_1, \theta_1), \ldots, (q_{k+1}, \theta_{k+1})$, where $q = q_1$, each $q_i \in Q$, each $\theta_i \in \Gamma$, the initial stack $\theta_1$ is the empty sequence $\epsilon$, 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 $q = q_i$, $q' = q_{i+1}$, $a \in [\varphi]$, and $\theta_{i+1} = \theta_i$;

**Call:** if $a_i = \langle a \rangle$, 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 $q = q_i$, $q' = q_{i+1}$, $\theta_i = \theta_{i+1} = \epsilon$, 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 $t_1 = (q_1, \varphi_1, q'_1), t_2 = (q_2, \varphi_2, q'_2) \in \delta$, if $q_1 = q_2$ and $\text{IsSat}(\varphi_1 \land \varphi_2)$, then $q'_1 = q'_2$;

- for each two transitions $t_1 = (q_1, \varphi_1, q'_1, p_1), t_2 = (q_2, \varphi_2, q'_2, p_2) \in \delta$, if $q_1 = q_2$ and $\text{IsSat}(\varphi_1 \land \varphi_2)$, 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 \( \text{IsSat}(\varphi_1 \land \varphi_2) \), 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 \( \text{IsSat}(\varphi_1 \land \varphi_2) \), 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 \( a \in \llbracket \varphi \rrbracket \);
- a transition \( (q, \varphi, q', p') \in \delta_c \), such that \( a \in \llbracket \varphi \rrbracket \);
- a transition \( (q, \varphi, p, q') \in \delta_r \), such that \( (a, b) \in \llbracket \varphi \rrbracket \);
- a transition \( (q, \varphi, q') \in \delta_b \), such that \( a \in \llbracket \varphi \rrbracket \).

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 $\Phi$ a minterm is a minimal satisfiable Boolean combination of all predicates that occur in $\Phi$. We use the notation $Mt(\Phi)$ to denote the set of minterms of $\Phi$. For example the set of predicates $\Phi = \{x > 2, x < 5\}$ over the theory of linear integer arithmetic has minterms $Mt(\Phi) = \{x > 2 \land x < 5, x > 2 \land \neg x < 5\}$. 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 $\Phi_1^A$ of unary predicates of $A$ as the set $\{\varphi \mid \exists q, q', p. (q, \varphi, q') \in \delta_i \lor (q, \varphi, q', p) \in \delta_i \lor (q, \varphi, q') \in \delta_b\}$;
- the set $\Phi_2^A$ 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 $\Phi$ 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 $\varphi$, 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 $\psi$. 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. \cite{AM09} and the symbolic version is in the use of minterms. Similarly to the approach used to minimize symbolic automata \cite{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. \cite{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 $(S_k, q_k)$ on top of the stack, where $q_k$ 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 $\delta'_i$ 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 $(q',q'',q'') \in \delta_i$ such that $\text{IsSat}(\varphi \land \varphi')$.

The call transition function $\delta'_c$ 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 $(q',q',q'',p) \in \delta_c$ such that $\text{IsSat}(\varphi \land \varphi')$.

The return transition function $\delta'_r$ is given by: for $S,S' \in Q'$ and $\varphi_1 \in Mt_1^A, \varphi_2 \in Mt_1^A$, 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 $(q_2, p, q_2', q'') \in \delta_c$, and a return transition $(q_2', q_2', q'') \in \delta_r$ such that $\text{IsSat}(\varphi_1 \land \varphi_2')$ and $\text{IsSat}(\varphi_2 \land \varphi_2')$.

The empty-stack return transition function $\delta'_e$ is given by: for $S \in Q'$ and $\varphi \in Mt_1^A$, the state $\delta'_e(S, \varphi)$ consists of pairs $(q,q'')$ such that there exists $(q,q') \in S$ and a return transition $(q',q',q'') \in \delta_e$ such that $\text{IsSat}(\varphi \land \varphi')$.

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 $\Sigma$. 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 $\ell$. 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 $O(2^n \ell)$ states, and $O(2^n 2^m \ell)$ stack symbols. The determinization procedure has complexity $O(2^n 2^m m + 2^p f(p \ell))$. 

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 \( q_{\text{sink}} \) and adding a new transition from each state \( q \) to \( q_{\text{sink}} \) covering the inputs for which \( q \) has no transition. Finally \( q_{\text{sink}} \) 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 \( B = (Q \cup \{q_{\text{sink}}\}, q_0, P, \delta'_i, \delta'_c, \delta'_r, \delta'_b, Q_F) \) equivalent to \( A' \). The new state \( q_{\text{sink}} \) is used as a sink for the missing transitions. We now define the new transition relation. We only show \( \delta'_i \), 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 \( \delta'_i = \delta_i \cup \{ (q, \neg \psi, q_{\text{sink}}) \mid \psi = \vee_{\varphi \in I_q} \varphi \} \cup \{ (q_{\text{sink}}, \text{true}, q_{\text{sink}}) \} \). 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 \( \ell \). Let \( f(a) \) be the cost of checking the satisfiability of a predicate of size \( a \). The procedure has complexity \( O(nmf(\ell p)) \). \( \square \)

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 \( A_1 = (Q_1, q'_{01}, P^1, \delta^1_i, \delta^1_c, \delta^1_r, \delta^1_b, Q^1_F) \) and \( A_2 = (Q_2, q'_{02}, P^2, \delta^2_i, \delta^2_c, \delta^2_r, \delta^2_b, Q^2_F) \) (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 \( q'_{0} = (q'_{01}, q'_{02}) \), 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 \( (q_2, \varphi_2, q'_2) \in \delta^2_i \), \( \delta'_i \) will contain the transition \( ((q_1, q_2), \varphi_1 \land \varphi_2, (q'_1, q'_2)) \);
- for each \( (q_1, \varphi_1, q'_1, p_1) \in \delta^1_c \), and \( (q_2, \varphi_2, q'_2, p_2) \in \delta^2_c \), \( \delta'_c \) will contain the transition \( ((q_1, q_2), \varphi_1 \land \varphi_2, (q'_1, q'_2), (p_1, p_2)) \);
for each \((q_1, q_1, p_1, q_1') \in \delta^1_r\), and \((q_2, q_2, p_2, q_2') \in \delta^2_r\), \(\delta_r^r\) will contain the transition

\(((q_1, q_2), q_1 \land q_2, (p_1, p_2), (q_1', q_2'))\);

for each \((q_1, q_1, q_1') \in \delta^1_b\), and \((q_2, q_2, q_2') \in \delta^2_b\), \(\delta_b^b\) will contain the transition

\(((q_1, q_2), q_1 \land q_2, (q_1', q_2'))\).

Assume each S-VPA \(A_i\) has \(n_i\) states, \(m_i\) stack symbols, and \(p_i\) different predicates of size at most \(\ell_i\). Let \(f(a)\) be the cost of checking the satisfiability of a predicate of size \(a\). The intersection procedure has complexity \(O(n_1n_2m_1m_2 + p_1p_2f(\ell_1 + \ell_2))\). \(\Box\)

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 \(A = (Q, q_0, P, \delta, \delta_c, \delta_r, \delta_b, Q_f)\) over a decidable theory \(\Psi\). 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 \(q, q' \in Q\), \(p \in P\), \(q_1 \in P_{x}(\Psi)\), \(q_2 \in P_{y}(\Psi)\), such that \((q, q_1, q_1, p) \in \delta_c\), \((q_2, q_2, p, q') \in \delta_r\), and \(\text{IsSat}(q_1 \land q_2)\), 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 \(q \in Q\), \(\varphi \in P_{x}(\Psi)\), such that \((q, \varphi, q_1) \in \delta_c\), and \(\text{IsSat}(\varphi)\), 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.
Symbolic visibly pushdown automata

- 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 \(\text{IsSat}(\varphi)\), 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 \((q, \varphi, q_1) \in \delta_r\), and \(\text{IsSat}(\varphi)\), 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 \(\ell\). Let \(f(a)\) be the cost of checking the satisfiability of a predicate of size \(a\). The emptiness procedure has complexity \(O(n^3 mt + p^2 f(\ell))\). \(\square\)

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 \(\text{EXPTIME-hard}\) \[AM09\]. If the function \(\text{IsSat}\) 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
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 4.2:** Example XML Schema with an equivalent S-VPA.
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 4.3. 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
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 nondeterministic 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.

```plaintext
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 \( \text{Fib} \), and the values returned by \( \text{Fib} \). 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 \( \text{Fib} \) is greater or equal than 0, then the same holds for all the subsequent inputs of \( \text{Fib} \);
2. if the output of \( \text{Fib} \) is negative, than \( \text{Fib} \) was called exactly once in the whole execution and with a negative input;
3. the output of \( \text{Fib} \) 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 \( \text{Fib} \). 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 \( F = A \cap \overline{B} \cap W \) 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].
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 $\phi_0$;
3. the input contains a subword $\langle a \ b \ c \rangle$ such that $a \in [\phi_1], b \in [\phi_2]$, and $a = c$;
4. the input contains a subword $\langle a \langle b \rangle$ such that $a \in [\phi_3], b \in [\phi_4]$;
5. for every internal symbol $a$ in the input, $a \in [\phi_4]$, or $a \in [\phi_5]$.

The predicates $\Phi = \{\phi_0, \ldots, \phi_5\}$ 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 $D'_1$. Finally we checked equivalence of any two S-VPAs $A$ and $B$ in $D'_2$.

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.\footnote{Available http://www.emn.fr/z-info/hnguyen/vpa/}

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 $\Phi$ 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.
Figure 4.6: Running times for S-VPA equivalence and determination.
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 \textsc{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 \cite{NSV04, MSV00, MGHK09}. In this chapter, we present \textit{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 transformations 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 $\langle a \langle b b \rangle \langle c d d \rangle d d \rangle c a \rangle a$.

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 \textsf{ExpTime} upper bound for type-checking, and provide a \textsf{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 \textsf{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;
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 $\langle a, x \rangle$, $x$ is stored on the stack, and is reset to $\varepsilon$; when reading a return $a$, $x$ is updated to $x_p(\langle a \rangle x)$ where $x_p$ is the variable stored on the top of the stack. In state $q_0$, when reading a call $\langle b, \rangle$, the STT pushes $q_0$ along with the current value of $x$ on the stack, resets $x$ to $\varepsilon$, 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 $x_p(\langle a \rangle x)$ or $x_p(\langle b \rangle x)$, depending on whether the current return symbol is $a$ or $b$. When it pops $q_0$, it updates $x$ to $\langle b \rangle x$ (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 $\varepsilon$ 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 $\epsilon$, 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 $\epsilon$ 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 placeholders 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 $\Sigma$ 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 $\Sigma$ 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 $\Sigma$ is a nested word with only internal positions. Nested words naturally encode ordered trees. The empty tree is encoded by the empty string $\epsilon$. 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 $\langle a a \rangle$, we will use $\langle a \rangle$ as its abbreviation. Thus, a binary
Streaming tree transducers

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 an \( a \)-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 \( \Sigma \) is defined by the grammar

\[
W_0 := \epsilon \mid a \mid \langle a \ W_0 \ b \rangle \mid W_0 \ W_0,
\]

for \( a, b \in \Sigma \). The set \( W_1(\Sigma) \) of type-1 nested words over the alphabet \( \Sigma \) 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 \( \Sigma \) is a subset \( L \) of \( W_0(\Sigma) \), and a nested-word transduction from an input alphabet \( \Sigma \) to an output alphabet \( \Gamma \) 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 \( \Sigma \) and a set \( X \) of typed variables, a valuation \( \alpha \) is a function that maps \( X_0 \) to \( W_0(\Sigma) \) and \( X_1 \) to \( W_1(\Sigma) \). Given an alphabet \( \Sigma \) and a set \( X \) of typed variables, we define the sets \( E_k(\Sigma, X) \), for \( k = 0, 1 \), of type-\( k \) expressions by
the grammars
\[
E_0 := \epsilon | a | x_0 | (a E_0 b) | E_0 E_0 | E_1[E_0], \\
E_1 := ? | x_1 | (a E_1 b) | E_0 E_1 | E_1 E_0 | E_1[E_1]
\]

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 \(\alpha\) 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 \(\eta\): 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 \((x, y) := (((xa)[y], a?)\) 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 \(\eta\), and an alphabet \(\Sigma\), an expression \(e\) in \(E(X, \Sigma)\) is said to be consistent with \(\eta\) 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 \(\eta\), and an alphabet \(\Sigma\), a single-use-restricted assignment is a function \(\rho\) 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 \(\eta\), 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 \(A(X, Y, \eta, \Sigma)\).

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 $\eta$ 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 $A(X, X \cup X_p, \eta, \Sigma)$.

When the conflict relation $\eta$ 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 $\Sigma$ to output alphabet $\Gamma$ 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 $\eta$;
• a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that each expression $F(q)$ is consistent with $\eta$;
• 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 $\rho_i : Q \times \Sigma \mapsto A(X, X, \eta, \Gamma)$;
• a call variable-update function $\rho_c : Q \times \Sigma \mapsto A(X, X, \eta, \Gamma)$;
• a return variable-update function $\rho_r : Q \times P \times \Sigma \mapsto A(X, X \cup X_p, \eta, \Gamma)$.

5.3.7 STT semantics

To define the semantics of a streaming tree transducer, we consider configurations of the form $(q, \Lambda, \alpha)$, where $q \in Q$ is a state, $\alpha$ is a type-consistent valuation from variables $X$ to typed nested words over $\Gamma$, and $\Lambda$ is a sequence of pairs $(p, \beta)$ such that $p \in P$ is a stack symbol and $\beta$ is a type-consistent valuation from variables in $X$ to typed nested words over $\Gamma$. The initial configuration is $(q_0, \varepsilon, \alpha_0)$, where $\alpha_0$ maps each type-0 variable to $\varepsilon$ and each type-1 variable to $?$. The transition function $\delta$ 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 $\Lambda$ 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 $\alpha$.

2. Call transitions: if for some $b \in \Sigma$, $a = \langle b \rangle$, 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 $\eta$, 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 $q$;
• $\Lambda' = (p, a \cdot \rho_c(q, b))\Lambda$ is the new stack resulting from pushing the pair $(p, \alpha')$ on top of the old stack $\Lambda$, 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 $\alpha'$ resulting from applying the variable update function $\rho_c(q, b)$ using the variable values in $\alpha$;
• $\alpha_0$ is the evaluation function that sets every type-0 variable to $\varepsilon$ 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 $\delta((q, (p, \beta)p, \alpha), 0) = (q', \Lambda', \alpha')$ 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 $\beta_p$ is the valuation for variables $X_p$ defined by $\beta_p(x) = \beta(x)$ for $x \in X$, is the result of applying the variable update function $\rho_r(q, p, b)$ using the variable values in $\alpha$, and the stack variable values in $\beta_p$.

We define the closure of $\delta$, $\delta^*$, as $\delta^*((q, \Lambda, \alpha), 0) = (q, \Lambda, \alpha)$ and $\delta^*((q, \Lambda, \alpha), aw) = \delta^*\left(\delta((q, \Lambda, \alpha), a)\right)$. For an input nested word $w \in W_0(\Sigma)$, if $\delta^*((q_0, \varepsilon, \alpha_0), 0) = (q, \varepsilon, \alpha)$ then if $F(q)$ is undefined then so is $\llbracket S \rrbracket(w)$, otherwise $\llbracket S \rrbracket(w) = \alpha(F(q))$. We say that a nested-word transduction $f$ from input alphabet $\Sigma$ to output alphabet $\Gamma$ is $\text{STT-definable}$ if there exists an STT $\mathcal{S}$ such that $\llbracket S \rrbracket = f$.

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 \( a \), and \( b_j = a \) if \( a_j \) is a call symbol \( a \). As a tree transformation, reverse corresponds to recursively reversing the order of children at each node: the reverse of \( \langle a \langle b \langle d \langle e \rangle b \rangle c \rangle \rangle \) is \( \langle a \langle c \langle b \langle e \rangle \rangle \rangle b \rangle a \). This transduction can be implemented by a streaming tree transducer with a single state, a single type-0 variable \( x \), and stack symbols \( \Sigma \): 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 \( f \):

\[
f(\langle c_1x_1x_2(c_2)c_1 \rangle) = \begin{cases} 
  \langle c_1 f(x_2) f(x_1) c_1 \rangle, & \text{if } c_2 = a; \\
  \langle c_1 \text{id}(x_2) \text{id}(x_1) c_1 \rangle, & \text{if } c_2 = b;
\end{cases}
\]

\[
id(\langle c_1x_1x_2(c_2)c_1 \rangle) = \langle c_1 \text{id}(x_1) \text{id}(x_2) c_1 \rangle.
\]
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 \( \eta \) 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 \( \Sigma \) to \( \Gamma \), for all nested words \( w \in W_0(\Sigma) \), \( |f(w)| = \mathcal{O}(|w|) \).

Proof. Given an STT-definable transduction \( f \) from \( \Sigma \) to \( \Gamma \), let \( \mathcal{A}_f = (Q, q_0, P, X, \eta, F, \delta, \rho) \) be an STT defining \( f \). We will show that for every input \( w \), \( ||\mathcal{A}_f||_1(w) = \mathcal{O}(|\mathcal{A}_f|^2|X||w|) \), and since \( |\mathcal{A}_f| \) 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 \( \delta^*(v, (q_0, \varepsilon, a_0), w) = (q, (p_1, \beta_1) \ldots (p_n, \beta_n), a) \), then for every set \( X' \) of non-conflicting variables

\[
\sum_{x \in X'} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) = \mathcal{O}(|\mathcal{A}_f||X||w|).
\]

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

\[
\delta^*((q_0, \varepsilon, a_0), w') = (q, (p_1, \beta_1) \ldots (p_n, \beta_n), a)
\]

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

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

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 \( \mathcal{O}(|\mathcal{A}_f|) \) symbols in \( \Gamma \). Let \( a' = a \cdot \rho_i(q, a) \), then

\[
\sum_{x \in X_1} (|\alpha'(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) \leq \sum_{x \in X_1} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) + \mathcal{O}(|\mathcal{A}_f|) =
\]

\[
\mathcal{O}(|\mathcal{A}_f||X||w'| + |\mathcal{A}_f|) = \mathcal{O}(|\mathcal{A}_f||X||w|).
\]

Notice that we can apply the induction hypothesis since \( X_2 \) is non-conflicting.
\[ a = \langle b \rangle \text{ is a call:} \text{ given a set of non-conflicting variables } X_1, \text{ let } X_2 \text{ be the set of variables appearing in the right-hand sides of } \rho_r(q, b) \text{ for variables in } X_1. \text{ The right hand side also contains } O(|A_f|) \text{ symbols in } \Gamma. \text{ Let } \beta_{n+1} = \alpha \cdot \rho_r(q, b). \text{ Then}
\]
\[
\sum_{x \in X_1} (|\alpha'(x)| + \sum_{1 \leq i \leq n+1} |\beta_i(x)|) \leq \sum_{x \in X_2} (|\alpha_0(x)| + |\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) + O(|A_f|) = \sum_{x \in X_2} |\alpha_0(x)| + \sum_{x \in X_1} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) + O(|A_f|) = O(|A_f| |X||w|).
\]

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

\[ a = b \) \text{ is a return:} \text{ given a set of non-conflicting variables } X_1, \text{ let } X_2 \text{ be the set of variables appearing in the right-hand sides of } \rho_r(q, b, p_n) \text{ for variables in } X_1. \text{ Since both } X_1 \text{ and } X_2 \text{ is non-conflicting, each variable } x \in X_2 \text{ can appear at most once as } x \text{ and once as } x_f \text{ in the right-hand sides of variables in } X_1 \text{ in } \rho_r(q, b, p_n). \text{ The right hand side also contains } O(|A_f|) \text{ symbols in } \Gamma. \text{ Let } \alpha' = \alpha \cdot \beta_n \cdot \rho_r(q, a, p_n). \text{ Then}
\]
\[
\sum_{x \in X_1} (|\alpha'(x)| + \sum_{1 \leq i \leq n-1} |\beta_i(x)|) \leq \sum_{x \in X_2} (|\alpha(x)| + \beta_n(x) + \sum_{1 \leq i \leq n-1} \beta_i(x)) + O(|A_f|) = \sum_{x \in X_2} (|\alpha(x)| + \sum_{1 \leq i \leq n} |\beta_i(x)|) + O(|A_f|) = O(|A_f| |X||w|).
\]

As a consequence, after processing a word \( w \), every variable has a value of size \( O(|A_f||X||w|) \). Since the output function has \( O(|A_f|) \) symbols, the final output has size \( O(|A_f|^2|X||w|) \). This concludes the proof. \qed

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 \cite{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 $\Sigma$ 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 $\Sigma$ to output alphabet $\Gamma$, the domain of $f$ is the set $\text{Dom}(f) \subseteq W_0(\Sigma)$ of input nested words $w$ for which $f(w)$ is defined, and the image of $f$ is the set $\text{Img}(f) \subseteq W_0(\Gamma)$ 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 $\Sigma$ to $\Gamma$, $\text{Dom}(f)$ is a regular language of nested words over $\Sigma$. There exists an STT-definable transduction $f$ from $\Sigma$ to $\Gamma$, such that $\text{Img}(f)$ is not a regular language of nested words over $\Gamma$.

**Proof.** Given an STT-definable transduction $f$ from $\Sigma$ to $\Gamma$, let $A_f = (Q, q_0, P, X, \eta, \delta, \rho)$ be an STT defining it. The NWA $A$ accepting the domain $\text{Dom}(f)$ 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 $\text{Img}(f')$ 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 $a^nb^n$. The image $\text{Img}(f')$ of $f'$ is the language $\{a^nb^n \mid n \geq 0\}$, which is not a regular language of nested words over $\Gamma$. □

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
Streaming tree transducers

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 \( \eta \) is purely reflexive (i.e. \( \{(x, x) | x \in X\} \)) 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 \( \eta \). 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 \( a, b \in \Sigma \), \( x := \epsilon \), \( x := (a?b) \), 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 \( x := yx \), \( y := \epsilon \) 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 \( y := \epsilon \), 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 \( \rho_1 \) belongs to \( \mathcal{A}(X \cup Y, X, \Sigma) \) and it is atomic;
3. for every \( 2 \leq j \leq n \), the assignment \( \rho_j \) 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 \( e \), \( \text{SIZE}(e) \), as the size of its parse tree:

- if \( e \in \{\epsilon, a, (a?b), ?\} \), then \( \text{SIZE}(e) = 1 \);
• if for some \( e_1 \) different from \( ? \), \( e = \langle ae_1b \rangle \), then \( \text{SIZE}(e) = 1 + \text{SIZE}(e_1) \);

• if for some \( e_1, e_2, e = e_1e_2 \), or \( e = e_1[e_2] \), then \( \text{SIZE}(e) = 1 + \text{SIZE}(e_1) + \text{SIZE}(e_2) \).

Given an assignment \( \rho \in \mathcal{A}(X_1, X_2, \Sigma) \), we define \( \text{SIZE}(\rho) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \) to be the value \((a, b, c)\) such that \( a = \max_{x \in X_1} \text{SIZE}(\rho(x)) \) is the maximum size of an 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 \( \{ x \mid \exists y. y \neq x \land \rho(x) = y \} \). 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 \( \rho \) that is not atomic, we show that \( \rho \) can always be transformed into a sequence of atomic assignments \( s = \rho_1\rho_2 \), such that \( \text{SIZE}(\rho_1) < \text{SIZE}(\rho) \), \( \rho_2 \) 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 \( \text{size}(\rho) = (s_1, s_2, s_3) \).

• If \( s_1 = 0 \) the assignment is already atomic.

• If \( s_1 = 1 \) we have two possibilities.

  – If \( \rho \) 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 \( \rho \) 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 \( \rho_1 \) 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 \( e = w_1w_2 \), and \( e \) has size \( s_1 \). Replace \( \rho \) with the sequence \( \rho_1\rho_2 \), such that \( \rho_1, \rho_2 \in \mathcal{A}(X \cup \{v\}, X \cup \{v\}, \Sigma) \): \( \rho_1(x) = w_1, \rho_1(v) = w_2, \rho_2(x) = xv, \rho_2(v) = e \), 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 \( \rho_2 \) is atomic, and since \( w_1 \) and \( w_2 \) both have smaller size than \( e \), \( \text{size}(\rho_1) \) is smaller than \( \text{size}(\rho) \).

  – One variable \( x \) has right-hand side \( e = w_1[w_2] \), and \( e \) has size \( s_1 \). Replace \( \rho \) with the sequence \( \rho_1\rho_2 \) of assignment over \( X \cup \{n\} \) such that: \( \rho_1(x) = w_1, \rho_1(n) = w_2, \rho_2(x) = x[n], \rho_2(x) = e \), 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 \( \rho_2 \) is atomic, and since \( w_1 \) and \( w_2 \) both have smaller size than \( e \), \( \text{size}(\rho_1) \) is smaller than \( \text{size}(\rho) \).

  – One variable \( x \) has right-hand side \( e = \langle awb \rangle \), \( e \) has size \( s_1 \), and \( w \neq ? \). Replace \( \rho \) 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)
\[ \rho_2(x) = x[n], \rho_2(n)e, \text{ and for each } y \in X, \text{ such that } y \neq x, \rho_3(y) = y. \] The assignment \( \rho_2 \) is atomic. Since \( w \) has smaller size than \( e \), \( \text{size}(\rho_1) \) is smaller than \( \text{size}(\rho) \).

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. \( \square \)

**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 \( \rho_1 \) 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 \( \rho_j \) belongs to \( \mathcal{A}(X \cup Y \cup Z, X \cup Y \cup Z, \Sigma) \) and it is atomic;
4. the assignment \( \rho_n \) 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 \( \rho_1 \). This means that there exists two variables \( y_1, y_2 \in Y \) appearing in the right hand side of \( \rho' \), such that both \( \rho(y_1) \) and \( \rho(y_2) \) contain \( x \). If \( y_1 \neq y_2 \), the assignment \( \rho \) cannot be copyless, since it would contain two occurrences of \( x \). If \( y_1 = y_2 \), \( \rho' \) cannot be copyless, since it would contain two occurrences of \( y_1 \). \( \square \)

### 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 $w = a_1a_2\ldots a_n$, for each position $1 \leq i \leq n$, let $WMS(w, i)$ be the longest well-matched subword $a_j\ldots a_i$ ending at position $i$. Formally, given a well-matched nested word $w = a_1a_2\ldots a_n$, we define $WMS(w, i)$ as follows:

- $WMS(w, 0) = \varepsilon$;
- if $a_i$ is internal, then $WMS(w, i) = WMS(w, i - 1)a_i$;
- if $a_i$ is a call, then $WMS(w, i) = \varepsilon$;
- if $a_i$ is a return with matching call $a_j$, then $WMS(w, i) = WMS(w, j - 1)a_ja_{j+1}\ldots a_i$.

The nested word $WMS(w, i)$ 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 $n$, $WMS(w, n)$ equals $w$. Moreover let $LC(w, i)$ denote the last unmatched call at position $i$:

- $LC(w, 0) = \perp$ is undefined;
- if $a_i$ is internal, then $LC(w, i) = LC(w, i - 1)$;
- if $a_i$ is a call, then $LC(w, i) = i$;
- if $a_i$ is a return, and $WMS(w, i) = a_j\ldots a_i$, then $LC(w, i) = LC(w, j - 1)$. 
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 \( WMS(w, i) \) 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 \( \Sigma \).

Next, we define the transition relation \( \delta' \). 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 \((a, \alpha)\), 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 \( b \), 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 \( WMS(w, \text{LC}(w, i - 1)) \) starting in \( q \)), \( \delta_c(q_1, \langle a \rangle) = \langle q_2, p \rangle \), and \( f(q_2) = q_3 \) (the state reached by \( S \) when reading \( WMS(w, i - 1) \) starting in \( q_2 \)), then \( f''(q) = \delta_i(q_3, p, b) \).

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 \( WMS(w, i) \) starting in state \( q \). Given an assignment \( \alpha \in A(X, X, \Sigma) \cup A(X, X \cup X_p, \Sigma) \), a state \( q \in Q \), and a set of variables \( Y \subseteq X \cup X_p \), we define \( \text{SUB}_{q, Y}(\alpha) \) to be the assignment \( \alpha' \in A(X', X', \Sigma) \cup A(X', X' \cup X_p', \Sigma) \), 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 \( \epsilon \), 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 \( x_q := \text{SUB}_{q, X}(\rho_i(q', a, x)) \) 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, \( x_q := x_q; \)
- \( a = b \) 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

\[
x_q := \text{SUB}_{q_2, X}(\text{SUB}_{q_3, X_p}(\rho_r(q_3, b, p, x)\{y_p / \rho_c(q_1, c, y)\{z / z_p\}\}))
\]
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 \( x \in X \) 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 \( \eta' \) 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 \( \eta' \) is consistent with the update function \( \rho' \). 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 \( \eta \), no variables are conflicting in \( \text{SUB}_{q}(\rho(q,a,x)) \) with respect to \( \eta' \). 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 \( \eta' \), 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 \( \text{SUB}_{q_2}(\rho_1(f(q_2),a,w)) \) and \( \text{SUB}_{q_2}(\rho_1(f(q_2),a,z)) \), 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 \( \eta' \).
- 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 \( \eta' \). If \( q_2 \neq q_2' \), \( \eta'(w_{q_2},z_{q_2}') \) follows from the first rule of \( \eta' \).

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 \( w = a_1 a_2 \ldots a_n \), for each position \( 1 \leq i \leq n \), let \( \text{WMP}(w, i) \) be the longest well-matched subword \( a_i \ldots a_j \) starting at position \( i \). Formally given a well-matched nested word \( w = a_1 a_2 \ldots a_n \), \( \text{WMP}(w, n + 1) = \varepsilon \), 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 \( \text{REV}(a) = a \), \( \text{REV}(\langle a \rangle) = \langle a \rangle \), and \( \text{REV}(\langle a \rangle) = \langle a \rangle \). We use \( \text{REV}(w) = \text{REV}(a_n) \ldots \text{REV}(a_1) \), for the reverse of \( w = a_1 \ldots a_n \), 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 \( \text{REV}(a_1 \ldots a_n) \). This motivates the following formalization. Let \( w = a_1 \ldots a_n \) be a nested word over \( \Sigma \), and let \( A \) be a bottom-up NWA with states \( R \) processing nested words over \( \Sigma \). Given a state \( r \in R \), we define the \((r, A)\)-look-ahead labeling of \( w \) to be the nested word \( w_r = r_1 r_2 \ldots r_n \) 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 \(a_i\), and the corresponding symbol is the state of the NWA \(A\) after reading \(\text{REV}(a_j \ldots a_n)\) 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 \(\Sigma\) with states \(R\), and an STT \(S\) from \(R\) to \(\Gamma\). Such a transducer defines a streaming tree transduction from \(\Sigma\) to \(\Gamma\): 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 \(\delta''\). Let \(S_A\) be an STT from \(R\) to \(\Gamma\). 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 \(a_j \ldots a_i\) ending at position \(i\); 2) \(\text{SUB}_{q,Y}(a)\) is the function that substitutes each variable \(x \in Y\) in an assignment \(\alpha\) 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 \(\Lambda\) 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 \(w = a_1 \ldots a_n\), 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 \(\text{WMS}(w, i)\) 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 \( \Sigma \).

We now describe the transition relation \( \delta' \). 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 \( g(r') = q_i \), 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 \( \text{WMS}(w,i-1),r' \);

- \( a_i = \langle a \text{ is a call:} \rangle \) \( \delta'_c(q,a) = (q_0,p) \), 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 \( q_0' \);

- \( a_i = \langle b \) 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 \( r \in R \), \( \delta''_c(r,a) = (p_0, p_r) \) 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 \( r_0 \), and \( g(r_0) = q_1 \) is the state reached by \( S \) after processing \( \text{WMS}(w,i-1),r_0 \). Finally, \( \delta'_c(q,p,a) = (f'',g'') \), where for each \( r \in R \), if \( \delta''_i(r_1,p_r,b) = r_2 \)
  - 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 \( \delta'_i(q_1,p',r_0) = q' \), 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 \( \rho' \). 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 \( x_{r'} \);

- \( a_i = \langle a \text{ is a call:} \rangle S' \) 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\) is a return: let \(a_j = \langle b \rangle\) 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 \(\delta'\). Assume \(\delta''(r, a) = (r_0, p), f(r_0) = r_1, \delta''(r_1, p, b) = r_2, g(r_0) = q_1, g'(r_2) = q_2, \delta_c(q_2, r_2) = (q_3, p')\). For each \(x \in X\), let \(t_c(x)\) be the expression \(\rho_c(q_2, r_2, x)\), and let \(t_r(x)\) 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 \(\rho'(q, a, x_r) = \text{SUB}_r x_p (\text{SUB}_r x(t''(x)))\) where each non stack variable \(x\) is replaced with \(x'_r\), and each stack variable \(y\) is replaced with \(y_r\).

**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 \(F'(f, g) = \text{SUB}_r F(q)\).

**Conflict relation.** Finally, we define the conflict relation \(\eta'\) 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 \(\rho'\) is consistent with \(\eta'\) is analogous to the proof of Theorem 5.7\[5.7\]

**Copyless STTs with RLA.** Recall that an STT is said to be copyless if \(\eta\) 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\[5.8\] to construct an equivalent STT.
We now prove the ⇒ direction. Let $S_1$ be an STT from $\Sigma$ to $\Gamma$. 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 $\Gamma$, such that $\llbracket S', A \rrbracket$ is equivalent to $\llbracket S \rrbracket$.

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 $\text{wmp}(w, i)$ of longest well-matched subword $a_i \ldots a_j$ starting at position $i$. Given a nested word $w = a_1 \ldots a_n$, we also define $\text{nr}(w, i)$ to be the position of the first unmatched return in $a_i \ldots a_n$. By definition, $\text{nr}(w, n + 1) = n + 1$, and for each position $i$ such that $1 \leq i \leq n$:

1. if $a_i$ is internal, $\text{nr}(w, i) = \text{nr}(w, i + 1)$;
2. if $a_i$ is a call, with matching return $a_j$, $\text{nr}(w, i) = \text{nr}(w, j + 1)$;
3. if $a_i$ is a return, $\text{nr}(w, i) = i$.

**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 $\text{rev}(w)$, 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 $\text{nr}(w, i), r$ 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 $\text{nr}(w, i)$.

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 $x \in_s s$ to say that a variable $x \in X$ appears in $s$.

Given a nested word $w = a_1 \ldots a_n$, and an STT $S$, we define the function $\text{cv}_{S,w} : Q \times \{0, \ldots, n\} \times U_X^F \rightarrow U_X^F$, such that, for every state $q \in Q$, position $i$, and set of variables $Y$, $\text{cv}_{S,w}(q, i, Y) = Y'$ 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 $\text{nr}(w, i)$ is $Y$. For every $Y \in U_X^F$, $\text{cv}_{S,w}(q, n, Y) = Y$. For every $0 \leq i \leq n - 1$ we have the following possibilities.
Streaming tree transducers

For every $i$

Lemma 5.10.

Let

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 $\langle a \rangle$, 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 $Y_3 = Y_2$.

$a_{i+1}$ is a call: let $a_{j+1} = b$ be the matching return;

if $\delta_c(q, a) = (q_0, p)$, $\delta_i(q_0, a_{i+2} \ldots a_j)) = q_1$, $\delta_c(q_1, a_{i+1}, p) = q_2$, and $CV_{S, \omega}(q_2, j+1, Y_1) = Y_2$, then $CV_{S, \omega}(q, i, Y_1) = Y_3$ 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 $Y_3 = \{x \mid \exists y \in Y_2 \land x^p \in_a \rho_r(q_1, b, p, y)\}$.

$a_{i+1}$ is a return: $CV_{S, \omega}(q, i, Y) = Y$ if $Y \neq x_F$, and undefined otherwise.

Before continuing we prove that the function $CV_{S, \omega}$ always returns a set of non-conflicting variables.

Lemma 5.10. For every $i \in \{0, \ldots, n-1\}$, $q \in Q$, $Y \in U_X^F$, if $CV_{S, \omega}(q, i, Y) = Y'$ then $Y' \in U_X^F$.

Proof. We proceed by induction on $i$. The base case, $i = n$ and $CV_{S, \omega}(q, n, Y, Y) = Y$, is trivial. We now have to show that for all $i < n$, if $CV_{S, \omega}(q, i, Y) = Y'$, then $Y' \in U_X^F$. We assume by induction hypothesis that, for all $q' \in Q$, $Z \in U_X^F$, if $CV_{S, \omega}(q', i + 1, Z) = Z'$ then $Z' \in U_X^F$. We have the following three cases.

$a_{i+1}$ is internal: we need to prove that $Y_3 \in U_X^F$. By IH, we know that $Y_2 \in U_X^F$. 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 $a_{i+2} = \langle a \rangle$ 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 $q(x, y)$ holds. If this is the case there must exist either two variables $x', y' \in Y_3$ such that $x \in_a \rho_r(q', a_{i+2}, x')$ 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 $Y_2 \in U_X^F$ contains only conflict free variables is violated.

$a_{i+1}$ is a call: similar to the previous case;
where

$s$ a nested word

ing backward, what is the set of variables that will be relevant at the next call. Given

Next, we define the set of states

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 $w = a_1 \ldots a_n$, and a position $i$, we define $NC(w, i)$ as the next call in

$WMP(w, i)$. Formally, $NC(w, n + 1) = \bot$, and, for each $1 \leq i \leq n$,

- if $a_i$ is internal, then $NC(w, i) = NC(w, i + 1)$;
- if $a_i$ is a call, then $NC(w, i) = i$;
- if $a_i$ is a return, then $NC(w, i) = \bot$.

Next, we define the set of states $R$ of $A$. Each state $r \in R$, is a quadruple $(s, f, g, h)$

where $s \in \Sigma \cup \{\bot\}$, $f : Q \times U^F_X \mapsto U^F_X$, $g : Q \mapsto Q$, and $h : Q \times U^F_X \mapsto (U^F_X \cup \bot)$,

where $f$ computes the function $CV_{S,w}$, $g$ summarizes the execution of $S$ on

$WMP(w, i)$, 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 $w = a_1 \ldots a_n$, after processing $REV(a_1 \ldots a_n)$, $A$ is in state $(s, f, g, h)$, where:

1. for all $Y \in U^F_X$, and $q \in Q$, if $CV_{S,w}(q, i, Y) = Y'$, then $f(q, Y) = Y'$;
2. if $a_i$ is a return, then $s = \bot$, otherwise $s = a_i$;
3. if $q' = \delta^*(q, WMP(w, i))$, then $g(q) = q'$;
4. for all $Y \in U^F_X$, and $q \in Q$, $h(q, Y) = Y_1$, where
   
   - if $NC(w, i) = \bot$, then $Y_1 = \bot$,
   - if $NC(w, i) = i_c$, $a_{i_c} = (a$ has matching return $a_{i_c} = b$), $\delta^*(q, a_i \ldots a_{i_c-1}) = q_1$, $\delta(q_1, a_{i_c}) = (q_0, p)$, $\delta^*(q_0, WMP(w, i_c + 1)) = q_2$, $\delta_r(q_2, p, a_{i_c}) = q_3$, and $CV_{S,w}(q_3, i_r, Y) = Y_2$, then

This concludes the proof. □

**RLA automaton construction.** We now construct the NWA $A$ that computes the function $CV_{S,w}$. The NWA $A$ mimics the definition of $CV_{S,w}$ 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 $CV_{S,w}$ 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 $CV_{S,w}$.

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
Streaming tree transducers

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 $Y \in \mathcal{U}_X^F$.

Next we define the transition relation $\delta''$ 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: $\delta''(r, a) = (a, f_1, g_1, h_1)$ where for each $q \in Q$, $Y_1 \in \mathcal{U}_X^F$, 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 $s = \langle c \rangle$ is a call, then $Y_3 = \{x \mid \exists y \in Y_2. \exists z \in_a \rho_c(q_2, c, y). x \in_a \rho_i(q_2, b, z)\}$;
- 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 $\langle b \rangle$ (return reading backward): let $r_1 = ((s_1, f_1, g_1, h_1), s)$ be the state popped from the stack, then, $\delta''(r, r_1, b) = (b, f_2, g_2, h_2)$, where for each $q \in Q$, $Y \in \mathcal{U}_X^F$, if $\delta_c(q, b) = (q_0, p)$, $g(q_0) = q_1$, $\delta_i(q_1, s, p) = q_2$, 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 $Y_3 = \{x \mid \exists y \in_a F(q_2) \land x \in_a \rho_r(q_1, s, p, y)\}$;
- if $Y_2 \neq x_F$ and $s_1 = \langle c \rangle$ 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 $Y_3 = \{x \mid \exists y \in Y_2. x \in_a \rho_c(q_1, s_1, y). x \in_a \rho_r(q_2, s, z)\}$;
- if $Y_2 \neq x_F$ and $s_1 = c$ 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 $b$) (call reading backward): $\delta''(r, b) = (r_0, (r, b))$. 


Next, we define the variable update function symbol $r$. The set of stack states is $Q' = Q \times U_X^r$, 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 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 $\delta'$ 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 $\delta_e(q, a) = (q', p)$, and $h(q, Y) = Y'$, then $\delta'_e((q, Y), r) = (q', (p, Y'))$;
- $a$ 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 $\rho'$. 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: $\rho'_e(q, r, x)$ is the same as $\rho_e(q, a, x)$;
  - $a$ is a return: $\rho'_r(q, p, r, x)$ is the same as $\rho_r(q, p, a, x)$;
- if $x \not\in f(q, Y)$, 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 $\rho'_e(q, r, x) = \varepsilon$, otherwise $\rho'_e(q, r, x) = ?$;
  - $a$ 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 $\mathcal{CV}_{5, w}$ 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 $\Sigma$ using the parameters in $\Pi$, is defined by the grammar

$$H := \epsilon \mid a \mid \pi \mid \langle a \ H \ b \rangle \mid H \ H \quad \text{for } a, b \in \Sigma \text{ and } \pi \in \Pi.$$  

For example, the nested word $\langle a \pi_1 \langle b \rangle \pi_2 \rangle a$ 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 $\pi_1$ and $\pi_2$ 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 $\Sigma$, a set $X$ of variables, and a set $\Pi$ of parameters, the set $E(\Sigma, X, \Pi)$ of expressions is defined by the grammar

$$E := \epsilon \mid a \mid \pi \mid x \mid \langle a \ E \ b \rangle \mid E \ E \mid E[\pi \mapsto E] \quad \text{for } a, b \in \Sigma, x \in X, \text{ and } \pi \in \Pi.$$  

A valuation $\alpha$ 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 $\Pi$ and variables $X$, an occurrence-type $\varphi$ is a function from $X$ to $2^\Pi$. A valuation $\alpha$ from $X$ to $H(\Sigma, \Pi)$ is said to be consistent with the occurrence-type $\varphi$ 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 $\pi$, and if $\pi \notin \varphi(x)$, then $\pi$ does not occur in $\alpha(x)$. An occurrence-type from $X$ to $\Pi$ 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 $\varphi$. An occurrence-type $\varphi_X$...
from variables $X$ to $2^\Pi$ is said to be type-consistent with an occurrence-type $\varphi_Y$ from $Y$ to $2^\Pi$ and an assignment $\rho$ from $Y$ to $X$, if for every variable $x$ in $X$:

- the expression $\rho(x)$ is consistent with the occurrence-type $\varphi_Y$, 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 $\alpha$ from $Y$ to $H(\Sigma, \Pi)$ consistent with $\varphi$, the updated valuation $\alpha \cdot \rho$ from $X$ to $H(\Sigma, \Pi)$ is guaranteed to be consistent with $\varphi'$.

Now we can define the transducer model that uses multiple parameters. A multi-parameter STT $\mathcal{S}$ from input alphabet $\Sigma$ to output alphabet $\Gamma$ consists of:

- a finite set of states $Q$;
- an initial state $q_0$;
- a set of stack symbols $P$;
- state-transition functions $\delta_i$, $\delta_c$, and $\delta_r$, 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 $\eta$;
- 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 $\eta$, 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 $\Gamma$ is consistent with $\eta$ 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 $\Gamma$ is consistent with $\eta$ 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 $\Gamma$ is consistent with $\eta$ 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 $\varepsilon$.

Configurations of a multi-parameter STT are of the form $(q, \Lambda, \alpha)$, where $q \in Q$ is a state, $\alpha$ is a valuation from variables $X$ to $H(\Gamma, \Pi)$ that is consistent with the occurrence-type $\varphi(q)$, and $\Lambda$ is a sequence of pairs $(p, \beta)$ such that $p \in P$ is a stack symbol and $\beta$ 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 $\Pi$ 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 $\Pi$ with $|\Pi| = k$, variables $X$ with $|X| = n$, occurrence-type $\varphi$, output function $F$, state-transition functions $\delta_i$, $\delta_c$, and $\delta_r$, and variable-update functions $\rho_i$, $\rho_c$, and $\rho_r$. 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 \( x_2 x_3 \) and the holes in \( x_2 \) and \( x_3 \), with \( \pi_1 \) and \( \pi_2 \) 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 \( f : X' \mapsto (X' \times X') \cup \Pi \cup \{ \varepsilon \} \cup \perp \) maintains information on the symbolic tree representing each variable. Given a variable \( x \in X' \), \( f(x) = \perp \) 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 \( f(x) = \perp \), then \( f^*(x) = \perp \).

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 \( \varepsilon \), 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 \( a, \pi, x, (a\pi b) \));

Concatenation: \( \{ x := xy; y := \epsilon \} \) (and similar cases such as \( \{ x := yx; y := \epsilon \} \));

Parameter substitution: \( \{ x := x[\pi \mapsto y]; y := \epsilon \} \) (and similar cases such as \( \{ x := y[\pi \mapsto x]; y := \epsilon \} \));

Swap: \( \{ x := y; y := x \} \).

**Update functions.** We now describe at the same time the transition relation \( \delta' \) and the variable update function \( \rho' \) 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 \( \bot \). We show that the state invariant is inductively preserved:

\[
\{ x := w \} : \text{ where } w \text{ is a constant. Similarly to what we showed earlier in the informal description, the content of } x \text{ can be summarized using } 2|\varphi(x)| - 1 \text{ variables.}
\]

\[
\{ x := xy; y := \epsilon \} : \text{ in order for the assignment to be well-defined } \varphi(q', x) \text{ must be the same as } \varphi(q, x) \cup \varphi(q, y), \text{ and } |\varphi(q', x)| = |\varphi(q, x)| + |\varphi(q, y)| \leq k. \text{ Let’s assume wlog that both } \varphi(q, x) \text{ and } \varphi(q, y) \text{ are not empty. By IH } x \text{ and } y \text{ use } 2|\varphi(x)| - 1 + 2|\varphi(y)| - 1 = 2(|\varphi(q, x)| + |\varphi(q, y)|) - 2 \leq 2k - 2 \text{ variables. First we assign each } y_i \text{ in the tree rooted in } g(y) \text{ to some unused } x_{i'} \text{ in } V(x). \text{ Let } a(y_i) = x_{i'} \text{ be such a mapping. From the IH we know that at least one type-1 variable } x_j \in V(x) \text{ is unused. We can use } x_j \text{ to concatenate the variables summarizing } x \text{ and } y. \text{ We can reflect such an update in the tree shape of } x \text{ as follows: for every } z \in V(x),
\]

- 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 \( V(x) \):
  \[
  f(z) = f(y\{y'/a(y')\}), z = y, \text{ and } y' = \epsilon;
  \]
- 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 \( V(y) \), \( g'(y) = y' \), \( f'(y') = \epsilon \), and \( y_0 = \epsilon \).

\[
\{ x := x[\pi \mapsto y]; y := \epsilon \} : \text{ in order for the assignment to be well-defined } \varphi(q', x) \text{ must be the same as } (\varphi(q, x) \setminus \{ \pi \}) \cup \varphi(q, y), \text{ and } |\varphi(q', x)| = (|\varphi(q, x)| - 1) + |\varphi(q, y)| \leq k. \text{ Let’s assume wlog that both } \varphi(q, x) \text{ and } \varphi(q, y) \text{ are not empty. By IH } x \text{ and } y \text{ use } 2|\varphi(x)| - 1 + 2|\varphi(y)| - 1 = 2(|\varphi(q, x)| + |\varphi(q, y)|) - 2 \leq 2(k + 1) - 2 = 2k \text{ variables. First we assign each } y_i \neq g(y) \text{ in the tree rooted in } g(y) \text{ to some unused } x_{i'} \text{ in } V(x). \text{ Let } a(y_i) = x_{i'} \text{ be such a mapping. So far we used } 2k - 1 \text{ variables}
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 \( V(x) \):
  \[
  f(z) = f(y \{ y' / a(y') \}) \land y' = \varepsilon;
  \]
- if \( z = x' \pi \) we append the summary of \( y \) to it: if \( g(y) = y' \), then
  - if \( f(y') = y_1 y_2 \), \( a(y') = x' \), \( a(y_1) = x_1 \) and \( a(y_2) = x_2 \), then
    \[f'(z) = (x_1, x_2), \text{ and } \rho(z) = z[y']\];
  - if \( f(y') = \pi' \), and \( a(y') = x' \), then
    \[f'(z) = \pi', \text{ and } \rho(z) = z[y']\].

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 vari-
able 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 \( \delta' \) and \( \rho' \) are defined at calls and returns. The functions main-
tained 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 \(\Sigma_1\) to \(\Sigma_2\) and \(f_2\) from \(\Sigma_2\) to \(\Sigma_3\), the composite transduction \(f_2 \cdot f_1\) from \(\Sigma_1\) to \(\Sigma_3\) 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 $g^l_1(q_1, x, y)$ representing the value of $y$, when $S_2$ reads the value of $x$ on the left of $?$, starting in state $q_1$;
- a variable $g^r_1(q_1, q_2, x, y)$ 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 $g^R_1(q', q, x, y)$ 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 $\eta$ 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 $p_0(q, x, y) = \epsilon$ 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 $l^R_1$ and $l^R_0$, such that $l^R_1(r_2, x) = r'_2$ and $l^R_0(r_1, r_2, x) = r'_1$, 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 $(q, f_0, f^L_1, f^R_1, l_0, l^L_1, l^R_1, p_0, p^L_1, p^R_1)$, where

- $q \in Q_1$ keeps track of the current state of $S_1$;
- $f_0 : Q_2 \times R \times X_{1,0} \to Q_2$ is the summarization function for type-0 variable;
- $f^L_1 : Q_2 \times R \times R \times X_{1,1} \to Q_2$, and $f^R_1 : Q_2 \times Q_2 \times R \times X_{1,1} \to Q_2$ are the summarization functions for type-1 variables;
- $l_0 : R \times X_{1,0} \to R$ is the look-ahead summarization function for type-0 variable;
- $l^L_1 : R \times R \times X_{1,1} \to R$, and $l^R_1 : R \times X_{1,1} \to R$ are the look-ahead summarization functions for type-1 variables;
- $p_0 : Q_2 \times R \times X_{1,0} \to X_{2,1}$ is the function keeping track of the $?$ for type-0 variables;
- $p^L_1 : Q_2 \times R \times X_{1,1} \to X_{2,1}$, and $p^R_1 : Q_2 \times Q_2 \times R \times X_{1,1} \times X_{2,1} \to X_{2,1}$ 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 $w = a_1 \ldots a_n$, after reading the symbol $a_i$, $S$ is in state $(q, f_0, f^L_1, f^R_1, l_0, l^L_1, l^R_1, p_0, p^L_1, p^R_1)$.

- The component $q$ is the state reached by $S_1$ on the prefix $a_1 \ldots a_i$, $\delta^*_i(q_01, a_1 \ldots a_i) = q$.
- Given $q_1 \in Q_2$, and $r \in R$, if $x$ contains the value $s \in W_0(\Sigma_2)$, and $\delta^*_i(q_1, s_r) = q'_1$, 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 $s?t \in W_1(\Sigma_2)$, $\delta^*_2(q_1, s_{r_1, \Lambda_{r_2}}) = (q'_1, \Lambda)$, and $\delta^*_2(q_2, \Lambda, r_{t_2}) = q'_2$, then $f^L_1(q_1, r_1, r_2, x) = q'_1$, and $f^R_1(q_1, q_2, r_1, r_2, x) = q'_2$. We use the notation $s_{r_1, \Lambda_{r_2}}$ 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 \( \delta^*_x(r_1, \text{REV}(s)) = r'_1 \), then \( l_0(r_1, x) = r'_1 \).

• Given \( r_1, r_2 \in R \), if \( x \) contains the value \( s \cap t \in W_1(\Sigma_2) \), \( \delta^*_2(r_2, \text{REV}(t)) = (r'_2, \Lambda) \), and \( \delta^*_2(r_1, \Lambda, \text{REV}(s)) = r'_1 \), then \( l^1_1(r_1, r_2, x) = r'_1 \) and \( l^1_2(r_2, x) = r'_2 \).

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 \( p_0, \Lambda_1 \), and \( p_1^R \). Let’s assume \( S \) is in state \((q, f_0, f_1, f_2^R, l_0, l_1, l_2, p_0, p_1^L, p_1^R)\). 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 \( \delta_R(r_2, \langle b \rangle) = (r'_2, p) \) for some \( p \in P_r \), \( \delta_R(r_1, p, a) = r'_1 \), \( \delta^*_2(q_1, r'_1) = (q'_1, \Lambda) \), and \( \delta^*_2(q_2, \Lambda, r'_2) = q'_2 \), then the new state has the following components: \( f^L_1(q_1, r_1, r_2, x) = q'_1 \), \( f^L_2(q_1, q_2, r_1, r_2, x) = q'_2 \), \( l^1_1(r_1, r_2, x) = r'_1 \), and \( l^1_2(r_2, x) = r'_2 \).

\{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 \( l_0(r_l, y) = r_1 \), \( l^R_1(r_l, x) = r_2 \), \( l^L_1(r_l, r_r, x) = r_3 \), \( f^L_1(q_l, r_l, r_r, x) = q_1 \), \( f^R_1(q_l, r_l, r_r, x) = q_2 \), and \( f_0(q_r, r_r, y) = q_3 \), then for every \( q \in Q_2 \), and \( r \in R \), the new state has the following components: \( l^L_1(q_l, r_l, r_r, x) = q_3 \), \( l^R_1(r_r, x) = r_2 \), \( l_0'(r_l, y) = r \), \( f^L_1(q_l, r_l, r_r, x) = q_1 \), \( f^R_1(q_l, q_r, r_l, r_r, x) = q_3 \), and \( f_0'(q_r, r_r, y) = q \).

\{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 \( l^R_1(r_l, x) = r_1 \), \( l^L_1(r_l, y) = r_2 \), \( l^L_1(r_3, r_r, r_r, x) = r_4 \), \( f^L_1(q_l, r_l, r_r, r_r, x) = q_1 \), \( f^L_1(q_l, r_l, r_l, y) = q_2 \), \( f^R_1(q_l, q_r, r_l, r_l, y) = q_3 \), and \( f^R_1(q_l, q_r, r_r, r_r, x) = q_4 \). For every \( q, q' \in Q_2 \), and \( r, r' \in R \), the new state has the following components: \( l^L_1(q_l, r_l, r_r, x) = r_4 \), \( l^R_1(r_r, x) = r_2 \), \( l^L_1(r_r, r', y) = r \), \( l^R_1(r, y) = r_1 \), \( f^L_1(q_l, r_l, r_r, x) = q_2 \), \( f^R_1(q_l, q_r, r_l, r_r, x) = q_4 \), \( f^L_1(q_l, r_r, r', y) = q_4 \), and \( f^R_1(q_l, q_r, r', y) = q' \).

\{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 \( w \), 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:

- \( \{g_0(q, r, x, y) \mid (q, r, x, y) \in Q_2 \times R \times X_{1,0} \times X_2\} \);
- \( \{g_0(q, r, x, y) \mid (q, r, x, y) \in Q_2 \times R \times X_{1,0} \times X_{2,1}\} \);
- \( \{g_1^f(q, r_1, r_2, x, y) \mid (q, r_1, r_2, x, y) \in Q_2 \times R \times X_{1,1} \times X_2\} \);
- \( \{g_1^f(q, r_1, r_2, x, y) \mid (q, r_1, r_2, x, y) \in Q_2 \times R \times X_{1,1} \times X_{2,1}\} \);
- \( \{g_1^R(q, q', r_1, r_2, x, y) \mid (q, q', r_1, r_2, x, y) \in Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_2\} \);
- \( \{g_1^R(q, q', r_1, r_2, x, y) \mid (q, q', r_1, r_2, x, y) \in Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_{2,1}\} \).

Given a nested word \( w \), and a stack \( \Lambda \), we use \( w_{r,\Lambda} \) to denote the regular look-ahead labeling of \( w \) when processing \( \text{REV}(w) \) in the starting configuration \((r, \Lambda)\). 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 \( \delta^v(q_1, v_{r_1, \Lambda_{r_2}}) = (q_2, \Lambda_{r_2}) \), then:

- \( g_1^f(q_1, r_1, r_2, x, y) \) 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 \((q_2, \Lambda_{r_2})\).

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 STS \( S \) has set of parameters \( \Pi = \{x' \mid x \in X_2\} \cup \{?\} \).

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 \( \alpha \) 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 \( w \), with initial configuration \( \alpha \) can be retrieved from the variables in \( X \) and the state components \( p_0, p_1^L, p_1^R \).

Given a nested word \( w = a_1 \ldots a_n \), we consider the configuration of \( S \) and \( S_1 \) right after processing the symbol \( a_i \). Let’s call \((q_1, \Lambda_1, a_1)\) the current configuration of \( S_1 \), and \((q_S, \Lambda, a)\), 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 $\{v'_1, \ldots, v'_k\} \subseteq \Pi$ be the set of parameters in $\varphi(g_0(q_2, r, x_0, y_0))$; for every valuation $\alpha_2$ over $X_2$, if $\delta^*((q_2, \alpha_2), s^{s_0}) = (q_3, \alpha'_2)$, then:

$$a'_2(y_0) = t[v'_1 \mapsto \alpha_2(v_1)] \ldots [v'_k \mapsto \alpha_2(v_k)].$$

$x_0, y_1$: 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 $S$, $p_0(q_2, r, x_0, y_1) = z_1 \ldots z_i$ be the sequence of parameters to follow to reach the hole $\omega$ in the representation of $y_1$, and $\{v'_1, \ldots, v'_k\} \subseteq \Pi$ be the set of parameters in

$$(\varphi(g_0(q_2, r, x_0, y_1)) \cup \varphi(g_0(q_2, r, x_0, z_1)) \cup \ldots \cup \varphi(g_0(q_2, r, x_0, z_i))) \setminus \{z_1, \ldots, z_i\}$$

then, for every valuation $\alpha_2$ over $X_2$, if $\delta^*((q_2, \alpha_2), s^{s_0}) = (q_3, \alpha'_2)$, then:

$$a'_2(y_1) = t[z'_1 \mapsto \varphi_0(q_2, x_0, z_1)[\ldots [z'_i \mapsto \varphi_0(q_2, x_0, z_i)[\ldots]]] \ldots [v'_k \mapsto \alpha_2(v_k)].$$

$x_1, y_0$: let $\alpha_1(x_1) = s^{s_1} ? s^{s_0}$ 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 $\{v'_1, \ldots, v'_k\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^l(q_2, r, r', x_1, y_0))$, and let $\{v''_1, \ldots, v''_k\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^r(q_2, q'_2, r, r', x_1, y_0))$; for every valuations $\alpha_2, \alpha'_2$ over $X_2$, if $\delta^*((q_2, \alpha_2), s^{s_1}) = (q_3, \Lambda_2, \alpha_3)$, and $\delta^*((q_2, \alpha'_2), s^{s_1}) = (q_3, \alpha'_3)$, then $a_3(y_0) = t_L[v'_1 \mapsto \alpha_2(v'_1)] \ldots [v'_k \mapsto \alpha_2(v'_k)]$, and $a'_3(y_0) = t_R[v''_1 \mapsto \alpha_2(v''_1)] \ldots [v''_k \mapsto \alpha_2(v''_k)].$

$x_1, y_1$: let $\alpha_1(x_1) = s^{s_1} ? s^{s_0}$ 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 $S$, $p_1^l(q_2, r, r', x_1, y_1) = z_1^l \ldots z_i^l$ be the sequence of variables to follow to reach the hole $\omega$ in the representation of $y_1$ on the right of the $\omega$, $p_1^r(q_2, q'_2, r, r', x_1, y_1) = z_1^R \ldots z_i^R$ be the sequence of variables to follow to reach the hole $\omega$ in the representation of $y_1$ on the right of the $\omega$, $\{v'_1, \ldots, v'_k\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^l(q_2, r, r', x_1, y_1))$, and $\{v''_1, \ldots, v''_k\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^r(q_2, q'_2, r, r', x_1, y_1))$; for every valuations $\alpha_2, \alpha'_2$ over $X_2$, if $\delta^*((q_2, \alpha_2), s^{s_1}) = (q_3, \Lambda_2, \alpha_3)$, and
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 $\varphi: Q \times X \rightarrow 2^I$ to be well defined according to the following assignments (we will prove consistency later).

\[
\{x_0 := w\}: \text{ where without loss of generality } w \text{ is a constant without a } ?. \text{ We fix the variable components to be the state } q \in Q_2, \text{ look-ahead state } r \in R, \text{ and we are summarizing the values of } y_0 \in X_{2,0} \text{ and } y_1, y_2 \in X_{2,1}, \text{ when reading the value in } x_0 \in X_{1,0}. \text{ We consider the variable updates performed by } S_2 \text{ when reading the } w, \text{ and assume } u = u_1 \ldots u_k \text{ to be the sequence of atomic updates (using Theorem[5,4]) performed by } S_2 \text{ when reading } w, \text{ starting in state } q. \text{ We now provide the updates of } S \text{ corresponding to the updates in } u. \text{ At the beginning of the procedure } g_0(q, r, x_0, y_1) = y'_1, \text{ and } p_0(q, r, x_0, y_1) = y_1. \]

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:

\[
\{y_1 := w\}: \text{ where } w = \langle a?b \rangle \text{ (the other cases are similar). We have } p'_0(q, r, x_0, y_1) = \varepsilon, \text{ and we define the current assignment to be } g_0(q, r, x_0, y_1) := w.
\]

\[
\{y_0 := \varepsilon; y_1 := y_0 y_1\}: p'_0(q, r, x_0, y_1) = p_0(q, r, x_0, y_1), \text{ and we define the current assignment to be } g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_0)g_0(q, r, x_0, y_1), \text{ and } g_0(q, r, x_0, y_0) = \varepsilon.
\]

\[
\{y_1 := y_1[y_2]; y_2 := ?\}: \text{ the summary } p_0 \text{ is updated as } p'_0(q, r, x_0, y_1) = p_0(q, r, x_0, y_1)p_0(q, r, x_0, y_1)p_0(q, r, x_0, y_2), p_0(q, r, x_0, y_2) = \varepsilon; \text{ if } p_0(q, r, x_0, y_1) = v_1 \ldots v_k, \text{ then we define the current assignment to be}
\]

- if $k = 0$, then $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_1)[g_0(q, r, x_0, y_2)], \text{ and } g_0(q, r, x_0, y_2) = ?;$

- if $k > 0$, then $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_1), g_0(q, r, x_0, v_k ?) := g_0(q, r, x_0, v_k ?)[g_0(q, r, x_0, y_2)], \text{ and } g_0(q, r, x_0, y_2) = ?.$
\{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 \(ax'\), 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 \(l_0(r, x_1) = r_1\), \(f_0(q_2, r_1, x_0) = q_3\), \(\varphi(q_3, g_0(q_3, r, x_1, y_1)) = \{v'_1, \ldots, v'_k\}\), 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

\[g_0'(q_2, r, x_0, v) := g_0(q_3, r, x_1, v)\]
\[p'_i \mapsto g_0(q_3, r_1, x_0, p_i)\]
\[\ldots p'_{i-1} \mapsto g_0(q_3, r_1, x_0, p_{i-1})\]
\[\vdots \mapsto \vdots\];

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 \( \eta \) over \( X \) consistent with \( \rho \). 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 \( \eta \) 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 \( \eta(g_1(q_1, r_1, r_2, y, u), g_1(q'_1, r'_1, r'_2, y, v)) \);
- if \( (q_1, r_1, r_2) \neq (q'_1, r'_1, r'_2) \), then \( \eta(g_1(q_1, r_1, r_2, y, u), g_1(q'_1, r'_1, r'_2, y, v)) \);
- if \( (q_1, q_2, r_1, r_2) \neq (q'_1, q'_2, r'_1, r'_2) \), then \( \eta(g^R(q_1, q_2, r_1, r_2, y, u), g^R(q'_1, q'_2, r'_1, r'_2, y, v)) \);
- if \( (q_1, q_2, r_1, r_2) \neq (q'_1, q'_2, r'_1, r'_2) \), then \( \eta(g^R(q_1, q_2, r_1, r_2, y, u), g^R(q'_1, q'_2, r'_1, r'_2, y, v)) \).

We now show that the variable update function \( \rho \) does not violate the conflict relation \( \eta \). 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. \( x \neq y \), \( \eta(x, y) \), and both \( x \) and \( y \) occur on the right-hand side of some variable;
2. \( \eta(x, y) \), and there exists two variables \( x' \) and \( y \), such that \( x' := \text{FUN}(x) \), \( y' = \text{FUN}(y) \) for which \( \eta(x', y') \) doesn’t hold.

Case (1) can be ruled out by simply inspecting all the possible assignments in the definition of \( \rho \). The only interesting cases are \( \{ x_0 := x_0, x_1 := \varepsilon \} \) 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 \( x, y \) assigned to different variables can be in conflict. The first case is \( x = y \). The case for \( \{ x_0 := w \} \) can be ruled out by inspecting the assignments. For the cases \( \{ x_0 := x_0x_1; x_1 := \varepsilon \} \) 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 \( \eta \) 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 \( \rho \). Let’s assume we are in state \( q_S = (q, f_0, f_L^0, f_R^0, l_0, l_L^0, l_R^0, p_0, p_L^0, p_R^0) \), 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 $\Sigma$. 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 $\Gamma$ 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) $\mathcal{S}$ from input alphabet $\Sigma$ to output alphabet $\Gamma$ 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 $\eta$; a partial output function $F : Q \to E_0(X, \Gamma)$ such that for each state $q$, a variable $x$ appears at most once in $F(q)$; a state-transition function $\delta : Q \times \Sigma \to Q$; and a variable-update function $\rho : Q \times \Sigma \to A(X, X, \eta, \Gamma)$. Configurations of such a transducer are of the form $(q, \alpha)$, where $q \in Q$ is a state, and $\alpha$ is a type-consistent valuation for the variables $X$. The semantics $[\mathcal{S}]$ of such a transducer is a partial function from $\Sigma^*$ 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 $\eta$ 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 $\Sigma$, 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 $\Sigma$. 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 $f \in \mathcal{SF}(f')$, then $f \in \mathcal{SF}(f')$;
• the empty forest $\epsilon$ belongs to $\mathcal{SM}(f')$;
• let $f \equiv s_0(t_0) \ldots s_n(t_n)$, and $f' \equiv s'_0(t'_{0}) \ldots s'_{m}(t'_{m})$. If there exist $0 \leq i \leq m - n$ such that $s_0 \ldots s_n = s'_i \ldots s'_{i+n}$, and for every $0 \leq j \leq n, t_j \in SFM(t'_{i+j})$, then $f \in SF(f')$.

Intuitively, the first rule in the definition of $SF$ allows to move down in the forest until an exact match in $SF\mathcal{M}$ is found by the other rules. Given two forests $f_1, f_2 \in \mathcal{F}(U)$, we write $\mathcal{S}(f_1, f_2) \equiv SF(f_1) \cap SF(f_2)$ for the set shared sub-forests of $f_1$ and $f_2$. Finally the set of maximal shared sub-forests is defined as

$$\mathcal{M}(t_1, t_2) = \{ f | f \in \mathcal{S}(t_1, t_2) \land \neg \exists f' \in \mathcal{S}(t_1, t_2), f' \neq f \land f \in SF(f') \}.$$  

**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 $\delta'$ 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 $\delta'$. 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 \( r \) 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 \( \Pi \) 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 \( u(f) \equiv u_i(s_0,f_0) \ldots u_i(s_n,f_n) \), where for every \( s \in X \), and \( g_0 \ldots g_m \in X^* \), \( u_i(s,g_0 \ldots g_m) \equiv s[\pi_0 \mapsto u(g_0), \ldots, \pi_m \mapsto u(g_m)] \).

After reading \( i \)-th symbol \( a_i \) of an input word \( w \), \( S' \) is in a configuration \(( (l,f,g),a) \) iff for every look-ahead state \( r \in R \), and variable \( x \in X \), if \( \delta^*(q_0,REV((a_1 \ldots a_i)_r)) = (q,a_1) \), and \( u(g(r,x)) = s \), then \( a_1(x) = a(s) \).

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 $r, r' \in R$, \( \sum_{x, y \in X} |M(g(r, x), g(r', y))| \leq |X| \).

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 $f f' \in SF(g(r, x))$, such that $? \not\in ff'$, then there must be a shape $g(r', x')$, with $(r', x') \neq (r, x)$, such that either $f \in SF(g(r', x'))$ and $ff' \not\in SF(g(r', x'))$, or $f' \in SF(g(r', x'))$ and $ff' \not\in SF(g(r', x'))$.

Vertical compression: for every $s(f) \in SF(g(r, x))$, such that $? \not\in s(f)$, then there must be a shape $g(r', x')$, with $(r', x') \neq (r, x)$, such that either $s() \in SF(g(r', x'))$ and $s(f) \not\in SF(g(r', x'))$, or $f \in SF(g(r', x'))$ and $s(f) \not\in SF(g(r', x'))$.

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 $X'_0 = \{z_1, \ldots, z_k\}$, $X'_1 = \{z'_1, \ldots, z'_k\}$, $X_0 = \{x_1, \ldots, x_i\}$, $X_1 = \{x'_1, \ldots, x'_i\}$. For each type-0 variable $x_i \in X_0$, for each type-1 variable $x'_j \in X_1$, and look-ahead state $r \in R$, we have that $g_0(r, x_i) = z_i$, $z_i = \epsilon$, $g_0(r, x'_j) = z'_j(?)$, 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 \(z_f\).

\[
\{ x := xy, y := \epsilon \}: \text{ we perform the following update: } g'(r, x) = g(r', x)g(r', y). \text{ Let } g(r', x) = s_1(f_1) \ldots s_n(f_n) \text{ and } g(r', y) = s'_1(f'_1) \ldots s'_m(f'_m). \text{ We now have two possibilities:}
\]

- 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 \(f_n = t_1 \ldots t_i\) and \(f'_1 = t'_1 \ldots t'_k\), then

\[
s_n := s_n s'_1[\pi_0 \mapsto \pi_{i+1}, \ldots, \pi_k \mapsto \pi_{k+i}] \text{ and } g'(r, x) = s_1(f_1) \ldots s_n(f_n) s'_1(f'_1) \ldots s'_m(f'_m). \text{ In this new assignment to } s_n, \text{ the parameters in } s'_1 \text{ have been shifted by } i \text{ to reflect the concatenation with } s_n.
\]

In both cases, due to the preserved counting invariant, we can take an unused variable \(z_f\) and use it to update \(g(r, y)\): \(z_f := \epsilon\), and \(g'(r, y) = z_f\).

\[
\{ x := x[y], y := ? \}: \text{ without loss of generality let } x \text{ and } y \text{ be type-1 variables. Let } s(?) \text{ be the subtree of } g(r', x) \text{ containing } ? \text{. We perform the following update: } g'(r, x) = g(r', x)\{?/g(r', y)\}, \text{ where } a\{b/c} \text{ replaces the node } b \text{ of } a \text{ with } c. \text{ Let } g(r', y) = s'_1(f'_1) \ldots s'_m(f'_m) \text{. For every } s_i, \text{ we now have two possibilities:}
\]

- 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 \(f'_p = t_1 \ldots t_k\), then \(s := s[\pi_i \mapsto s'_p[\pi_0 \mapsto \pi_{i+1}, \ldots, \pi_k \mapsto \pi_{i+k}], \pi_{i+1} \mapsto \pi_{i+1+k}, \ldots, \pi_{n+k+1}]\), 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.

\[
\{ x := y, y := x \}: \text{ we symbolically reflect the swap. } g'(r, x) = g(r', y), \text{ and } g'(r, y) = g(r', x). \text{ Similarly to before, we compress if necessary.}
\]

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
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 $\pi$ 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.

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 $0 \notin \Sigma$, and every symbol in $\Sigma$ is binary. The set $BT(\Sigma)$ of binary trees over the alphabet $\Sigma$ is then a subset of nested words defined by the grammar $T := 0 | \langle a T T a \rangle$, 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 $a(t_l, t_r)$ based on the symbol $a$, and the results of processing the subtrees $t_l$ and $t_r$.

A bottom-up ranked-tree transducer (BRRT) § from binary trees over $\Sigma$ to nested words over $\Gamma$ 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 $\eta$; a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that for each state $q$, the expression $F(q)$ is consistent with $\eta$; a state-combinator function $\delta : Q \times Q \times \Sigma \mapsto Q$; and a variable-combinator function $\rho : Q \times Q \times \Sigma \mapsto A(X, X_l \cup X_r, \eta, \Gamma)$, where $X_l$ denotes the set of variables $\{x_l | x \in X\}$, $X_r$ denotes the set of variables $\{x_r | x \in X\}$, and the conflict relation $\eta$ 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 $BT(\Sigma)$: $\delta^*(0) = q_0$ and $\delta^*(a(t_l, t_r)) = \delta(\delta^*(t_l), \delta^*(t_r), a)$. The variable-combinator is used to map trees to valuations for $X$: $\alpha^*(0) = a_0$, where $a_0$ maps each type-0 variable to $\epsilon$ and each type-1 variable to $?$, and $\alpha^*(a(t_l, t_r)) = \rho(\delta^*(t_l), \delta^*(t_r), a)|X_l \mapsto \alpha^*(t_l)||X_r \mapsto \alpha^*(t_r)|$. 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 $\alpha_l$ and setting variables in $X_r$ to values given by $\alpha_r$. 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 $a$. 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 $\alpha$.

**Theorem 5.17** (Expressiveness of ranked tree transducers). A partial function from $BT(\Sigma)$ to $W_0(\Gamma)$ is STT-definable iff it is BRRT-definable.

**Proof.** From STT to BRRT $\Rightarrow$. Using Theorem 5.7 let $S = (Q, P, q_0, X, F, \eta, \delta, \rho)$ be a binary bottom-up STT. We construct a BRRT $S' = (Q', q'_0, X', F', \eta', \delta', \rho')$ equivalent to $S$. We can assume w.l.o.g. that the set of variables $X$ is partitioned into two disjoint sets $X_l = \{x_{l1}, \ldots, x_{ln}\}$ and $X_r = \{x_{r1}, \ldots, x_{rn}\}$ such that, given a tree $t = a(t_l, t_r)$, 1) after processing the left child $t_l$, all the variables in $X_l$ depend on the run over $t_r$, 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 $x \in X_t$ is exactly assigned the corresponding value $x^0$ 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 $f_1$ 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 $\delta_2(q, \langle a \rangle) = (q_1, p)$, $f_1(q_1) = q_2$, $f_r(q_2) = q_3$, and $\delta_3(q_3, p, \langle a, x \rangle) = q_4$, then $f'(q) = q_4$, and $x_q$ is assigned the value $\rho(q_3, p, \langle a, x \rangle)$, 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 $\eta'$ 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 $\iff$.** 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 $\langle a \langle t_l, t_r, a \rangle \rangle$, 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 $\delta'$ 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 $\langle b \rangle$: 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 $b$): 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 $\langle a \ t_1 \ t_r \ a \rangle$, 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 $\rho'$ of $S'$. Given a variable $x \in X'$, let the function $\text{INI}(x)$ 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 $\text{INI}(x) = \varepsilon$, and 2) if $x$ is a type-1 variable, then $\text{INI}(x) = \varnothing$. 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 $\langle b \rangle$: every variable is copied on the stack $\rho'(q,b,x) = x$;

$a$ is a return $b$): 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 $\rho'(q, p, b, x) = \text{INI}(x)$.

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 $\eta'$ 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 $\eta$ 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 output 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 $\Gamma$. The set of expressions used in the right-hand sides can be simplified to $E_0 := \epsilon | a | x_0 | E_0 E_0$. 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 $\Gamma$, then it does not need to use holes and substitution.

**Theorem 5.18 (STST expressiveness).** A partial function from $W_0(\Sigma)$ to $\Gamma^*$ is STT-definable iff it is STST-definable.
Proof. Since STSTs are also STTs, the \( \leq \) direction of the proof is immediate.

We now prove the \( \Rightarrow \) direction. Given an STT \( S \) that only outputs strings over \( \Gamma^* \), 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 \( \eta \) is the conflict relation of \( S \), then the conflict relation \( \eta' \) 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 \( \eta'(x_{d},y_{d'}) \). \( \Box \)

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

\[
E_0 := \ 0 \mid x_0 \mid a\langle E_0 E_0 \rangle \mid E_1[E_0] \\
E_1 := \ ? \mid x_1 \mid a\langle E_0 E_1 \rangle \mid a\langle E_1 E_0 \rangle \mid E_1[E_1]
\]

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 $w = a_1 \ldots a_k$ over $\Sigma$ is viewed as an edge-labeled graph $G_w$ with $k + 1$ nodes $v_0 \ldots v_k$ 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:

$$\phi := a(x, y) \mid X(x) \mid x \xrightarrow{\sim} y \mid \phi \lor \phi \mid \neg \phi \mid \exists x.\phi \mid \exists X.\phi$$

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 $x \xrightarrow{\sim} y$ holds if the nodes $x$ and $y$ are connected by a nesting edge.

An MSO nested-word transducer $\Phi$ from input alphabet $\Sigma$ to output alphabet $\Gamma$ consists of a finite set $C$ called the copy set, node formulas $\phi^c$ for each $c \in C$, each of which is an MSO formula over nested words over $\Sigma$ with one free first-order variable $x$, and edge formulas $\phi^{c, d}$ and $\phi^{d, c}$ for each $a \in \Gamma$ and $c, d \in C$, each of which is an MSO formula over nested words over $\Sigma$ with two free first-order variables $x$ and $y$. Intuitively, given an input nested word, $\Phi$ 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 $\phi^c$ 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 $\phi^{c,d,a}$ 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 $\phi^c$ 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 $\phi^{c,d}$ 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 $\Gamma$ then $\llbracket \Phi \rrbracket (w) = u$, and otherwise $\llbracket \Phi \rrbracket (w)$ is undefined. A nested-word transduction $f$ from input alphabet $\Sigma$ to output alphabet $\Gamma$ is MSO-definable if there exists an MSO nested-word transducer $\Phi$ such that $\llbracket \Phi \rrbracket = f$.

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 $\varepsilon$-labeled edge from the $x_i$ node to the $x_i?$ node.
and from the $x_0?\node$ to the $x_0\node$. 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 $\varepsilon$-labeled edge from the $x_o?\node$ in column $i$ to the $x_o?\node$ in column $i - 1$, and an $\varepsilon$-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 $\varepsilon$-labeled edges between the $x_p\node$s in column $i + 1$ and the $x\node$s 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 $\varepsilon$ edges from the $x_p\node$s at position $j$ to the $x_p\node$s 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 $j$. 

Figure 5.4: Encoding of an STT’s computation in MSO.
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 $\triangleleft$ 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 $\varepsilon$ 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 $\varepsilon$) 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).

**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:

$$
nw.bt(\varepsilon) = 0;$$
$$
nw.bt(aw) = a\langle nw.bt(w), 0 \rangle;$$
$$
nw.bt(\langle a w_1 b \rangle w_2) = a\langle nw.bt(w_1), b\langle nw.bt(w_2), 0 \rangle \rangle.
$$

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:

- $nw.bt(\langle a a a \rangle) = a\langle a\langle 0, 0 \rangle, 0 \rangle, 0\rangle;$
- $nw.bt(\langle a a a \rangle = a\langle 0, 0 \rangle, a\langle 0, 0 \rangle \rangle.$

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 \( w = a_1 \ldots a_n \), after reading the symbol \( a_i \), \( x[0] \) contains the value of \( nw_{bt}(\text{WMS}(w, i)) \) (see Theorem 5.7 for the definition of \( \text{WMS}(w, i) \)). 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 \( F(q) = x[0] \). Next we define the update functions. For every \( a, b \in \Sigma \),

- \( \delta_i(q, a) = \delta_c(q, a) = (q, a), \) and \( \delta_r(q, a, b) = q; \)
- \( \rho_i(q, a, x) = x[a(?, 0)], \rho_c(q, a, x) = x, \) and \( \rho_r(q, a, b) = x^b[a(x[0], b(?), 0)] \).

**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 \( \epsilon \). For every symbol \( a \in \Sigma \),

**Right Child is 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 \( \rho((q_b, C), q_0, a, x_i) = ax_i^b, \) and \( \rho((q_b, IR), q_0, a, x_r) = x_i^b \).
• \( q' = (q_b, \text{IR}) \) for some \( b \in \Sigma \), then \( \rho((q_b, \text{IR}), q_0, a, x_i) = ax_i^l \), and \\
\( \rho((q_b, \text{IR}), q_0, a, x_r) = x_i^r \).

Right child is not 0: for every \( q_l, q_r \in Q \), \( \delta(q_l, q_r, a) = (q_b, \text{C}) \), and \( q_b \) must be of the form \( (q_b, \text{IR}) \). The variable update is defined as follows:

• if \( q_l = q_0 \), then \( \rho(q_0, (q_b, \text{IR}), a, x_c) = \langle ab \rangle x_r^i \);

• \( q_l = (q_c, \text{C}) \) for some \( c \in \Sigma \), then \( \rho((q_c, \text{C}), (q_b, \text{IR}), a, x_c) = \langle ax_i^l b \rangle x_r^i \);

• \( q_l = (q_c, \text{IR}) \) for some \( c \in \Sigma \), then \( \rho((q_c, \text{IR}), (q_b, \text{IR}), a, x_c) = \langle ax_i^l b \rangle x_r^i \).

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 \( \Sigma \) to binary trees over \( \Gamma \): 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 \(\Delta\). 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: \(\theta^*(0) = r_0\) and \(\theta^*(a(s_l, s_r)) = \theta(a, \theta^*(s_l), \theta^*(s_r))\). Assume that only the tree 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 \(T(s_0)\) denote the set of all subtrees of the input tree \(s_0\). Then, the output \(t\) at any step is a ranked tree over \((Q \times T(s_0)) \cup \Gamma \cup \{0\}\). 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 \(u = [q, s](t_1, \ldots, t_n)\), 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 \(t_1, \ldots, t_n\). 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 \(\chi\) 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 \(y_j\) by the output subtree \(t_j\). Then, in one step, the MTTR can replace the subtree \(u\) with the tree \(\chi\). 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 \(\Sigma\) 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 $f = \llbracket M \rrbracket$.

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 $\theta$. 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 $\delta_1$, and the variable update function $\rho_1$ of $S_1$ are as follows: if $\theta(r_1, r_2, a) = r$, then $\delta_1(r_1, r_2, a) = r$, and $\rho_1(r_1, r_2, a, x) = r(x_1 x_r)$. Finally, for every $r \in R$, $F(r) = x$. 
For every state $s$, we can now define the transition relation $\Delta(s)$ that describes the sequence of states processing the child in the tree. For every sequence $q$ in $Q$, the initial state $q_0 = q_0 \in Q^{\leq K}$ means that the root is only processed by $q_0$. The construction of $S_2$ then maintains the following invariants: assume $t_i$ and $t_r$ are the left and right children of a node, $m_i \in Q^{\leq K}$ is the sequence of states that processes $t_i$ in $M$, and $m_r \in Q^{\leq K}$ is the sequence of states that processes $t_r$ in $M$.

- before processing $t_i$, $S$ is in state $(m_i, m_r)$;
- before processing $t_r$ (after processing $t_i$), $S$ 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 $S_d(q, r_1)$, with $d \in \{l, r\}$, be the sequence of states processing the child $s_d$ in $A(q, r_1)$, assuming a linearization of such a tree. For every sequence $m = q_1 \ldots q_n \in Q^{\leq K}$, and symbol $r_1 \in R$, $S_d(m, r_1) = S_d(q_1, r_1) \ldots S_d(q_n, r_1)$. The STT $S_2$ only has one variable $x$.

We can now define the transition relation $\delta_2$, and the variable update function $\rho_2$ 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 $r \in R$:

1. $r_1$ is a call $r_2$: store the variables on the stack and update the state consistently:
   - $\delta_2(m', r_2) = (m_1, m')$, where $m_1 = (S_1(m', r_2), S_r(m', r_2))$, and
   - $\delta_2((m_1, m_2), r_2) = (m_3, (m_1, m_2))$, where $m_3 = (S_1(m_1, r_2), S_r(m_1, r_2))$;
   - $\rho_2((m_1, m_2), r_2, x) = x$, and $\rho_2((m_1, m_2), r_2, x) = x$;

2. $r$ is a return $r_2$: use the values on the stack to compute the labeling:
   - $\delta_2(m, (m_1, m_2), r_2) = m_2$, and $\delta_2(m, m', r_2) = m$;
   - $\rho_2(m, (m_1, m_2), r_2, x) = (\langle r, m_1 \rangle x (r, m_1))$, and
   - $\rho_2(m, m', r_2, x) = x^\rho (\langle r_2, m' \rangle x (r_2, m'))$.

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 $BT(R') \Rightarrow BT(\Gamma)$, 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 $\varepsilon$. The parameter set $\Pi_3$ 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

$$t_i(Y, (q^l_{i,1}, s_l), \ldots, (q^l_{i,b_i}, s_l), (q^r_{i,1}, s_r), \ldots, (q^r_{i,b_i}, s_r))$$

where $q^l_{i,1} \ldots q^l_{i,b_i}$ is the sequence of states processing the left subtree, and $q^r_{j,1} \ldots q^r_{j,b_j}$ 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 $m_l = q^l_{1,1} \ldots q^l_{1,i}, \ldots q^l_{1,1} \ldots q^l_{1,b_i}$ such that $|m_l| \leq K$. Moreover, we have that for all $x_i \in X_i$ (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 $\rho$ is copyless $\eta_3$ 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 $f : W_0(\Sigma) \rightarrow W_0(\Gamma)$ 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 $\Sigma$, and an STT $S$ from $\Sigma$ to $\Gamma$, consider the problem of computing the output $\llbracket S \rrbracket (w)$. 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 $\llbracket S \rrbracket (w)$ can be computed in time $O(k|w|)$ in a single pass.

**Proof.** A naive implementation of the transducer would cause the running time to be $O(k|w|^2)$ 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 $O(k|w|)$ since every variable might contain a string of length $O(|w|)$.

We now explain how we achieve the $O(k|w|)$ 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 \( L_{\text{pre}} \) of nested words over \( \Sigma \), a regular language \( L_{\text{post}} \) of nested words over \( \Gamma \), and an STT \( S \) from \( \Sigma \) to \( \Gamma \), the type-checking problem is to determine if \( \llbracket S \rrbracket (L_{\text{pre}}) \subseteq L_{\text{post}} \) (that is, if for every \( w \in L_{\text{pre}} \), \( \llbracket S \rrbracket (w) \in L_{\text{post}} \)).

**Theorem 5.26 (Type-checking).** Given an STT \( S \) from \( \Sigma \) to \( \Gamma \), an NWA \( A \) accepting nested words over \( \Sigma \), and an NWA \( B \) accepting nested words over \( \Gamma \), checking \( \llbracket S \rrbracket (L(A)) \subseteq L(B) \) is solvable in \( \mathsf{EXP} \mathsf{TIME} \), and more precisely in time \( O(|A|^3 \cdot |S|^3 \cdot n^{kn^2}) \) 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 \( \llbracket S \rrbracket (w) \) 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 \( \Lambda \);
  - if \( B \) reads \( w_2 \) starting in state \( q_2 \), and with stack \( \Lambda \), 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 \( \Sigma \) to output alphabet \( \Gamma \), and a language \( L \subseteq W_0(\Gamma) \) of output nested words, the set \( \text{PreImg}(L, S) \) consists of input nested words \( w \) such that \( \llbracket S \rrbracket (w) \in L \).
**Theorem 5.27** (Computing pre-image). Given an STT $S$ from $\Sigma$ to $\Gamma$, and an NWA $B$ over $\Gamma$, there is an EXPTIME algorithm to compute an NWA $A$ over $\Sigma$ such that $L(A) = \text{PreImg}(L(B), S)$. The NWA $A$ has size $O(|S|^3 \cdot n^{kn^2})$ 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 $\llbracket S \rrbracket \neq \llbracket S' \rrbracket$.

**Proof.** Two STTs $S_1$ and $S_2$ are inequivalent if one of the following holds:

1. for some input $u$, only one of $\llbracket S_1 \rrbracket(u)$ and $\llbracket S_2 \rrbracket(u)$ is defined;
2. for some input $u$, the lengths of $\llbracket S_1 \rrbracket(u)$ and $\llbracket S_2 \rrbracket(u)$ are different;
3. for some input $u$, there exist two symbols $a$ and $b$, such that $a \neq b$, $\llbracket S_1 \rrbracket(u) = u_1au_2$, $\llbracket S_2 \rrbracket(u) = v_1bv_2$, 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 $[\text{MOS04}]$ 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 $\Sigma$ to $\{0\}$ such that $V_1$ produces $0^n$ on input $u$, if $[S_1](u) = u_1au_2$ and $|u_1| = n$. Similarly, given $S_2$, and a symbol $b \neq a$, we construct a VPT $V_2$ from $\Sigma$ to $\{1\}$ such that $V_2$ produces $1^n$ on input $u$, if $[S_2](u) = u_1bu_2$ 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 $a_1a_2a_3$, $S_2$ outputs $b_1b_2b_3$, and $|a_1| = |b_2|$.

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 $V$ $[\text{Esp97, SSMH04}]$. A solution to such semi-linear set can be found in NP (in the number of states of $V$) $[\text{KT10}]$. 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 $\text{NExpTime}$.

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 $u$, $[S_1](u) = [S'_1](u\#)$.

Each state of $V_1$ is a pair $(q_f, f)$, where $q_f$ is a state of $S'_1$, and $f$ is a partition of the variables $X$ of $S'_1$ into 6 categories $\{l, m1, m2, r, n\}$, such that a variable $x$ is in the set:

1: 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 \( a \in E(X,\Sigma) \) (i.e. an assignment right hand side), we use \( x \in_a a \) to say that a variable \( x \in X \) appears in \( a \).

\( b \) 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 \( \rho_i(q,b,x) = a_1a_2?a_3 \), 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:

   - \( \forall y \in_a a_1, y \in f_l, \forall y \in_a a_2a_3, y \in f_r, f'_{m1} = \{x\} \), 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 \( f'_{m'} \);
   - if \( p_l \neq p_r \), then \( y \) belongs to \( f'_{m2} \).
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 $a$ and in $\{\rho_i(q, b, y) | y \in f'_i\}$.

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 $b'$: 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 $\Sigma$ 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 $\delta^1$ and $\delta^2$ 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 $\delta$ of $V$ is defined as
  - if $\delta^1(q_1, a) = q'_1, 0^m$ and $\delta^2(q_2, a) = q'_2, 1^m$, then $\delta((q_1, q_2), a) = (q'_1, q'_2), 0^m1^m$;
If the number of variables is bounded, then the size of $V$ is polynomial, and this gives an upper bound of $\text{NP}$. For the transducers that map strings to nested words, that is, for streaming string-to-tree transducers (SSTT), the above construction yields a $\text{PSPACE}$ bound \cite{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 $\llbracket S \rrbracket = \llbracket S' \rrbracket$ is solvable in $\text{PSPACE}$.

Improving these bounds remains a challenging open problem.

\section*{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.

\subsection*{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) \cite{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) \cite{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 \cite{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 \cite{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 \cite{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 \cite{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 \cite{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 \cite{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 lookahead, 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 \([\text{RS09}]\) and Macro forest transducers \([\text{PS04}]\)). 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, HMN14] (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 MTIs 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, DVLMT14]. 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
Future work

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 [ARI3].

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
Future work

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. 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

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


