Streaming Tree Transducers

RAJEEV ALUR, University of Pennsylvania
LORIS D'ANTONI, University of Pennsylvania

A

Theory of tree transducers provides a foundation for understanding expressiveness and complexity of analysis problems for specification languages for transforming hierarchically structured data such as XML documents. We introduce *streaming tree transducers* as an analyzable, executable, and expressive model for transforming unranked ordered trees (and forests) in a single pass. Given a linear encoding of the input tree, the transducer makes a single left-to-right pass through the input, and computes the output in linear time using a finite-state control, a visibly pushdown stack, and a finite number of variables that store output chunks that can be combined using the operations of string-concatenation and tree-insertion. We prove that the expressiveness of the model coincides with transductions definable using monadic second-order logic (MSO). Existing models of tree transducers either cannot implement all MSO-definable transformations, or require *regular look-ahead* that prohibits single-pass implementation. We show a variety of analysis problems such as *type-checking* and checking *functional equivalence* are decidable for our model.

Categories and Subject Descriptors: F.2.5.1 [**Theory of Computation**]: Formal languages and automata theory, Formalisms, Automata extensions, Transducers

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Nested words, tree transducers

ACM Reference Format:
J. ACM V, N, Article A (January YYYY), 43 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Finite-state machines and logics for specifying tree transformations offer a suitable theoretical foundation for studying expressiveness and complexity of analysis problems for languages for processing and transforming XML documents. Representative formalisms for specifying tree transductions include finite-state top-down and bottom-up tree transducers, Macro tree transducers (MTT), attribute grammars, MSO (monadic second-order logic) definable graph transductions, and specialized programming languages such as XSLT and XDuce [Comon et al. 2002; Courcelle 1994; Engelfriet and Maneth 1999; Engelfriet and Vogler 1985; Milo et al. 2000; Hosoya and Pierce 2003; Martens and Neven 2005; Hosoya 2011; Engelfriet 1975].

In this paper, we propose the model of *streaming tree transducers* (STT) which has the following three properties: (1) *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; (2) *Expressiveness:* STTs specify exactly the class of MSO-definable tree transductions; and (3) *Analyzability:* decision problems such as type-checking and checking functional equivalence of two STTs, are decidable. The last two features indicate that our model has the commonly accepted trade-off between analyzability and expressiveness in formal language theory. The motivation for designing streaming algorithms that can process a document in a single pass has led to streaming models for checking membership in a regular tree language and for querying [Segoufin and Vianu 2002; Neven and Schwentick 2002; Milo et al. 2000; Madhusudan and Viswanathan 2009], but there is no previous model that can compute all MSO-definable transformations in a single pass (see Section 6 for detailed comparisons of STTs with prior models).

This research was partially supported by NSF Expeditions in Computing award CCF 1138996.
Authors addresses: R. Alur, Computer Science Department, University of Pennsylvania; L. D'Antoni, Computer Science Department, University of Pennsylvania.

DOI:http://dx.doi.org/10.1145/0000000.0000000

The transducer model integrates features of *visibly pushdown automata*, equivalently *nested word automata* [Alur and Madhusudan 2009], and *streaming string transducers* [Alur and Cerný 2010; 2011]. 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. 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.

To understand the novel features of our model, let us consider two kinds of transformations. First, suppose we want to select and output the sequence of all subtrees that match a pattern, that is specified by a regular query over the entire input, and not just the prefix read so far. To implement this query, the transducer uses multiple variables to store alternative outputs, and exploiting regularity to maintain only a bounded number of choices at each step. Second, suppose the transformation requires *swapping* of subtrees. The operations of concatenation and tree-insertion allows an STT to implement this transformation easily. This ability to combine previously computed answers seems to be missing from existing transducer models. We illustrate the proposed model using examples such as reverse, swap, tag-based sorting, where the natural single-pass linear-time algorithms for implementing these transformations correspond to STTs.

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 paper 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 [Engelfriet and Maneth 1999]. Our proof proceeds by establishing two key properties of STTs: the model is 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 that can be updated by assignments that obey the single-use-restriction rules, and we develop them in a modular fashion by introducing intermediate results (for example, we establish that allowing variables to range over trees that contain multiple parameters that can be selectively substituted during updates, does not increase expressiveness).

We show a variety of analysis questions for our transducer model to be decidable. Given a regular language L_1 of input trees and a regular language L_2 of output trees, the *type-checking* problem is to determine if the output of the transducer on an input in L_1 is guaranteed to be in L_2. We establish an ExpTime upper bound on type-checking. For checking functional equivalence of two streaming tree transducers, we show that if the two transducers are inequivalent, then we can construct a pushdown automaton A over the alphabet $\{0, 1\}$ such that A accepts a string with equal number of 0's and 1's exactly when there is an input on which the two transducers compute different outputs. Using known techniques for computing the Parikh images of context-free languages [Engelfriet and Maneth 2006; Seidl et al. 2004; Esparza 1997; Kopczynski and To 2010], this leads to a NExpTime upper bound for checking functional inequivalence of two STTs. Assuming

a bounded number of variables, the upper bound on the parametric complexity becomes NP. Improving the NExpTime bound remains a challenging open problem.

2. TRANSDUCER MODEL

We first introduce some preliminary notions, then formally define streaming tree transducers, and finally provide few examples.

2.1. Preliminaries

In this section we first recall the notion of nested word and then introduce the basic building blocks of streaming tree transducers.

Nested Words. Data with both linear and hierarchical structure can be encoded using nested words [Alur and Madhusudan 2009]. Given a set Σ of symbols, the *tagged alphabet* $\hat{\Sigma}$ consists of the symbols a, $\langle a$, and $a\rangle$, for each $a \in \Sigma$. A *nested word* over Σ is a finite sequence over $\hat{\Sigma}$. For a nested word $a_1 \cdots a_k$, a position j, for $1 \leq j \leq k$, is said to be a *call* position if the symbol a_j is of the form $\langle a$, a *return* position if the symbol a_j is of the form $a\rangle$, and an *internal* position otherwise. The tags induce a natural matching relation between call and return positions, and in this paper, we are interested only in *well-matched* nested words in which all calls/returns have matching returns/calls. A string over Σ is a nested word with only internal positions. Nested words naturally encode ordered trees. The empty tree is encoded by the empty string ε. The tree with a-labeled root with subtrees $t_1, \ldots t_k$ as children, in that order, is encoded by the nested word $\langle a \, \langle\!\langle t_1 \rangle\!\rangle \cdots \langle\!\langle t_k \rangle\!\rangle \, a\rangle$, where $\langle\!\langle t_i \rangle\!\rangle$ is the encoding of the subtree t_i. This transformation can be viewed as an inorder traversal of the tree. The encoding extends to *forests* also: the encoding of a forest is obtained by concatenating the encodings of the trees it contains. An a-labeled leaf corresponds to the nested word $\langle aa\rangle$, we will use $\langle a\rangle$ as its abbreviation. Thus, a binary tree with a-labeled root for which the left-child is an a-labeled leaf and the right-child is a b-labeled leaf is encoded by the string $\langle a \, \langle a\rangle \, \langle b\rangle \, a\rangle$.

Nested Words with Holes. A key operation that our transducer model relies on is *insertion* of one nested word within another. In order to define this, we consider nested words with holes, where a hole is represented by the special symbol ?. For example, the nested word $\langle a \, ? \, \langle b\rangle \;\; a\rangle$ represents an incomplete tree with a-labeled root where its 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 a binary type 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 one hole. We can view a type-1 nested word as a unary function from nested words to nested words. The set $W_0(\Sigma)$ of type-0 nested words over the alphabet Σ is defined by the grammar

$$W_0 \;:=\; \varepsilon \mid a \mid \langle a \, W_0 \, b\rangle \mid W_0 \, W_0 \qquad \text{for } a, b \in \Sigma$$

The set $W_1(\Sigma)$ of type-1 nested words over the alphabet Σ is defined by the grammar

$$W_1 \;:=\; ? \mid \langle a \, W_1 \, b\rangle \mid W_1 \, W_0 \mid W_0 \, W_1 \qquad \text{for } a, b \in \Sigma$$

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

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

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

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

is obtained by replacing each variable x by $\alpha(x)$, and applying the substitution: in particular, $\alpha(e[e'])$ is obtained by replacing the symbol ? in the type-1 nested word $\alpha(e)$ by the nested word $\alpha(e')$.

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

At a return, the transducer assigns the values to its variables X using the values popped from the stack as well as the values returned. For each variable x, we will use x_p to refer to the popped value of x. Thus, each variable x is updated using an expression over the variables $X \cup X_p$. The conflict relation η extends naturally to variables in X_p: $\eta(x_p, y_p)$ holds exactly when $\eta(x, y)$ holds. However $\eta(x, y_p)$ does not hold for any $x \in X$ and $y_p \in X_p$. Then, the update at a return is specified by assignments in $\mathcal{A}(X, X \cup X_p, \eta, \Sigma)$.

2.2. 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. Such variables are equipped with a conflict relation that restricts which variables can be combined, and 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 variable values 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 nested 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.

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

— a finite set of states Q;
— a finite set of stack symbols P;
— an initial state $q_0 \in Q$;
— a finite set of typed variables X with a reflexive symmetric binary conflict relation η;
— a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that each expression $F(q)$ is consistent with η;
— an internal state-transition function $\delta_i : Q \times \Sigma \mapsto Q$;
— a call state-transition function $\delta_c : Q \times \Sigma \mapsto Q \times P$;
— a return state-transition function $\delta_r : Q \times P \times \Sigma \mapsto Q$;
— an internal variable-update function $\rho_i : Q \times \Sigma \mapsto \mathcal{A}(X, X, \eta, \Gamma)$;
— a call variable-update function $\rho_c : Q \times \Sigma \mapsto \mathcal{A}(X, X, \eta, \Gamma)$; and
— a return variable-update function $\rho_r : Q \times P \times \Sigma \mapsto \mathcal{A}(X, X \cup X_p, \eta, \Gamma)$.

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

(1) **Internal transitions:** if a is internal, and $\delta_i(q, a) = q'$, then $\delta((q, \Lambda, \alpha), a) = (q', \Lambda, \alpha')$, where
   — $q'$is the state resulting from applying the internal transition that reads a in state q,
   — the stack Λ remains unchanged, and
   — the new evaluation function $\alpha' = \alpha \cdot \rho_i(q, a)$ is the result of applying the variable update function $\rho_i(q, a)$ using the variable values in α.

(2) **Call transitions:** if for some $b \in \Sigma$, $a = \langle b$, and $\delta_c(q, b) = (q', p)$, then $\delta((q, \Lambda, \alpha), a) = (q', \Lambda', \alpha_0)$, where
   — q' is the state resulting from applying the call transition that reads b in state q;
   — $\Lambda' = (p, \alpha \cdot \rho_c(q, b))\Lambda$ is the new stack resulting from pushing the pair (p, α') on top of the old stack Λ, where the stack state p is the one pushed by the call transition function, and $\alpha' = \alpha \cdot \rho_c(q, b)$ is the new evaluation function α' resulting from applying the variable update function $\rho_c(q, b)$ using the variable values in α, and
   — α_0 is the evaluation function that sets every variable to their initial value.

(3) **Return transitions:** if for some $b \in \Sigma$, $a = b\rangle$, and $\delta_r(q, p, b) = q'$, then $\delta((q, (p, \beta)\Lambda, \alpha), a) = (q', \Lambda', \alpha')$ where
   — q' is the state resulting from applying the return transition that reads b in state q with p on top of the stack,
   — the stack $\Lambda' = \Lambda$ is the result of popping the top of the stack value from the current stack, and
   — the new evaluation function $\alpha' = \alpha \cdot \beta_p \cdot \rho_r(q, p, b)$, where β_p is the valuation for variables X_p defined by $\beta_p(x_p) = \beta(x)$ for $x \in X$, is the result of applying the variable update function $\rho_r(q, p, b)$ using the variable values in α, and the stack variable values in β_p.

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

2.3. 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, swap, and sorting based on fixed number of tags. In each of these cases, the transducer mirrors the natural algorithm for implementing the desired operation in a single pass.

Reverse. Given a nested word $a_1 a_2 \cdots a_k$, its *reverse* is the nested word $b_k \cdots b_2 b_1$, where for each $1 \leq j \leq k$, $b_j = a_j$ if a_j is an internal symbol, $b_j = \langle a$ if a_j is a return symbol $a\rangle$, and $b_j = a\rangle$ if a_j is a call symbol $\langle a$. As a tree transformation, *reverse* corresponds to recursively reversing the order of children at each node: the reverse of $\langle a \, \langle b \, \langle d\rangle \, \langle e\rangle \, b\rangle \, \langle c\rangle \, a\rangle$ is $\langle a \, \langle c\rangle \, \langle b \, \langle e\rangle \, \langle d\rangle \, b\rangle \, a\rangle$. This transduction can be implemented by a streaming tree transducer with a single state, a single type-0 variable x, and stack symbols Σ: the internal transition on input a updates x to $a\,x$; the call transition on input a pushes a onto the stack, stores the current value of x on the stack, and resets x to the empty nested word; and the return transition on input b, while popping the symbol a and stack value x_p from the stack, updates x to $\langle b\,x\,a\rangle\, x_p$.

Tree Swap. Figure 1 shows the 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. For clarity of presentation, we assume that the input nested word encodes a tree: it does not contain any internal symbols and if a call position is labeled $\langle a$ then its matching return is labeled $a\rangle$.

The initial state is q_0 which means that the transducer has not yet encountered a b-label. In state q_0, the STT records the tree traversed so far using a type-0 variable x: upon an a-labeled call, x is stored on the stack, and is reset to ε; and upon an a-labeled return, x is updated to $x_p\langle a\,x\,a\rangle$. In state q_0, upon a b-labeled call, the STT pushes q_0 along with the current value of x on the stack, resets x to ε, and updates its state to q'. In state q', the STT constructs the first b-labeled subtree t_1 in the variable x: as long as it does not pop the stack symbol q_0, at a call it pushes q' and x, and at a return, updates x to $x_p\langle a\,x\,a\rangle$ or $x_p\langle b\,x\,b\rangle$,

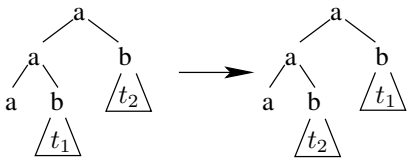

Fig. 1. Tree Swap

depending on whether the current return symbol is a or b. When it pops q_0, it updates x to $\langle b\, x\, b\rangle$ (at this point, x contains the tree t_1, and its value will be propagated), sets another type-1 variable y to $x_p\,?$, and changes its state to q_1. 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 the type-1 variable y. At a b-labeled call, it pushes q_1 along with y on the stack, resets x to ε, and updates the state to q'. Now in state q', the STT constructs the second b-labeled subtree t_2 in variable x as before. When it pops q_1, the subtree t_2 corresponds to $\langle b\, x\, b\rangle$. The transducer updates x to $y_p[\langle b\, x\, b\rangle]x_p$ capturing the desired swapping of the two subtrees t_1 and t_2 (the variable y is no longer needed and is reset to ε to ensure the single use restriction), and switches to state q_2. In state q_2, the remainder of the tree is traversed adding it to x. The output function is defined only for the state q_2 and maps q_2 to x.

Tag-based Sorting. Suppose that given a sequence of trees $t_1 t_2 \cdots t_k$ (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.

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 defined as follows:

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

Informally, while going top-down, f swaps the first two children until it reaches for the first time a node for which the third child is a symbol different from b. The third child is always deleted in the process. The STT implementing such transduction uses four variables x_1, y_1, x_2, y_2. Given a tree t, let $\mathrm{c}_i(t)$ be its i-th child. After finishing processing the first two children $t_1 = \mathrm{c}_1(t)$ and $t_2 = \mathrm{c}_2(t)$ of a tree t, 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 t_1 and t_2.

When starting processing the third child $t_3 = \mathrm{c}_3(t)$ of a node t 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 the 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 is the first child of some node t'. In this case, since t_3 is labeled with b we have that $f(t) = \langle s\ t_2\ t_1\ s\rangle$ and $t = \langle s\ t_1\ t_2\ s\rangle$. In order to maintain the invariants that $x_i = f(t_i)$ and $y_i = t_i$, 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 is the first child of some node t'. Following the same reasoning as for the previous case the variables are updated as follows: $x_1 := x_1^p$, $x_2 := \langle s\ y_2\ y_1\ s\rangle$, $y_1 := y_1^p$, $y_2 := \langle s\ y_1\ y_2\ s\rangle$;

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

3. 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 3.1 (LINEAR-BOUNDED OUTPUTS). *For an STT-definable transduction f from Σ to Γ, for all nested words $w \in W_0(\Sigma)$, $|f(w)| = O(|w|)$.*

We now examine some of the features in the definition of STTs in terms of how they contribute to the expressiveness. First, having multiple variables is essential, and this follows from results on streaming string transducers [Alur and Cerný 2010; 2011]. 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 classical finite-state automata over strings.

3.1. 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 definition of NWAs and regular nested-word languages does not need the input nested word to be well-matched (that is, the input is a string over $\hat{\Sigma}$), but this distinction is not relevant for our purpose. A nested word automaton A over an input alphabet Σ is specified by a finite set of states Q; a finite set of stack symbols P; an initial state $q_0 \in Q$; a set $F \subseteq Q$ of accepting states; an internal state-transition function $\delta_i : Q \times \Sigma \mapsto Q$; a call state-transition function $\delta_c : Q \times \Sigma \mapsto Q \times P$; and a return state-transition function $\delta_r : Q \times P \times \Sigma \mapsto Q$. A language $L \subseteq W_0(\Sigma)$ of nested words is *regular* if it is accepted by such an automaton. This class includes all regular string languages, regular tree languages, and is a subset of deterministic context-free languages [Alur and Madhusudan 2009].

Given a nested-word transduction f from input alphabet Σ to output alphabet Γ, the *domain* of f is the set $Dom(f) \subseteq W_0(\Sigma)$ of input nested words w for which $f(w)$ is defined, and the *image* of f is the set $Img(f) \subseteq W_0(\Gamma)$ of output nested words w' such that $w' = f(w)$ for some w. It is easy to establish that for STT-definable transductions, the domain is a regular language, but the image is not necessarily regular:

PROPOSITION 3.2 (DOMAIN-IMAGE REGULARITY). *For an STT-definable transduction f from Σ to Γ, $Dom(f)$ is a regular language of nested words over Σ. There exists an STT-definable transduction f from Σ to Γ, such that $Img(f)$ is not a regular language of nested words over Γ.*

Proof. Given an STT-definable transduction f from Σ to Γ, let $A_f = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be an STT defining it. The NWA A accepting the domain $Dom(f)$ of f has set of states $Q' = Q$, initial state $q'_0 = q_0$, set of stack states $P' = P$, set of final states $F' = \{q \mid F(q) \text{ is defined}\}$, and transition function $\delta' = \delta$.

We now construct an STT B from $\Sigma = \{a, b\}$ to $\Gamma = \{a, b\}$ computing a function f' for which the image $Img(f')$ is not regular. The STT B only has one state q which is also initial and only has transition function $\delta_i(q, a) = \delta_i(q, b) = q$, and has only one type-0 variable x that is updated as follows: $\rho_i(q, a, x) = \delta_i(q, b, x) = axb$. The output function F_B of B is defined as $F_B(q) = x$. The STT B computes the following transduction f': if the input word w has length n, B outputs the word $a^n b^n$. The image $Img(f')$ of f' is the language $\{a^n b^n \mid n \geq 0\}$, which is not a regular language of nested words over Γ. □

3.2. Copyless STTs

When the conflict relation η is purely reflexive (i.e. $\{(x, x) \mid x \in X\}$) we call an STT *copyless*. The set of copyless assignments from Y to X is denoted by $\mathcal{A}(X, Y, \Sigma)$ where we drop the relation η. We now define the notion of atomic assignment that will be fundamental in many proofs.

Definition 3.3. A copyless assignment $\rho \in \mathcal{A}(X, X \cup Y, \Sigma)$ is *atomic* iff it has one of the following forms:

Reset: for some variable $x \in X$, and some $a, b \in \Sigma$, $x := \varepsilon$, $x :=?$, $x := \langle a?b \rangle$, or $x := a$, and for every variable $y \in X$, if $y \neq x$, then $y := y$;

Concatenation: for some two distinct variables $x, y \in X$, $x := xy$ or $x := yx$, $y := \varepsilon$ or $y :=?$, and for every variable $z \in X$, if $z \neq x$ and $z \neq y$, then $z := z$;
Substitution: for some two distinct variables $x, y \in X$, $x := x[y]$ or $x := y[x]$, $y :=?$ or $y := \varepsilon$, and for every variable $z \in X$, if $z \neq x$ and $z \neq y$, then $z := z$; and
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 3.4. *For every copyless assignment $\rho \in \mathcal{A}(X, X, \Sigma)$ there exists a set of variables Y disjoint from X, and a sequence of assignments $s = \rho_1, \ldots, \rho_n$ such that:*

(1) for every variable $x \in X$, $s(x) = \rho(x)$,
(2) the assignment ρ_1 belongs to $\mathcal{A}(X \cup Y, X, \Sigma)$ and it is atomic, and
(3) for every $2 \leq j < n$, the assignment ρ_j belongs to $\mathcal{A}(X \cup Y, X \cup Y, \Sigma)$, and it is atomic.

Proof. We sketch the proof and gloss over the fact the variables can be of both type-0 and type-1. Given an expression $e \in E = E_0 \cup E_1$ we define the size of e, $\text{SIZE}(e)$, as the size of its parse tree:

— if $e \in \{\varepsilon, a, \langle a?b\rangle, ?\}$, then $\text{SIZE}(e) = 1$;
— if for some e_1 different from ?, $e = \langle ae_1b\rangle$, then $\text{SIZE}(e) = 1 + \text{SIZE}(e_1)$; and
— 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 $\text{SIZE}(\rho) \in \mathbb{N} \times \mathbb{N}$ to be the value (a, b) such that $a = max_{x \in X_1} \text{SIZE}(\rho(x))$ is the maximum size of a expression on the right hand side of a variable, and b is the number of variables for which the right-hand side is an expression of size a. We define the order between two pairs $(a, b), (c, d) \in \mathbb{N}^2$, as $(a, b) < (c, d)$ if $a < c$, or if $a = c$ and $b < d$.

Given an assignment ρ that is not atomic, we show that ρ can be always transformed into a sequence of atomic assignments $s = \rho_1 \ldots \rho_n$ such that for every $x \in X$ $s(x) = \rho(x)$. The new assignments can have new variables. We now proceed by induction on $size(\rho) = (s_1, s_2)$:

— if $s_1 = 1$ we have the following possibilities:
  — ρ is atomic, then we are done; or
  — there exist a set of variables X', for which every variable $x \in X'$ has a variable $y_x \neq x$ as right-hand side. Replace ρ with a sequence of atomic swap assignments, so that the resulting variable permutation is the same as for ρ.
— if $s_1 > 1$ we have the following possibilities:
  — one variable x has right-hand side $e = w_1w_2$, and e has size s_1. Replace ρ with the sequence $\rho_1\rho_2$, such that $\rho_1, \rho_2 \in \mathcal{A}(X \cup \{v\}, X \cup \{v\}, \Sigma)$: $\rho_1(x) = w_1$, $\rho_1(v) = w_2$, $\rho_2(x) = xv$, $\rho_2(v) = \varepsilon$, and for each y such that $y \neq x$ and $y \neq n$, $\rho_1(y) = \rho(y)$ and $\rho_2(y) = y$. We then have that ρ_2 is atomic, and since w_1 and w_2 both have smaller size than e, $size(\rho_1)$ is smaller than $size(\rho)$. We now apply the IH, and obtain that there exists a set X_1 and a sequence of assignments $s' = \rho'_1 \cdots \rho'_k$ over $\mathcal{A}(X \cup X_1 \cup \{v\}, X \cup X_1 \cup \{v\}, \Sigma)$, such that for every $x \in X \cup \{v\}$ (and in particular for every $x \in X$), $s'(x) = \rho_1(x)$. We can now build our final sequence ρ_f as $\rho'_1 \cdots \rho'_k\rho''_2$, such that for each $x \in X_1$, $\rho''_2(x) = x$. The assignment ρ''_2 is atomic, therefore we are done.
  — one variable x has right-hand side $e = w_1[w_2]$, and e has size s_1. Replace ρ with the sequence $\rho_1\rho_2$ of assignment over $X \cup \{n\}$ such that: $\rho_1(x) = w_1$, $\rho_1(n) = w_2$, $\rho_2(x) = x[n]$, $\rho_2(x) = \varepsilon$, and for each y such that $y \neq x$ and $y \neq n$, $\rho_1(y) = \rho(y)$ and $\rho_2(y) = y$. We then have that ρ_2 is atomic, and since w_1 and w_2 both have smaller size than e, $size(\rho_1)$ is smaller than $size(\rho)$. By IH ρ_1 can be transformed into a sequence of assignments that is atomic, and using the same technique as before we can build our final sequence of assignments.
  — one variable x has right-hand side $e = \langle awb\rangle$, e has size s_1, and $w \neq ?$. Replace ρ with the sequence $\rho_1\rho_2$ of assignment over $X \cup \{n\}$ such that: 1) $\rho_1(x) = \langle a?b\rangle$, $\rho_1(n) = w$, and for each $y \in X$, if $y \neq x$, $\rho_1(y) = y$, 2) $\rho_2(x) = x[n]$, $\rho_2(n)\varepsilon$, and for each $y \in X$, such that $y \neq x$, $\rho_3(y) = y$. The assignment ρ_2 is atomic. Since w has smaller size than e, $size(\rho_1)$ is smaller than $size(\rho)$. By IH ρ_1 can be transformed into a sequence of assignments that is atomic, and using the same technique as before we can build our final sequence of assignments.

This concludes the proof. □

We can then further refine the lemma.

COROLLARY 3.5. *For every copyless assignment $\rho \in \mathcal{A}(X, X \cup Y, \Sigma)$, with X disjoint from Y, there exists a set of variables$Z$ disjoint from $X \cup Y$, and a sequence of assignments $s = \rho_1, \ldots, \rho_n$ such that:*

(1) for every variable $x \in X$, $s(x) = \rho(x)$,
(2) the assignment ρ_1 belongs to $\mathcal{A}(X \cup Y \cup Z, X \cup Y, \Sigma)$ and it is atomic,
(3) for every $2 \leq j \leq n$, the assignment ρ_j belongs to $\mathcal{A}(X \cup Y \cup Z, X \cup Y \cup Z, \Sigma)$, and it is atomic, and
(4) the assignment ρ_n belongs to $\mathcal{A}(X, X \cup Y \cup Z, \Sigma)$ and it is atomic.

Moreover, if two copyless assignments are composed, the resulting assignment is still copyless.

LEMMA 3.6. *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 exist a variable $x \in X$, such that x appears twice in the right hand side of ρ_1. This means that there exists two variables $y_1, y_2 \in Y$ appearing in the right hand side of ρ', such that $\rho(y_1)$ and $\rho(y_2)$ contain the two incriminated occurrences of x. If $y_1 \neq y_2$, the assignmentρ cannot be copyless, since it would contain two occurrences of x. If $y_1 = y_2$, ρ' cannot be copyless, since it would contain two occurrences of y_1. □

3.3. 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 [Alur and Madhusudan 2009]. 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 3.7 (BOTTOM-UP STTS). *Every STT-definable transduction is definable by a bottom-up STT.*

Proof. Let $S = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be an STT. We construct an equivalent bottom-up STT $S' = (Q', q_0', P', X', \eta', F', \delta', \rho')$. Intuitively, S' delays the application of a call transition of S to the corresponding return. This is done by computing a summary of all possible executions of S on the subword between a call and the corresponding matching return. At the return this summary can be combined with the information stored on the stack to continue the summarization.

Auxiliary Notions. Given a nested word $w = a_1 a_2 \ldots a_n$, for each position $1 \leq i \leq n$, let $\text{WMS}(w, i)$ be the longest well-matched subword $a_j \ldots a_i$ ending at position i. Formally, given a well-matched nested word $w = a_1 a_2 \ldots a_n$, we define $\text{WMS}(w, i)$ as follows:

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

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

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

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 $\text{WMS}(w, i)$ starting in state q, it reaches the state q'. The initial state of S' is the identity function f_0 mapping each state $q \in Q$ to itself. A stack state in P' is a pair (f, a) where f is a function mapping Q to Q and a is a symbol in Σ.

Next, we define the transition relation δ'. When reading an internal symbol a, starting in state f, S' goes to state f' such that for each $q \in Q$, $f'(q) = \delta_i(f(q), a)$. When reading a call symbol $\langle a$, starting in state f, S' stores f on the stack along with the symbol a and goes to state f_0. When reading a return symbol $b\rangle$, starting in state f, and with (f', a) on top of the stack, S' goes to state f'' defined as: for each $q \in Q$, if $f'(q) = q_1$ (the state reached by S when reading $\text{WMS}(w, \text{LC}(w, i-1))$ starting in q), $\delta_c(q_1, \langle a) = (q_2, p)$, and $f(q_2) = q_3$ (the state reached by S when reading $\text{WMS}(w, i-1,)$ starting in q_2), then $f''(q) = \delta_r(q_3, p, b)$.

Variable Updates and Invariants. We now explain how S' achieves the summarization of the variable updates of S. For each variable $x \in X$, and state $q \in Q$, X' contains a variable x_q. After reading the i-th symbol, x_q contains the value of x computed by S, when reading $\text{WMS}(w, i)$ starting in state q. Given an assignment $\alpha \in \mathcal{A}(X, X, \Sigma) \cup \mathcal{A}(X, X \cup X_p, \Sigma)$, a state $q \in Q$, and a set of variables $Y \subseteq X \cup X_p$, we define $\text{SUB}_{q,Y}(\alpha)$ to be the assignment $\alpha' \in \mathcal{A}(X', X', \Sigma) \cup \mathcal{A}(X', X' \cup X'_p, \Sigma)$, where $X' = (X \setminus Y) \cup Y'$ with $Y' = \{y_q \mid y \in Y\}$, and each variable $y \in Y$ is replaced by $y_q \in Y'$.

Initially, and upon every call, each variable x_q is assigned the value ? or ε, if x is a type-1 or type-0 variable respectively. We use $e\{x/x'\}$ to denote the expression e in which every variable x is replaced with the variable x'. When processing the input symbol a, starting in state f, each variable x_q is updated as follows:

a is internal:	if $f(q) = q'$, then $x_q := \text{SUB}_{q,X}(\rho_i(q', a, x))$ 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, $x_q := x_q$; and
$a = b\rangle$ is a return:	if (f', c) is the state popped from the stack, $f'(q) = q_1$, $\delta_c(q_1, c) = (q_2, p)$, and $f(q_2) = q_3$, then $x_q := \text{SUB}_{q_2,X}(\text{SUB}_{q,X_p}(\rho_r(q_3, b, p, x)\{y_p/\rho_c(q_1, c, y)\{z/z_p\}\}))$ is the result of applying the the call variable update function in state q_1, followed by the return variable update function of S in state q' where each variable $x \in X$ is renamed to x_{q_2}, and each variable $x \in X_p$ is renamed to x_q.

Output Function. The output function F' of S' is defined as follows: for each state $f \in Q'$, $F'(f) = \text{SUB}_{q_0,X}(F(f(q_0)))$ is the result of applying the output function of S in state $f(q_0)$ where each variable $x \in X$ is renamed to x_{q_0}.

Conflict Relation. The conflict relation η' contains the following rules:

(1) Variable summarizing different states are in conflict: for all $x, y \in X$, for all $q \neq q' \in Q$, $\eta'(x_q, y_{q'})$; and
(2) Variables that conflict in S, also conflict in S' for every possible summary: for all $q \in Q$, for all $x, y \in X$, if $\eta(x, y)$, then $\eta'(x_q, y_q)$.

Next, we prove that η' is consistent with the update function ρ'. We assume the current state $f \in Q'$ to be fixed. We first show that two conflicting variables never appear in the same right hand side. Each assignment of S' is of the form $\text{SUB}_{q,Y}(\rho(q, a, x))$. Therefore if no variables of S are conflicting in $\rho(q, a, x)$ w.r.t. η, no variables are conflicting in $\text{SUB}_q(\rho(q, a, x))$ w.r.t. η'. Secondly, we show that for each $x, y, x', y' \in X'$, if $\eta'(x, y)$ holds, x appears in $\rho'(q, x')$, and y appears in $\rho'(q, y')$, then $\eta(x', y')$ holds. From the definition of η', we have that two variables in X' can conflict for one of the following 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 $\eta'(w_{q_2}, z_{q'_2})$. We now have two possibilities:
  — if $q_2 = q'_2$, assuming the current symbol is internal, we have that w_{q_2} and z_{q_2} are updated to $\text{SUB}_{q_2}(\rho_i(f(q_2), a, w))$ and $\text{SUB}_{q_2}(\rho_i(f(q_2), a, z))$, where all variables are labeled with q_2. This violates the assumption that $q_1 \neq q'_1$. If the current symbol is a call or a return a similar reasoning holds.
  — if $q_2 \neq q'_2$, $\eta'(w_{q_2}, z_{q'_2})$ follows from the first rule of η'.
— two variables $x_{q_1}, y_{q_1} \in X'$ such that $\eta(x, y)$ appear in two different assignments to w_{q_2} and $z_{q'_2}$ respectively, for some $w, z \in X$, and $q_2, q'_2 \in Q$. We need to show $\eta'(w_{q_2}, z_{q'_2})$. If $q_2 = q'_2$, $\eta'(w_{q_2}, z_{q'_2})$ follows from the second rule of η', while if $q_2 \neq q'_2$, $\eta'(w_{q_2}, z_{q'_2})$ follows from the first rule of η'.

□

3.4. 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 presence of regular-look-ahead, the conflict relation can be trivial, and thus, copyless STTs suffice.

Definition of Regular Look-ahead. Given a nested word $w = a_1 a_2 \dots a_n$, for each position $1 \leq i \leq n$, let $\text{WMP}(w, i)$ be the longest well-matched subword $a_i \dots a_j$ starting at position i. Formally given a well-matched nested word $w = a_1 a_2 \dots a_n$, $\text{WMP}(w, n+1) = \varepsilon$, and for each position i such that $1 \leq i \leq n$:

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

Given a symbol $a \in \Sigma$, we define the reverse of a tagged symbol as $\text{REV}(a) = a$, $\text{REV}(\langle a) = a\rangle$, and $\text{REV}(a\rangle) = \langle a$. We use $\text{REV}(w) = \text{REV}(a_n) \dots \text{REV}(a_1)$, for the reverse of w, 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 [Alur and Madhusudan 2009]). Then, while processing a nested word, testing whether the nested word $\text{WMP}(w, i)$ belongs to L corresponds to testing whether the state of A after processing $\text{REV}(\text{WMP}(w, i))$ is an accepting state of A. Since regular languages of nested words are closed under intersection, the state of a single bottom-up NWA A reading the input nested word in reverse, can be used to test membership of the well-matched suffix at each step in different languages. Also note that since A is bottom-up, its state after reading $\text{REV}(\text{WMP}(w, i))$, is the same as its state after reading $\text{REV}(a_i \dots a_n)$. This motivates the following formalization. Let $w = a_1 \dots a_n$ be a nested word over Σ, and let A be a bottom-up NWA with states R processing nested words over Σ. Given a state $r \in R$, we define the (r, A)-look-ahead labeling of w to be the nested word $w_r = r_1 r_2 \dots r_n$ over the alphabet R such that for each position $1 \leq j \leq n$, the call/return/internal type of r_j is the same as the type of a_j, and the corresponding symbol is the state of the NWA A after reading $\text{REV}(a_j \dots a_n)$ starting in state r. Then, the *A-look-ahead labeling of w*, is the nested word $w_A = w_{r_0}$. An *STT-with-regular-look-ahead* (STTR) consists of a bottom-up NWA A over Σ with states R, and an STT S from R to Γ. Such a transducer defines a streaming tree transduction from Σ to Γ: for an input nested word $w \in W(\Sigma)$, the output $[\![S, A]\!](w)$ is defined to be $[\![S]\!](w_A)$.

Closure under Regular Look-Ahead. The critical closure property for STTs is captured by the next theorem which states that regular look-ahead does not add to the expressiveness of STTs. This closure property is key to establishing that STTs can compute all MSO-definable transductions.

THEOREM 3.8 (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 δ''. Let S_A be an STT from R to Γ. We construct an STT $S' = (Q', q'_0, P', X', \eta', F', \delta', \rho')$ equivalent to the STTR (T, A).

Using Theorem 3.7, 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 3.7: 1) $\text{WMS}(w, i)$ is the longest well-matched subword $a_j \dots a_i$ ending at position i; 2) $\text{SUB}_{q,Y}(\alpha)$ is the function that substitutes each variables $x \in Y$ in an assignment α with the variable x_q.

First of all, we observe that for a well-matched nested word w, and an STT S, if $\delta^*((q, \Lambda, \alpha), w) = (q', \Lambda', \alpha')$, then $\Lambda = \Lambda'$, and the value Λ does not influence the execution of S. Hence, for a well-matched nested word w, we can omit the stack component from configurations, and write $\delta^*((q, \alpha), w) = (q', \alpha')$.

State Components and Invariants. Given the input nested word $w = a_1 \dots a_n$, when processing the symbol a_i, the transition of the STT S depends on the state of A after reading $\text{REV}(\text{WMP}(w, i))$. Since the STT S' cannot determine this value based on the prefix read so far, it needs to simulate S for every possible choice of $r \in R$. We do this by keeping some extra information in the states of S'.

Each state $q' \in Q'$ is a pair (f, g), where $f : R \mapsto R$, and $g : R \mapsto Q$. After reading the input symbol a_i, for every state $r \in R$, $f(r)$ is the state reached by A after reading $\text{REV}(\text{WMS}(w, i))$ when starting in state r, and $g(r)$ is the state reached by S after reading $\text{WMS}(w, i)_r$ starting in state q_0. Recall that w_r is the state labeling produced by A, when reading w starting in state r. The initial state is $q'_0 = (f_0, g_0)$, where, for every $r \in R$, $f_0(r) = r$ and, $g_0(r) = q_0$. Each stack state $p' \in P'$ is a triplet (f, g, a), where the components f and g are the same as for Q', and a is a symbol in Σ.

We now describe the transition relation δ'. We assume S' to be in state $q = (f, g)$, and processing the input symbol a_i. We have the following 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', and
— if $g(r') = q$, and $\delta_i(q, r') = q'$, then $g'(r) = q'$; the state reached by S when reading $\text{WMS}(w, i)_r$, is the same as the state it reaches when reading $\text{WMS}(w, i-1)_{r'} r'$;

$a_i = \langle a$ is a call: $\delta'_c(q, a) = (q'_0, p)$, where $p = (f, g, a)$; the current state (f, g) is stored on the stack along with the current symbol a, and the control state is reset to q'_0; and

$a_i = a\rangle$ is a return: let $a_j = \langle b$ be the matching call of a_i, and let $p = (f', g', b)$ be the state popped from the stack. Since A is bottom-up for every $r \in R$, $\delta''_c(r, a) = (r_0, p_r)$ for some $p_r \in P''$. Let $f(r_0) = r_1$ be the state reached by A when reading $\text{REV}(\text{WMS}(w, i-1))$ (i.e. the reversed subword sandwiched between the matching call and return) starting in state r_0, and $g(r_0) = q_1$ is the state reached by S after processing $\text{WMS}(w, i-1)_{r_0}$. Finally, $\delta'_r(q, p, a) = (f'', g'')$, where for each $r \in R$, if $\delta''_r(r_1, p_r, b) = r_2$,
— if $f'(r_2) = r'$ is the state reached by A after processing $\text{REV}(\text{WMS}(w, j-1))$ starting in state r_2, then $f''(r) = r'$, and
— if $g'(r_2) = q_2$ is the state reached by S after processing $\text{WMS}(w, j-1)_{r_2}$, $\delta'_c(q_2, r_2) = (q_0, p')$, and $\delta'_r(q_1, p', r_0) = q'$. then $g''(r) = q'$.

Variable Updates and Invariants. The STT S' has variable set $X' = \{x_r \mid x \in X, r \in R\}$. After processing the i-th symbol a_i, the value of x_r is the same as the value of x computed by S after reading $\text{WMS}(w, i)_r$. We can now describe the variable update function ρ'. We assume that S' is in state $q = (f, g)$, and it is processing the symbol a_i. For each variable $x_r \in X'$, S', S' performs the following update:

a_i is internal: if $\delta''_i(r, a) = r'$, and $g(r') = q'$, then $\rho'(q, a, x_r) = \text{SUB}_{r'}(\rho_i(q', r', x))$ is the assignment of S to x where each variable x is replaced with $x_{r'}$;

$a_i = \langle a$ is a call: we perform 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; and

$a_i = a\rangle$ is a return: let $a_j = \langle b$ be the corresponding call, and let $p = (f', g', b)$ be the state popped from the stack. The update follows a similar reasoning to that of the transition function δ'. Assume $\delta''_c(r, a) = (r_0, p)$, $f(r_0) = r_1$, $\delta''_r(r_1, p, b) = r_2$, $g(r_0) = q_1$, $g'(r_2) = q_2$, $\delta_c(q_2, r_2) = (q_3, p')$. For each $x \in X$, let $t_c(x)$ be the expression $\rho_c(q_2, r_2, x)$, and $t_r(x)$ be the expression $\rho_r(q_1, r_0, x)$;
now for every $x \in X$, let $t'_c(x) = t_c(x)\{y/y_p\}$ be the expression $t_c(x)$ in which every variable $y \in X$ is replaced with the corresponding stack variable y_p, and let $t''(x) = t_r(x)\{y_p/t'_c(y)\}$ be the expression $t_r(x)$ in which every variable $y_p \in X_p$ is replaced with the expression $t'_c(y)$. The final update will be the expression $\rho'(q, a, x_r) = \text{SUB}_{r_2, X_p}(\text{SUB}_{r', X}(t''(x)))$ where each non stack variable x is replaced with x'_r, and each stack variable y is replaced with y_{r_2}.

Output Function. At the end of the run F' only outputs variables labeled with r_0: for every state $(f, g) \in Q'$,

Conflict Relation. if $g(r_0) = q$, then $F'(f, g) = \text{SUB}_{r_0} F(q)$. Finally, we define the conflict relation η' as follows:

(1) Variable summarizing different lookahead states are in conflict: for all $x, y \in X$, for all $r_1 \neq r_2 \in R$, then $\eta'(x_{r_1}, y_{r_2})$; and
(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 $r \in R$, then $\eta'(x_r, y_r)$.

The proof that ρ' is consistent with η' is analogous to the proof of Theorem 3.7. □

Copyless STTs with RLA. Recall that an STT is said to be copyless if η only contains 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. In presence of regular-look-ahead test, the

STT can check which variable contribute to the final output, avoid redundant updates, and can thus be copyless.

THEOREM 3.9 (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 3.8 to construct an equivalent STT.

We now prove the $\Rightarrow$ direction. Let S_1 be an STT from Σ to Γ. Using theorem 3.7, let $S = (Q, q_0, P, X, \eta, F, \delta, \rho)$ be a bottom-up STT, equivalent to S_1. We construct a bottom-up NWA $A = (R, r_0, P'', \delta'')$, and a copyless STT S' from R to Γ, such that $[\![S', A]\!]$ is equivalent to $[\![S]\!]$.

The NWA A keeps in the state information about which variables will contribute to the final output. The STT S' uses such information to update only the variables the 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 $a_i \dots a_j$ starting at position i. Given a nested word $w = a_1 \dots a_n$, we also define $\text{NR}(w, i)$ to be the position of the first unmatched return in $a_i \dots a_n$. 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, $\text{NR}(w, i) = \text{NR}(w, j+1)$; and
(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: $U_X \stackrel{\text{def}}{=} \{Y \mid Y \subseteq X \wedge \forall x, y \in Y, (x, y) \notin \eta\}$. We then enrich it with a special variable x_F that represents the final output: $U_X^F \stackrel{\text{def}}{=} U_X \cup x_F$. Moreover, given an expression $s \in E(X, \Sigma)$ (i.e. an assignment's right hand side), we use $x \in_a s$ to say that a variable $x \in X$ appears in s.

Given a nested word $w = a_1 \dots a_n$, and an STT S, we define the function $\text{CV}_{S,w} : Q \times \{0, \dots, n\} \times U_X^F \mapsto U_X^F$, such that, for every state $q \in Q$, position i, and set of variables Y, $\text{CV}_{S,w}(q, i, Y) = Y'$ 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 $0 \leq i \leq n-1$,

a_{i+1} is internal: if $\delta_i(q, a_{i+1}) = q'$, and $\text{CV}_{S,w}(q', i+1, Y_1) = Y_2$, then $\text{CV}_{S,w}(q, i, Y_1) = Y_3$ where
- if $Y_2 = x_F$, then $Y_3 = \{x \mid x \in_a F(q')\}$;
- if $Y_2 \neq x_F$ and a_{i+2} is a call $\langle a$, then $Y_3 = \{x \mid \exists y \in Y_2 . x \in_a \rho_c(q', a, y)\}$; Y_3 contains the variables that appear on the right-hand side of the variables Y_1 while reading the symbol a_{i+2};
- if $Y_2 \neq x_F$ and a_{i+2} is internal, then $Y_3 = \{x \mid \exists y \in Y_2 . x \in_a \rho_i(q', a_{i+2}, y)\}$; and
- if a_{i+2} is a return, then $Y_3 = Y_2$.

$a_{i+1} = \langle a$ is a call: let $a_{j+1} = b\rangle$ be the matching return; if $\delta_c(q, a) = (q_0, p)$, $\delta^*(q_0, (a_{i+2} \dots a_j)) = q_1$, $\delta_r(q_1, a_{j+1}, p) = q_2$, and $\text{CV}_{S,w}(q_2, j+1, Y_1) = Y_2$, then $\text{CV}_{S,w}(q, i, Y_1) = Y_3$ where
- if $Y_2 = x_F$, then $Y_3 = \{x \mid \exists y \in_a F(q_2) . x^p \in_a \rho_r(q_1, b, p, y)\}$; and
- if $Y_2 \neq x_F$, then $Y_3 = \{x \mid \exists y \in Y_3 \ \wedge x^p \in_a \rho_r(q_1, b, p, y)\}$.

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

Before continuing we prove that the above function is well-defined; i.e. for every possible input, $\text{CV}_{S,w}$ returns a set of non-conflicting variables.

LEMMA 3.10. *For every $i \in \{0, \dots, n-1\}$, $q \in Q$, $Y \in U_X^F$, if $\text{CV}_{S,w}(q, i, Y) = Y'$ then $Y' \in U_X^F$.*

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

We have 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 $a_{i+2} = \langle a$ is a call, $Y_3 = \{x \mid \exists y \in Y_3.x \in_a \rho_c(q', a, y))\}$. Let's assume by way of contradiction that there exist $x, y \in Y_2$, such that $\eta(x, y)$ holds. If this is the case there must exist either two variables $x', y' \in Y_3$ such that $x \in_a \rho_c(q', a_{i+2}, x')$ and $y \in_a \rho_c(q', a, y')$, or a variable $x' \in Y_2$ such that $x, y \in_a \rho_c(q', a, x')$. In both cases, using the definition of conflict relation, we can show that the hypothesis that $Y_2 \in U_X^F$ contains only conflict free variables is violated.
a_{i+1} is a call: similar to the previous case; and
a_{i+1} is a return: trivial.

□

RLA Automaton Construction. We now construct the NWA A that computes the function $\text{CV}_{S,w}$. The NWA A mimics the definition of $\text{CV}_{S,w}$ while reading the input nested word backward. At every call (return for the input) the state of A is reset to ensure that the A is bottom-up and the current value of $\text{CV}_{S,w}$ is stored on the stack. At the matching return (call for the input), the value popped from the stack is used to compute the new value of $\text{CV}_{S,w}$.

In order for the construction to work, we also need A to "remember", while reading backward, what is the set of variables that will be relevant at the next call. Given a nested word $w = a_1 \dots a_n$, and a position i, we define $\text{NC}(w, i)$ as the next call in $\text{WMP}(w, i)$. Formally, $\text{NC}(w, n+1) = \bot$, and, for each $1 \leq i \leq n$

— if a_i is internal, then $\text{NC}(w, i) = \text{NC}(w, i+1)$,
— if a_i is a call, then $\text{NC}(w, i) = i$, and
— if a_i is a return, then $\text{NC}(w, i) = \bot$.

Next, we define the set of states R of A. Each state $r \in R$, is a quadruple (s, f, g, h) where $s \in \Sigma \cup \{\bot\}$, $f : Q \times U_X^F \mapsto U_X^F$, $g : Q \mapsto Q$, and $h : Q \times U_X^F \mapsto (U_X^F \cup \bot)$, where f computes the function $\text{CV}_{S,w}(q, i, Y) = Y'$, g summarizes the execution of S on $\text{WMP}(w, i)$, and h, computes which variables will be necessary at the return matching the next call, and therefore which variables must be stored on the stack. We maintain the following invariants: given the input nested word $w = a_1 \dots a_n$, after processing $\text{REV}(a_i \dots a_n)$, A is in state (s, f, g, h), where:

(1) for all $Y \in U_X^F$, and $q \in Q$, if $\text{CV}_{S,w}(q, i, Y) = Y'$, then $f(q, Y) = Y'$,
(2) if a_i is a return, then $s = \bot$, otherwise $s = a_i$,
(3) if $q' = \delta^*(q, \text{WMP}(w, i))$, then $g(q) = q'$, and
(4) for all $Y \in U_X^F$, and $q \in Q$, $h(q, Y) = Y_1$, where
   — if $\text{NC}(w, i) = \bot$, then $Y_1 = \bot$,
   — if $\text{NC}(w, i) = i_c$, $a_{i_c} = \langle a$ has matching return $a_{i_r} = b\rangle$, $\delta^*(q, a_i \dots a_{i_c-1}) = q_1$, $\delta_c(q_1, a_{i_c}) = (q_0, p)$, $\delta^*(q_0, \text{WMP}(w, i_c + 1)) = q_2$, $\delta_r(q_2, p, a_{i_r}) = q_3$, and $\text{CV}_{S,w}(q_3, i_r, Y) = Y_2$, then
      — if $Y_2 = x_F$, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in_a F(q_3).\ x \in_a \rho_r(q_2, b, p, y)\}$;
      — if $Y_2 \neq x_F$ and $a_{i_r+1} = \langle c$ is a call, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in Y_2.\exists z \in_a \rho_c(q_3, c, y).x \in_a \rho_r(q_2, b, z)\}$;
      — if $Y_2 \neq x_F$ and a_{i_r+1} is internal, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in Y_2.\exists z \in_a \rho_i(q_3, a_{i_r+1}, y).x \in_a \rho_r(q_2, b, z)\}$; and
      — if $Y_2 \neq x_F$ and $a_{i_r+1} = c\rangle$ is a return, then $Y_1 = \{x \mid x \notin X^p \wedge \exists y \in_a Y_2.\ x \in_a \rho_r(q_2, a_{i_r}, p, y)\}$.

The initial state of A is $r_0 = (\bot, f_0, g_0, h_0)$, where $f_0(q, Y) = Y$, $g_0(q) = q$ and $h_0(q, Y) = \bot$, for every $q \in Q$, and $Y \in U_X^F$.

Next we define the transition relation δ'' of A, that preserves the invariants of above. For each $r = (s, f, g, h) \in R$, $a \in \Sigma$, $\delta''(r, a)$ is defined as follows:

a is internal: $\delta_i''(r, a) = (a, f_1, g_1, h_1)$ where for each $q \in Q$, $Y_1 \in U_X^F$, if $\delta_i(q, a) = q'$, then $g_1(q) = g(q')$, $h_1(q, Y_1) = h(q', Y_1)$, and $f_1(q, Y_1) = Y_3$ where
   — if $Y_2 = x_F$, then $Y_3 = \{x \mid x \in_a F(q')\}$;
   — if $Y_2 \neq x_F$ and $s = \langle c$ is a call, then $Y_3 = \{x \mid \exists y \in Y_2.\exists z \in_a \rho_c(q_3, c, y).x \in_a \rho_r(q_2, b, z))\}$;
   — if $Y_2 \neq x_F$ and s is internal, then $Y_3 = \{x \mid \exists y \in Y_2.x \in_a \rho_i(q', s, y)\}$;
   — if $s = \bot$, then $Y_3 = Y_2$;
a is a call $\langle b$ (return reading backward): let $r_1 = ((s_1, f_1, g_1, h_1), s)$ be the state popped from the stack, then, $\delta_r''(r, r_1, b) = (b, f_2, g_2, h_2)$, where for each $q \in Q$, $Y \in U_X^F$,

if $\delta_c(q,b) = (q_0,p)$, $g(q_0) = q_1$, $\delta_r(q_1,s,p) = q_2$, and $f_1(q_2,Y_1) = Y_2$, then,
$g_2(q) = g(q_2)$, $h_2(q,Y_1) = Y_3 \cap X$, and, $f_2(q,Y_1) = \{x \mid x_p \in Y_3\}$, where
— if $Y_2 = x_F$, then $Y_3 = \{x \mid \exists y \in_a F(q_2) \wedge x \in_a \rho_r(q_1,s,p,y)\}$;
— if $Y_2 \neq x_F$ and $s_1 = \langle c$ is a call, then $Y_3 = \{x \mid \exists y \in Y_2.\ \exists z \in_a \rho_c(q_1,s_1,y).\ x \in_a \rho_r(q_2,s,z)\}$;
— if $Y_2 \neq x_F$ and s_1 is internal, then $Y_3 = \{x \mid \exists y \in Y_2.\ \exists z \in_a \rho_c(q_1,s_1,y).\ x \in_a \rho_r(q_2,s,z)\}$;
— if $Y_2 \neq x_F$ and $s_1 = c\rangle$ is a return, then $Y_3 = \{x \mid \exists y \in Y_2.\ x \in_a \rho_r(q_1,s,p,y)\}$;
a is a return $b\rangle$ (call reading backward): $\delta''_c(r,b) = (r_0,(r,b))$.

STT Construction. We finally need to define the STT S' from R to Γ. When reading an input symbol in $(a,f,g,h) \in R$, S' uses the information stored in the function f to update only the variables that are relevant to the final output. The set of states of S' is $Q' = Q \times U_X^F$, with initial state $q'_0 = (q_0, x_F)$. When processing the symbol $r_i = (a,f,g,h)$, S' is in state (q,Y) iff S reaches the state q when processing $a_1 \ldots a_{i-1}$, starting in q_0, and the set of relevant variables at the end of $\text{WMP}(w,i)$ is Y. Similarly, the set of stack states is $P' = P \times U_X^F$. The set of variables is $X' = X$. The transition function δ' is defined as follows. For each state $(q,Y) \in Q'$, stack state $(p,Y') \in P'$, and symbol $r = (a,f,g,h) \in R$ we have the following possibilities:

a is internal: if $\delta_i(q,a) = q'$, then $\delta'_i((q,Y),r) = (q',Y)$;
a is a call: if $\delta_c(q,a) = (q',p)$, and $h(q,Y) = Y'$, then $\delta'_c((q,Y),r) = (q',(p,Y'))$; and
a is a return: if $\delta_r(q,p,a) = q'$, then $\delta'_r((q,Y),(p,Y'),r) = (q',Y')$.

Next, we define the variable update function ρ'. For each state $(q,Y) \in Q'$, stack state $(p,Y') \in P'$, symbol $r = (a,f,g,h) \in R$, variable $x \in X'$;

— if $x \in f(q,Y)$, then we have the following possibilities:
a is internal: $\rho'_i(q,r,x)$ is the same as $\rho_i(q,a,x)$;
a is a call: $\rho'_c(q,r,x)$ is the same as $\rho_c(q,a,x)$; and
a is a return: $\rho'_r(q,p,r,x)$ is the same as $\rho_r(q,p,a,x)$;
— if $x \notin f(q,Y)$, then we have the following possibilities:
a is internal: if x is a type-0 variable then $\rho'_i(q,r,x) = \varepsilon$, otherwise $\rho'_i(q,r,x) = ?$;
a is a call: if x is a type-0 variable then $\rho'_c(q,r,x) = \varepsilon$, otherwise $\rho'_c(q,r,x) = ?$; and
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 $\text{CV}_{S,w}$ we have that S' is copyless, and by construction $[\![S',A]\!]$ is equivalent to $[\![S]\!]$. □

3.5. Multi-parameter STTs

In our basic transducer model, the value of each variable can contain at most one hole. Now we generalize this definition to allow a value to contain multiple parameters. Such a definition can be useful in designing an expressive high-level language for transducers, and it is also used to simplify constructions in later proofs.

We begin by defining nested words with parameters. The set $H(\Sigma,\Pi)$ of *parameterized nested words* over the alphabet Σ using the parameters in Π, is defined by the grammar

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

For example, the nested word $\langle a\, \pi_1\, \langle b\rangle\, \pi_2\, a\rangle$ represents an incomplete tree with a-labeled root that has a b-labeled leaf as a child, such that trees can be added to its left as well as right by substituting the parameter symbols π_1 and π_2 with nested words. We can view such a nested word with 2 parameters as a function of arity 2 that takes two well-matched nested words as inputs and returns a well-matched nested word.

In the generalized transducer model, the variables range over parametrized nested words over the output alphabet. Given an alphabet Σ, a set X of variables, and a set Π of parameters, the set $E(\Sigma,X,\Pi)$ of expressions is defined by the grammar

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

A valuation α from X to $H(\Sigma,\Pi)$ naturally extends to a function from the expressions $E(\Sigma,X,\Pi)$ to $H(\Sigma,\Pi)$.

To stay within the class of regular transductions, we need to ensure that each variable is used only once in the final output and each parameter appears only once in the right-hand side at each step. To understand how we enforce single-use-restriction on parameters, consider the update $x := xy$ associated with a transition from state q to state q'. To conclude that each parameter can appear at most once in the value of x after the update, we must know that the sets of parameters occurring in the values of x and y before the update are disjoint. To be able to make such an inference statically, we associate, with each state of the transducer,

an occurrence-type that limits, for each variable x, the subset of parameters that are allowed to appear in the valuation for x in that state. Formally, given parameters Π and variables X, an *occurrence-type* φ is a function from X to 2^{Π}. A valuation α from X to $H(\Sigma, \Pi)$ is said to be *consistent* with the occurrence-type φ if for every parameter $\pi \in \Pi$ and variable $x \in X$, if $\pi \in \varphi(x)$, then the parameterized nested word $\alpha(x)$ contains exactly one occurrence of the parameter π, and if $\pi \notin \varphi(x)$, then π does not occur in $\alpha(x)$. An occurrence-type from X to Π naturally extends to expressions in $E(\Sigma, X, \Pi)$: for example, for the expression e_1e_2, if the parameter-sets $\varphi(e_1)$ and $\varphi(e_2)$ are disjoint, then $\varphi(e_1e_2) = \varphi(e_1) \cup \varphi(e_2)$, else the expression e_1e_2 is not consistent with the occurrence-type φ. An occurrence-type φ' from variables X to Π is said to be *type-consistent* with an occurrence-type φ from Y to Π and an assignment ρ from Y to X, if for every variable x in X, the expression $\rho(x)$ is consistent with the occurrence-type φ and $\varphi(\rho(x)) = \varphi'(x)$. Type-consistency ensures that for every valuation α from Y to $H(\Sigma, \Pi)$ consistent with φ, the updated valuation $\alpha \cdot \rho$ from X to $H(\Sigma, \Pi)$ is guaranteed to be consistent with φ'.

Now we can define the transducer model that uses multiple parameters. A *multi-parameter STT* S from input alphabet Σ to output alphabet Γ consists of:

— a finite set of states Q,
— an initial state q_0,
— a set of stack symbols P,
— state-transition functions δ_i, δ_c, and δ_r, that are defined in the same way as for STTs;
— a finite set of typed variables X equipped with a reflexive symmetric binary conflict relation η;
— for each state q, an occurrence-type $\varphi(q) : X \mapsto 2^{\Pi}$, and for each stack symbol p, an occurrence-type $\varphi(p) : X \mapsto 2^{\Pi}$,
— a partial output function $F : Q \mapsto E(X, \Gamma, \Pi)$ such that for each state q, the expression $F(q)$ is consistent with η, and $\varphi(q)(F(q))$ is the empty set,
— for each state q and input symbol a, the update function $\rho_i(q, a)$ from variables X to X over Γ is consistent with η and it is such that the occurrence-type $\varphi(\delta_i(q, a))$ is type-consistent with the occurrence-type $\varphi(q)$ and the update $\rho_i(q, a)$,
— for each state q and input symbol a, the update function $\rho_c(q, a)$ from variables X to X over Γ is consistent with η and it is such that, if $\delta_c(q, a) = (q', p)$ the occurrence-types $\varphi(p)$ and $\varphi(q')$ are type-consistent with the occurrence-type $\varphi(q)$ and the update $\rho_c(q, a)$,
— for each state q and input symbol a and stack symbol p, the update function $\rho_r(q, p, a)$ from variables $X \cup X_p$ to X over Γ is consistent with η and it is such that the occurrence-type $\varphi(\delta_r(q, p, a))$ is type-consistent with the occurrence-type $\varphi(q)$ and $\varphi(p)$ and the update $\rho_r(q, p, a)$.

We can assume that $\varphi(q_0) = \emptyset$, and therefore all variables are initialized to ε.

Configurations of a multi-parameter STT are of the form (q, Λ, α), where $q \in Q$ is a state, α is a valuation from variables X to $H(\Gamma, \Pi)$ that is consistent with the occurrence-type $\varphi(q)$, and Λ is a sequence of pairs (p, β) such that $p \in P$ is a stack symbol and β is a valuation from variables X to $H(\Gamma, \Pi)$ that is consistent with the occurrence-type $\varphi(p)$. The clauses defining internal, call, and return transitions are as in case of 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 STT. Then we add regular look-ahead and show, through the closure under such operation, that the property holds for general STT.

THEOREM 3.11 (COPYLESS MULTI-PARAMETER STTS). *A nested-word transduction is definable by copyless STT iff it is definable by a copyless multi-parameter STT.*

Proof. Given a copyless STT S constructing a multi-parameter copyless STT S' is trivial. The parameter set Π of S' is the singleton $\{?\}$. For every state q of S, there is a corresponding state q in S'. For every type-0 variable x and state q in S, $\varphi(q, x) = \emptyset$ while for every type-1 variable y, $\varphi(q, y) = \{?\}$.

We now prove the other direction of the iff. Let S be multi-parameter copyless STT with states Q, initial state q_0, stack symbols P, parameters Π with $|\Pi| = k$, variables X with $|X| = n$, occurrence-type φ,

output function F, state-transition functions δ_i, δ_c, and δ_r, and variable-update functions ρ_i, ρ_c, and ρ_r. We construct an equivalent copyless STT $S' = (Q', q'_0, P', X', F', \delta', \rho')$.

Variable Summarization Intuition. We need to simulate the multi-parameter variables using only variables with a single hole. We do this by using multiple variables to represent a single multi-parameter variable, and by maintaining in the state extra information on how to combine them.

The construction maintains a compact representation of every multi-parameter variable. To understand the construction, consider a variable x with value $\langle a\ \langle b\ \pi_1\ b\rangle\ \langle c\rangle\ \langle b\ \pi_2\ b\rangle\ a\rangle$. One possible way to represent x using multiple variables, each with only one parameter in its value, is the following: $x_1 = \langle a?a\rangle$, $x_2 = \langle b?b\rangle\langle c\rangle$, and $x_3 = \langle b?b\rangle$. Next, we need to maintain in the state some information regarding how to combine these three values to reconstruct x. For this purpose, we use a function of the form $f(x_1) = (x_2, x_3), f(x_2) = \pi_1, f(x_3) = \pi_2$, that tells us to replace the ? in x_1 with x_2x_3 and the holes in x_2 and x_3, with π_1 and π_2 respectively. The state will also maintain a function g that remembers the starting variable (the root of the tree): in this case $g(x) = x_1$ means that x_1 is the root of the symbolic tree representing the variable x.

We now formalize this idea. The set of variables X' contains $(2k-1)n$ variables of type-1 and n variables of type-0. When S' is in state q, for every variable $x \in X$,

— if $\varphi(x) \neq \emptyset$, the value of x is represented by $2|\varphi(x)| - 1$ type-1 variables in X', and
— if $\varphi(x) = \emptyset$, the value of x is represented by one type-0 variable in X'.

Since $\varphi(x) \leq k$, we can assume that for every variable $x \in X$, there are exactly $2k-1$ type-1 variables and one type-0 variable in S' corresponding to it. We denote this set by $V(x) = \{x_0, x_1, \ldots, x_{2k-1}\}$, where x_0 is the only type-0 variable. Therefore, the STT S' has the set of variables $X' = \bigcup_{x\in X} V(x)$.

State Components and Invariants. Each state of Q' is a triplet (q, g, f), where $q \in Q$ keeps track of the current state of S, $g : X \mapsto X'$ keeps track of the root of the symbolic tree representing each variable, and $f : X' \mapsto (X' \times X') \cup \Pi \cup \{\varepsilon\} \cup \bot$ maintains information on the symbolic tree representing each variable. Given a variable $x \in X'$, $f(x) = \bot$ means that x is not being used in any symbolic tree.

We now define the unfolding f^* of the function f that, given a variable in $x \in X'$ provides the value in $H(\Sigma, \Pi)$ corresponding to the symbolic tree rooted in x:

— if $f(x) = \varepsilon$, then $f^*(x) = x$,
— if $f(x) = \pi_i$, then $f^*(x) = x[\pi_i]$, and
— if $f(x) = (y, z)$, then $f^*(x) = x[f^*(y)f^*(z)]$.
— if $f(x) = \bot$, then $f^*(x) = \bot$,

Our construction maintains the following invariant: at every point in the computation, the value of $f^*(g(x))$ in S' is exactly the same as the value of x in S. We can assume that at the beginning every variable $x \in X'$ has value ε, and we represent it with $g(x) = x_0$, and $f(x_0) = \varepsilon$. For this base case the invariant holds.

Similarly to what we did for STT (Corollary 3.5), we observe that every assignment can be expressed as a sequence of elementary updates of the following form:

Constant assignment. $x := w$ where w is a constant (w is of the form $a, \pi, x, \langle a\pi b\rangle$);
Concatenation. $\{x := xy; y := \varepsilon\}$ (and similar cases such as $\{x := yx; y := \varepsilon\}$); and
Parameter substitution:. $\{x := x[\pi \mapsto y]; y := \varepsilon\}$ (and similar cases such as $\{x := y[\pi \mapsto x]; y := \varepsilon\}$).
Swap:. $\{x := y; y := x\}$.

Update Functions. We now describe at the same time the transition relation δ' and the variable update function ρ' of S'. Consider a state (q, f, g). We call (q', f', g') the target state and we only write the parts that are updated, and skip 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:

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

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

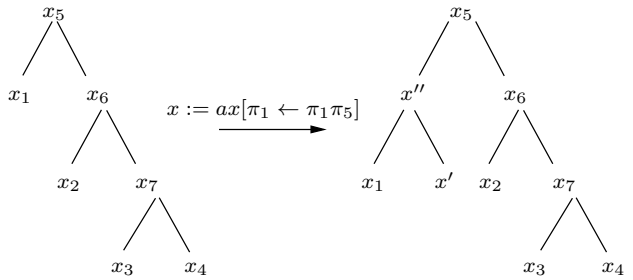

Fig. 2. Parameter tree for the variable $x = \pi_1\pi_2\pi_3\pi_4$. In this case (on the left) $g(x) = x_5$ and $f(x_5) = (x_1, x_6), f(x_6) = (x_2, x_7), f(x_7) = (x_3, x_4), f(x_1) = \pi_1, f(x_2) = \pi_2, f(x_3) = \pi_3, f(x_4) = \pi_4$. Each variable is of type-1. After the update we have that $x_5 := ax_5$ 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 $\pi_1\pi_2\pi_3\pi_4$ and after the update $f^*(x_5)$ evaluates to $a\pi_1\pi_5\pi_2\pi_3\pi_4$.

x_j to concatenate the variables summarizing x and y. We can reflect such an update in the tree shape of x as follows: for every $z \in V(x)$,

— if there exists y' such that $a(y') = z$, we copy the summary from y' to z and replace each variable in the summary with the corresponding one in $V(x)$: $f(z) = f(y)\{y'/a(y')\}$, $z = y$, and $y' = \varepsilon$;

— if $z = x_j$ we concatenate the previous summaries of x and y: if $g(x) = x'$, and $g(y) = y'$, and $a(y') = x''$, then $g'(x) = z$, and $f'(z) = x'x''$. Finally the variable z needs to hold the ?: $\rho(z) =?$.

Finally if y_0 is the type-0 variable in $V(y)$, $g'(y) = y'$, $f(y') = \varepsilon$, and $y_0 = \varepsilon$.

$\{x := x[\pi \mapsto y]; y := \varepsilon\}$: in order for the assignment to be well-defined we have that $\varphi(q', x)$ must be the same as $(\varphi(q, x) \setminus \{\pi\}) \cup \varphi(q, y)$, and $|\varphi(q', x)| = (|\varphi(q, x)| - 1) + |\varphi(q, y)| \leq k$. Let's assume wlog that both $\varphi(q, x)$ and $\varphi(q, y)$ are not empty. By IH x and y use $2|\varphi(x)| - 1 + 2|\varphi(y)| - 1 = 2(|\varphi(q, x)| + |\varphi(q, y)|) - 2 \leq 2(k + 1) - 2 = 2k$ variables. First we assign each $y_i \neq g(y)$ in the tree rooted in $g(y)$ to some unused $x_{i'}$ in $V(x)$. Let $a(y_i) = x_{i'}$ be such a mapping. So far we used $2k - 1$ variables in $V(x)$. When performing the updates we show how the variable representing the root $g(y)$ need not be copied allowing us to use at most $2k - 1$ variables to summarize the value of x. The root of the tree summarizing x will be the same as before: if $g'(x) = g(x)$. Every variable $z \in V(x)$ is updated as follows,

— if there exists y such that $a(y) = z$, we copy the summary from y to z and replace each variable in the summary with the corresponding one in $V(x)$: $f(z) = f(y)\{y'/a(y')\}$, $z = y$, and $y' = \varepsilon$;

— if $z = x_\pi$ we append the summary of y to it: if $g(y) = y'$, then

  — if $f(y') = y_1y_2$, $a(y') = x'$, $a(y_1) = x_1$ and $a(y_2) = x_2$, then $f'(z) = (x_1, x_2)$, and $\rho(z) = z[y']$; and

  — if $f(y') = \pi'$, and $a(y') = x'$, then $f'(z) = \pi'$, and $\rho(z) = z[y']$.

Finally if y_0 is the type-0 variable in $V(y)$, $g'(y) = y'$, $f(y') = \varepsilon$, and $y_0 = \varepsilon$.

$\{x := y; y := x\}$: we simply swap the summaries of x and y. Let $a : V(x) \mapsto V(y)$ be a bijection from $V(x)$ to $V(y)$, and let $b : V(y) \mapsto V(x)$ be the inverse of a. Then $g'(x) = a(g(y))$, $g'(y) = b(g(x)$, $f'(x) = f(y)\{y'/a(y')\}$, $f'(y) = f(x)\{x'/b(x')\}$, for each $x' \in V(x)$, $x' = a(x')$, and for each $y' \in V(y)$, $y' = b(y')$.

Figure 2 shows an example of update involving a combination of elementary updates. We still have to show how δ' and ρ' are defined at calls and returns. The functions maintained in the state are stored on the stack at every call, and such 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 Theorem 3.11 with regular look-ahead and get the following result.

COROLLARY 3.12 (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 3.9 to multi-parameter STTs.

LEMMA 3.13. *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 3.9 does not use parameters 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 3.14 (MULTI-PARAMETER STTS). *A nested-word transduction is definable by an STT iff it is definable by a multi-parameter STT.*

Proof. From Theorems 3.13, 3.12, and 3.14. □

3.6. 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 3.15 (COMPOSITION CLOSURE). *Given two STT-definable transductions, f_1 from Σ_1 to Σ_2 and f_2 from Σ_2 to Σ_3, the composite transduction $f_2 \cdot f_1$ from Σ_1 to Σ_3 is STT-definable.*

Proof. Using Theorem 3.9, we consider S_1 and S_2 to be copyless STTs with regular look-ahead. We are given two copyless STT:

— $S_1 = (Q_1, q_{01}, P_1, X_1, F_1, \delta_1, \rho_1)$ 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 3.8 and 3.14, to remove the parameters and then regular look-ahead, proving that there exists an STT equivalent to S.

Intuition Behind the Construction. The main idea is that we want to simulate all the possible executions of S_2 on the output of S_1 in a single execution. The composed STT 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 variables $x_1 \in X_1$ and $x_2 \in X_2$, the STT S has to remember what would be the value of x_2, if S_2 would read 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 lookaheads consist of only one state, and therefore we ignore them. Let's say that at some point in the computation x has value ? and the next input symbol is a. When reading a, S_1 updates x to $ax[?b]$. We need to reflect this update on the variable y of S_2 — i.e. what is the value of y when S_2 reads $ax[?b]$. However, we do not know what is the current state of S_2, and what value S_1 stores in the hole ?. For every possible state q of S_2, and variable x of S_1, the STT S tracks what is the state reached by S_2 after processing the value in x, starting in state q. However, we still need to deal with the unknown value of the hole ?. We can extend the previous idea to solve this problem. Consider the value of x to be $a?b$, where a and b are the nested words respectively before and after the hole. The STT S maintains a function f that, for every two states q_1 and q_2 of S_2, keeps track of the state reached by S_2 when reading a starting in state q_1, and the state reached by S_2 when reading b starting in state q_2 knowing that a was read starting in state q_1. In order to compute the second part of the function, S needs the stack computed by the first one and therefore needs to know that a is processed starting in state q_1.

Next, we describe how we summarize the variable updates of S_2. Again, the update of y depends on the state in which S_2 starts reading the value of x. Similarly to before, we need to deal with the unknown value of the hole ?. However this is not the only problem. Let's assume the variable update function of S_2 is as follows: $\rho_2(q, y, b) = cy$. We want to simulate the execution of S_2, but, at this point we do not know what is the previous value of y! We address this issue by treating the old value of y as a parameter. This tells us

that the set of parameters contains a parameter x' for every variable in $x \in X_2$. Similarly to what we did for the transition relation, for every two states q_1 and q_2 of S_2, and every variable y of S_1, there is

— a variable $g_1^L(q_1, x, y)$ representing the value of y, when S_2 reads the value of x on the left of ?, starting in state q_1; and
— a variable $g_1^R(q_1, q_2, x, y)$ representing the value of y, when S_2 reads the value of x on the right of ?, starting in state q_2, assuming that the value of y on the left of ? was read by S_2 starting in state q_1.

Both these variables at the beginning are be set to y', a parameter that represents the value of y before processing the current input. The updates then mimic the transition relation. For example, for the case in which $\rho_2(q, y, b) = cy$, the value of $g_1^R(q', q, x, y)$ is set to cy'.

Since S_2 itself uses type-1 variables, we use the parameter ? for such variable values. Let's analyze this case in detail. When ? directly appears in the g representation of a variable, we can treat it as a normal parameter. The problem occurs in the following case: let's say at a particular step $g(q, x, y) = y'$ but y is a type-1 variable. This can only mean that the ? appears in y'. Now let's assume that the next update is of the form $y := y[a]$. As we can see, we still do not have the ? appearing in the representation of y. We record this fact with a function and delay the substitution using an extra variable for the parameters. As an example, suppose that at some point the values of x and y, both of type 1, are x', y'. We use the variables $x_? = ?$ and $y_? = ?$ to represent their parameters. Then, after processing a well-matched subword, we may have an update of this form $x := ax[cy[a?c]]b$ and $y := a?$. Notice that the reflexivity of η ensures that x' and y' can appear at most once in the valuation of a variable at any point. This configuration is captured by (there is one such variable for every state, in this case let's assume is q) $x := axb$, $x_? = cy$, $y_? = a?c$ and $y = a?$. In addition we need to keep information about where the actual parameter of every variable is. Let's consider the case in which we are trying to summarize a type-0 variable y of S_2. We keep in the state a function p_0, such that $p_0(q, x, y) = \varepsilon$ if the ? appears in $g_0(q, x, y)$, while $p_0(q, x, y) = xy$, if, for example, in order to substitute the value of ? with the value v ($x := x[v]$), we need to perform the following update, where we omit the state q and the variable x for readability:

$$x := x[x' \mapsto x_?[y' \mapsto y_?[? \mapsto v]]]$$

Finally, we need to summarize the possible lookahead values of S_2 when reading the variable contents. For example if a variable x of S_1 contains the value $s?t$, we need to know, for every r_1 and r_2 in R, what state would A_2 reach when reading s and t backward. We use two functions l_1^L and l_1^R, such that $l_1^R(r_2, x) = r_2'$ and $l_1^L(r_1, r_2, x) = r_1'$, iff $\delta_R^*(r_2, \text{REV}(t)) = (r_2', \Lambda)$, and $\delta_R^*(r_1, \Lambda, \text{REV}(s)) = r_1'$.

State Components and Invariants. We denote with $X_{i,j}$ the set of type-j variables in X_i. Each state of Q is a tuple $(q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$, where

— $q \in Q_1$ keeps track of the current state of S_1,
— $f_0 : Q_2 \times R \times X_{1,0} \mapsto Q_2$ is the summarization function for type-0 variable,
— $f_1^L : Q_2 \times R \times R \times X_{1,1} \mapsto Q_2$, and $f_1^R : Q_2 \times Q_2 \times R \times R \times X_{1,1} \mapsto Q_2$ are the summarization functions for type-1 variables,
— $l_0 : R \times X_{1,0} \mapsto R$ is the lookahead summarization function for type-0 variable,
— $l_1^L : R \times R \times X_{1,1} \mapsto R$, and $l_1^R : R \times X_{1,1} \mapsto R$ are the lookahead summarization functions for type-1 variables,
— $p_0 : Q_2 \times R \times X_{1,0} \times X_{2,1} \mapsto X_{2,1}^*$ is the function keeping track of the ? for type-0 variables, and
— $p_1^L : Q_2 \times R \times R \times X_{1,1} \times X_{2,1} \mapsto X_{2,1}^*$, and $p_1^R : Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_{2,1} \mapsto X_{2,1}^*$ are the function keeping track of the ? for type-1 variables.

We first describe the invariants that S maintains for the first 7 components: given an input nested word $w = a_1 \dots a_n$, after reading the symbol a_i, S is in state $(q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, \_, \_, \_)$, such that

— q is the state reached by S_1 on the prefix $a_1 \dots a_i$, $\delta_1^*(q_{01}, a_1 \dots a_i) = q$,
— given $q_1 \in Q_2$, and $r \in R$, if x contains the value $s \in W_0(\Sigma_2)$, and $\delta_2^*(q_1, s_r) = q_1'$, then $f_0(q_1, r, x) = q_1'$;
— given $q_1, q_2 \in Q_2$, and $r_1, r_2 \in Q_2$, if x contains the value $s?t \in W_1(\Sigma_2)$, $\delta_2^*(q_1, s_{r_1, \Lambda_{r_2,t}}) = (q_1', \Lambda)$, and $\delta_2^*(q_2, \Lambda, t_{r_2}) = q_2'$, then $f_1^L(q_1, r_1, r_2, x) = q_1'$, and $f_1^R(q_1, q_2, r_1, r_2, x) = q_2'$. We use the notation $s_{r_1, \Lambda_{r_2,t}}$ to denote the run on s of A_2 starting in state r_1 assuming that A_2 has an initial stack value computed by A_2 on t starting in state r_2.
— given $r_1 \in R$, if x contains the value $s \in W_0(\Sigma_2)$, and $\delta_2^*(r_1, \text{REV}(s)) = r_1'$, then $l_0(r_1, x) = r_1'$; and

— given $r_1, r_2 \in R$, if x contains the value $s?t \in W_1(\Sigma_2)$, $\delta_2^*(r_2, \text{REV}(t)) = (r_2', \Lambda)$, and $\delta_2^*(r_1, \Lambda, \text{REV}(s)) = r_1'$, then $l_1^L(r_1, r_2, x) = r_1'$, and $l_1^R(r_2, x) = r_2'$.

State Transition Function. We now show how we maintain the invariants defined above at every update. We first investigate the update of all the components different from p_0, p_1^L, and p_1^R. Let's assume S is in state $(q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$. We are only going to write the parts that are updated, and as before we only consider atomic updates of S_1. We analyze the type-1 case (the 0 case is easier). At every step we indicate with a prime sign the updated components.

$\{x := w\}$: we consider the case where w is a constant $\langle a?b\rangle$ (the other cases are similar). For each q_1 and q_2 in Q_2, r_1, r_2 in R, if $\delta_R(r_2, \langle b) = (r_2', p)$ for some $p \in P_r$, $\delta_R(r_1, p, a\rangle) = r_1'$, $\delta_2^*(q_1, r_1') = (q_1', \Lambda)$, and $\delta_2^*(q_2, \Lambda, r_2') = q_2'$, then the new state has the following components: ${f_1^L}'(q_1, r_1, r_2, x) = q_1'$, ${f_1^R}'(q_1, q_2, r_1, r_2, x) = q_2'$, ${l_1^L}'(r_1, r_2, x) = r_1'$, and ${l_1^R}'(r_2, x) = r_2'$.

$\{x := xy; y := \varepsilon\}$: without loss of generality, let y be a type-0 variable, and x be type-1 variables. For each q_l and q_r in Q_2, r_l, r_r in R, if $l_0(r_1, y) = r_1$, $l_1^R(r_1, x) = r_2$, $l_1^L(r_l, r_1, x) = r_3$, $f_1^L(q_l, r_l, r_1, x) = q_1$, $f_1^R(q_r, r_l, r_1, x) = q_2$, and $f_0(q_2, r_r, y) = q_3$, then for every $q \in Q_2$, and $r \in R$, the new state has the following components: ${l_1^L}'(r_l, r_r, x) = r_3$, ${l_1^R}'(r_r, x) = r_2$, ${l_0}'(r, y) = r$, ${f_1^L}'(q_l, r_l, r_r, x) = q_1$, ${f_1^R}'(q_l, q_r, r_l, r_r, x) = q_3$, and ${f_0}'(q, r, y) = q$.

$\{x := x[y]; y :=?\}$: we consider the case where x, y are type-1 variables (the other cases are simpler). We need to "synchronize" the left and right parts to update the function f. For each q_l and q_r in Q_2, r_l, r_r in R, assume $l_1^R(r_r, x) = r_1$, $l_1^R(r_1, y) = r_2$, $l_1^L(r_l, r_1, y) = r_3$, $l_1^L(r_3, r_r, z) = r_4$, and $f_1^L(q_l, r_3, r_r, x) = q_1$, $f_1^L(q_1, r_l, r_1, y) = q_2$, $f_1^R(q_1, q_r, r_l, r_1, y) = q_3$, and $f_1^R(q_l, q_3, r_3, r_r, x) = q_4$. For every $q, q' \in Q_2$, and $r, r' \in R$, the new state has the following components:
${l_1^L}'(r_l, r_r, x) = r_4$, ${l_1^R}'(r_r, x) = r_2$, ${l_1^L}'(r, r', y) = r$, ${l_1^R}'(r, y) = r$, ${f_1^L}'(q_l, r_l, r_r, x) = q_2$, ${f_1^R}'(q_l, q_r, r_l, r_r, x) = q_4$, ${f_1^L}'(q, r, r', y) = q$, and ${f_1^R}'(q, q', r, r', y) = q'$.

$\{x := y; y := x\}$: every component involving x is swapped with the corresponding component involving y.

Variable Summarization. Similarly to what we did for state summarization, S will have variables of the form $g_0(q, r, x, y)$ with the following meaning: if x contains a nested word w, $g_0(q, r, x, y)$ is the value contained in $y \in X_2$ after reading w_r starting in state q. As we described earlier, there will also be variables of the form $g_0(q, r, x, y)?$ representing the value of the parameter of y. The set of variables X of S described by the union of the following sets:

— $\{g_0(q, r, x, y) \mid (q, r, x, y) \in Q_2 \times R \times X_{1,0} \times X_2\}$,
— $\{g_0(q, r, x, y)? \mid (q, r, x, y) \in Q_2 \times R \times X_{1,0} \times X_{2,1}\}$,
— $\{g_1^L(q, r_1, r_2, x, y) \mid (q, r_1, r_2, x, y) \in Q_2 \times R \times R \times X_{1,1} \times X_2\}$,
— $\{g_1^L(q, r_1, r_2, x, y)? \mid (q, r_1, r_2, x, y) \in Q_2 \times R \times R \times X_{1,1} \times X_{2,1}\}$,
— $\{g_1^R(q, q', r_1, r_2, x, y) \mid (q, q', r_1, r_2, x, y) \in Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_2\}$, and
— $\{g_1^R(q, q', r_1, r_2, x, y)? \mid (q, q', r_1, r_2, x, y) \in Q_2 \times Q_2 \times R \times R \times X_{1,1} \times X_{2,1}\}$.

Given a nested word w, and a stack Λ, we use $w_{r,\Lambda}$ to denote the regular look-ahead labeling of w when processing $\text{REV}(w)$ in the starting configuration (r, Λ). For every $q_1 \in Q_2$, $r_1, r_2 \in R$, $x \in X_{1,1}$, and $y \in X_2$, if x contains a nested word $v?w$, $\delta_r^*(r_2, w) = (r_2', \Lambda_{r_2})$, and $\delta_2^*(q_1, v_{r_1,\Lambda_{r_2}}) = (q_2, \Lambda_{q_2})$, then

— $g_1^L(q_1, r_1, r_2, x, y)$ is the variable representing the value of y, after S_2 reads $v_{r_1,\Lambda_{r_2}}$ starting in state q_1; and
— $g_1^R(q_1, q_2, r_1, r_2, x, y)$ is the variable representing the value of y, after S_2 reads w_{r_2} starting in the configuration (q_2, Λ_{q_2}).

The parameters of S are used to represent the values of the variables in X_2 when starting reading the values of a variable X_1. At this point we do not know what the values of the variables in X_2 are and for every variable $x \in X_2$, we use a parameter x' to represent the value of x before reading the value in X_1. The STT S has set of parameters $\Pi = \{x' | x \in X_2\} \cup \{?\}$.

At any point in the execution the variable values and the state of S will be related by the following invariant: for any starting valuation α of the variables in X_2, state $q \in Q_2$, look-ahead state $r \in R$, variable

$y \in X_2$, and variable $x \in X_1$ with value w, the value of y after reading w_r with initial configuration α can be retrieved from the variables in X and the state components p_0, p_1^L, p_1^R.

Given a nested word $w = a_1 \dots a_n$, we consider the configuration of S and S_1 right after processing the symbol a_i. Let's call $(q_1, \Lambda_1, \alpha_1)$ the current configuration of S_1, and (q_S, Λ, α), with $q_S = (q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$, the current configuration of S. For every two states $q_2, q_2' \in Q_2$, lookahead 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 $y_1 \in X_{2,1}$:

x_0, y_0: let $\alpha_1(x_0) = s^{x_0}$ be the current valuation of x_0 in S_1, $\alpha(g_0(q_2, r, x_0, y_0)) = t$ be the current valuation of $g_0(q_2, r, x_0, y_0)$ in S, and $\{v_1', \dots, v_k'\} \subseteq \Pi$ be the set of parameters in $\varphi(g_0(q_2, r, x_0, y_0))$; for every valuation α_2 over X_2, if $\delta^*((q_2, \alpha_2), s_r^{x_0}) = (q_3, \alpha_2')$, then

$$\alpha_2'(y_0) = t[v_1' \mapsto \alpha_2(v_1)] \dots [v_k' \mapsto \alpha_2(v_k)]$$

x_0, y_1: let $\alpha_1(x_0) = s^{x_0}$ be the current valuation of x_0 in S_1, $\alpha(g_0(q_2, r, x_0, y_1)) = t$ be the current valuation of $g_0(q_2, r, x_0, y_1)$ in S, $p_0(q_2, r, x_0, y_1) = z_1 \dots z_i$ be the sequence of variables to follow to reach the hole ? in the representation of y_1, and $\{v_1', \dots, v_k'\} \subseteq \Pi$ be the set of parameters in

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

then for every valuation α_2 over X_2, if $\delta^*((q_2, \alpha_2), s_r^{x_0}) = (q_3, \alpha_2')$, then

$$\alpha_2'(y_1) = t[z_1' \mapsto [g_0(q_2, x_0, z_1)?[\dots[z_i' \mapsto g_0(q_2, x_0, z_i)?]]]][v_1' \mapsto \alpha_2(v_1)] \dots [v_k' \mapsto \alpha_2(v_k)]$$

x_1, y_0: let $\alpha_1(x_1) = s^{x_1^L} ? s^{x_1^R}$ be the current valuation of x_1 in S_1, $\alpha(g_1^L(q_2, r, r', x_1, y_0)) = t_L$ be the current valuation of $g_1^L(q_2, r, r', x_1, y_0)$ in S, $\alpha(g_1^R(q_2, q_2', r, r', x_1, y_0)) = t_R$ be the current valuation of $g_1^R(q_2, q_2', r, r', x_1, y_0)$ in S, let $\{v_1^{L'}, \dots, v_k^{L'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^L(q_2, r, r', x_1, y_0))$, and let $\{v_1^{R'}, \dots, v_l^{r'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^R(q_2, q_2', r, r', x_1, y_0))$; for every valuations α_2, α_2' over X_2, if $\delta^*((q_2, \alpha_2), s^{x_1^L}_{r, \Lambda_r', s^{x_1, R}}) = (q_3, \Lambda_2, \alpha_3)$, and $\delta^*((q_2', \Lambda_2, \alpha_2'), s_{r'}^{x_1^R}) = (q_3', \alpha_3')$, then $\alpha_3(y_0) = t_L[v_1^{L'} \mapsto \alpha_2(v_1^L)] \dots [v_k^{L'} \mapsto \alpha_2(v_k^L)]$, and $\alpha_3'(y_0) = t_R[v_1^{R'} \mapsto \alpha_2(v_1^R)] \dots [v_l^{R'} \mapsto \alpha_2(v_l^R)]$.

x_1, y_1: let $\alpha_1(x_1) = s^{x_1^L} ? s^{x_1^R}$ be the current valuation of x_1 in S_1, $\alpha(g_1^L(q_2, r, r', x_1, y_1)) = t_L$ be the current valuation of $g_1^L(q_2, r, r', x_1, y_1)$ in S, $\alpha(g_1^R(q_2, q_2', r, r', x_1, y_1)) = t_R$ be the current valuation of $g_1^R(q_2, q_2', r, r', x_1, y_1)$ in S, $p_1^L(q_2, r, r', x_1, y_1) = z_1^L \dots z_i^L$ be the sequence of variables to follow to reach the hole ? in the representation of y_1 on the right of the ?, $p_1^R(q_2, q_2', r, r', x_1, y_1) = z_1^R \dots z_j^R$ be the sequence of variables to follow to reach the hole ? in the representation of y_1 on the right of the ?, $\{v_1^{L'}, \dots, v_k^{L'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^L(q_2, r, r', x_1, y_1))$, and $\{v_1^{R'}, \dots, v_l^{R'}\} \subseteq \Pi$ be the set of parameters in $\varphi(g_1^R(q_2, q_2', r, r', x_1, y_1))$; for every valuations α_2, α_2' over X_2, if $\delta^*((q_2, \alpha_2), s^{x_1^L}_{r, \Lambda_r', s^{x_1^R}}) = (q_3, \Lambda_2, \alpha_3)$, and $\delta^*((q_2', \Lambda_2, \alpha_2'), s_{r'}^{x_1, R}) = (q_3', \alpha_3')$, then

$$\alpha_3(y_1) = t_L[z_1^{L'} \mapsto [g_1^L(q_2, r, r', x_1, z_1^L)?[\dots[z_i^{L'} \mapsto g_1^L(q_2, r, r', x_1, z_i^L)?]]]][v_1^{L'} \mapsto \alpha_2(v_1^L)] \dots [v_k^{L'} \mapsto \alpha_2(v_k^L)]$$

$$\alpha_3'(y_1) = t_R[z_1^{R'} \mapsto [g_1^R(q_2, q_2', r, r', x_1, z_1^R)?[\dots[z_i^{R'} \mapsto g_1^R(q_2, q_2', r, r', x_1, z_i^R)?]]]][v_1^{R'} \mapsto \alpha_2(v_1^R)] \dots [v_k^{R'} \mapsto \alpha_2(v_k^R)]$$

Variable Update Function. We assume that S is reading the symbol a, starting in state $q_S = (q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$. We only describe the components that are updated, and assume w.l.o.g that $x_0, x_1 \in X_{1,0}$ are type-0 variable, and $x_2 \in X_{2,1}$ is a type-1 variable. We assume the occurrence-type function $\varphi : Q \times X \mapsto 2^\Pi$ to be well defined according to the following assignments (we will prove consistency later).

$\{x_0 := w\}$: where w.l.o.g. w is a constant without a ?. We fix the variable components to be the state $q \in Q_2$, lookahead state $r \in R$, and we are summarizing the values of $y_0 \in X_{2,0}$ and $y_1, y_2 \in X_{2,1}$, when reading the value in $x_0 \in X_{1,0}$. We consider the variable updates performed by S_2 when reading the w, and assume $u = u_1 \dots u_n$ to be the sequence of atomic updates (using Theorem 3.4) performed by S_2 when reading w_r starting in state q. We now provide the updates of S corresponding to the updates in u. At the beginning of the procedure $g_0(q, r, x_0, y_1) = y_1'$, and $p_0(q, r, x_0, y_1) = y_1$.

We denote with a prime the new state values and we only write the parts that are updated. Let's assume the atomic update is of the following form:

$\{y_1 := w\}$: where $w = \langle a?b\rangle$ (the other cases are similar). We have $p_0'(q, r, x_0, y_1) = \varepsilon$, and we define the current assignment to be $g_0(q, r, x_0, y_1) := w$.

$\{y_0 := \varepsilon; y_1 := y_0y_1\}$: $p_0'(q, r, x_0, y_1) = p_0(q, r, x_0, y_1)$, and we define the current assignment to be $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_0)g_0(q, r, x_0, y_1)$, and $g_0(q, r, x_0, y_0) = \varepsilon$.

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

- if $k = 0$, then $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_1)[g_0(q, r, x_0, y_2)]$, and $g_0(q, r, x_0, y_2) =?$; and
- if $k > 0$, then $g_0(q, r, x_0, y_1) := g_0(q, r, x_0, y_1)$, $g_0(q, r, x_0, v_k?) := g_0(q, r, x_0, v_k?)[g_0(q, r, x_0, y_2)]$, and $g_0(q, r, x_0, y_2) =?$.

$\{y_1 := y_2; y_2 := y_1\}$: every component involving y_1 is swapped with the corresponding component involving y_2.

The final variable update is the result of composing all the atomic updates. As shown in Lemma 3.6, the composition of copyless updates is itself copyless.

$\{x_0 := x_0x_1; x_1 := \varepsilon\}$: we need to substitute the values of the variables after reading x_0 in the corresponding parameters in x_1 in order to simulate the concatenation. Informally (we omit the states for readability) if $g_0(x_1, y)$ contains the value az', and $g_0(x_0, z)$ contains the value bw', the new summarization $g_0'(x_0, y)$ will contain the value abw' where the parameter z' is being replaced with the corresponding variable values.

For every state $q_2 \in Q_2$, lookahead state $r \in R$, variable $y_1 \in X_{2,1}$, if $l_0(r, x_1) = r_1$, $f_0(q_2, r_1, x_0) = q_3$, $\varphi(q_S, g_0(q_3, r, x_1, y_1)) = \{v_1', \dots, v_k'\}$, and $p_0(q_2, r, x_1, y_1) = b_1 \dots b_{k'}$, then $p_0{}'(q_2, r, x_0, y_1) := p_0(q_2, r_1, x_0, b_1) \dots p_0(q_2, r_1, x_0, b_{k'})$, 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 $P = b_1 \dots b_{k'}$, and the variable $v = y_1$.

(1) let p_i be the first element in $P = p_1 \dots p_n$ such that $p_0{}'(q_2, r, x_0, p_i) \neq \varepsilon$

(2) perform the following assignment

$$g_0{}'(q_2, r, x_0, v) := g_0(q_3, r, x_1, v)[p_1' \mapsto g_0(q_3, r_1, x_0, p_1)[? \mapsto g_0(q_3, r_1, x_1, p_1)?[\dots p_{i-1}' \mapsto g_0(q_3, r_1, x_0, p_{i-1})[? \mapsto g_0(q_3, r_1, x_1, p_{i-1})?] \dots]]]$$

$$g_0{}'(q_2, r, x_0, p_j)? :=? \text{ for each } 1 \leq j < i$$

(3) $P := p_{i+1} \dots p_n$, 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 relations η over X, which is consistent with ρ. We'll often use the fact that, in assignments of the form $x := yz$ or $x := y[z]$, it is always the case that $y \neq z$. The conflict relation η is defined as follows: for all $q_1, q_1', q_2, q_2' \in Q_2,\ r_1, r_1', r_2, r_2' \in R,\ x \in X_{1,0},\ y \in X_{1,1},\ u, v \in X_2$,

— if $(q_1, r_1) \neq (q_1', r_1')$, then $\eta(g_0(q_1, r_1, x, u), g_0(q_1', r_1', x, v))$,
— if $(q_1, r_1) \neq (q_1', r_1')$, then $\eta(g_0(q_1, r_1, x, u_?), g_0(q_1', r_1', x, v_?))$,
— if $(q_1, r_1, r_2) \neq (q_1', r_1', r_2')$, then $\eta(g_1^L(q_1, r_1, r_2, y, u), g_1^L(q_1', r_1', r_2', y, v))$,
— if $(q_1, r_1, r_2) \neq (q_1', r_1', r_2')$, then $\eta(g_1^L(q_1, r_1, r_2, y, u_?), g_1^L(q_1', r_1', r_2', y, v_?))$,
— if $(q_1, q_2, r_1, r_2) \neq (q_1', q_2', r_1', r_2')$, then $\eta(g_1^R(q_1, q_2, r_1, r_2, y, u), g_1^R(q_1', q_2', r_1', r_2', y, v))$, and
— if $(q_1, q_2, r_1, r_2) \neq (q_1', q_2', r_1', r_2')$, then $\eta(g_1^R(q_1, q_2, r_1, r_2, y, u_?), g_1^R(q_1', q_2', r_1', r_2', y, v_?))$.

We now show that the variable update function ρ does not violate the conflict relation η. Inspecting the updates we perform it is easy to see that the same variable never appears twice on the right-hand side of the same variable. Now, by way of contradiction let's assume there exists an assignment which violates the constraints (we indicate in bold the meta-variables of S and in italic those of S_2). There are two possibilities:

(1) $\mathbf{x} \neq \mathbf{y}$, $\eta(\mathbf{x}, \mathbf{y})$, and both $\mathbf{x}$ and $\mathbf{y}$ occur on the right-hand side of some variable;
(2) $\eta(\mathbf{x}, \mathbf{y})$, and there exists two variables $\mathbf{x}'$ and $\mathbf{y}$, such that $\mathbf{x}' := fun(\mathbf{x})$, $\mathbf{y}' = fun(\mathbf{y})$ for which $\eta(\mathbf{x}', \mathbf{y}')$ doesn't hold.

Case (1) can be ruled out by simply inspecting all the possible assignments in the definition of ρ. The only interesting cases are $\{x_0 := x_0x_1; x_1 := \varepsilon\}$ and $\{x_0 := x_2[x_0]; x_2 :=?\}$ where the reasoning is the following. We first need to show that, for every $q_2 \in Q_2$, $r \in R$, $x_0 \in X_{1,0}$, if $X_{2,1} = \{y_1, \ldots, y_n\}$ is the set of type-1 variables in S_2, then the sequence $p_0(q_2, r, x_0, y_1) \ldots p_0(q_2, r, x_0, y_n)$ is repetition free. It is easy to show that this property holds by induction. After we have this, it is easy to show that the assignments do not violate the conflict relation.

We now need to deal with the conflict case (2). Before starting it is worth to point out that every variable $x \in X_1$ appears in at most one of the assignments of S_2 due to the copyless restriction. We want to show that it cannot happen that two variables that are in conflict are assigned to two different variables that are not in conflict. Let's try to analyze when two variables $\mathbf{x}, \mathbf{y}$ assigned to different variables can be in conflict. The first case is that of $\mathbf{x} = \mathbf{y}$. 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 η is well defined. For the case $x_0 := x_2[x_0]$ the argument is analogous.

Next, we show how the construction deals with calls and returns. As in the proof of Theorem 3.14, 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 3.14, assignments do not violate the single use restriction. Notice that the fact that the variables are reset at calls is crucial for this construction.

Output Function. Finally, we define the output function F. When reaching the last symbol, we need to construct the final output, but now at this point, we know what states to use. We illustrate the construction with an example, since the general one is very similar to the construction of ρ. Let's assume we are in state $q_S = (q, f_0, f_1^L, f_1^R, l_0, l_1^L, l_1^R, p_0, p_1^L, p_1^R)$, then $F(q_S)$ is defined as follows. We assume w.l.o.g. 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. □

3.7. Restricted Inputs

A nested word captures both linear and hierarchical structure. There are two natural classes 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.

Mapping Strings. Suppose we restrict the inputs to contain only internal symbols, that is, strings over Σ. Then the STT cannot use its stack, and we can assume that the set P of stack symbols is the empty set. This restricted transducer can still map strings to nested words (or trees) over Γ with interesting hierarchical

structure, and hence, is called a *string-to-tree* transducer. This leads to the following definition: a *streaming string-to-tree transducer* (SSTT) S from input alphabet Σ to output alphabet Γ consists of a finite set of states Q; an initial state $q_0 \in Q$; a finite set of typed variables X together with a conflict relation η; a partial output function $F : Q \mapsto E_0(X, \Gamma)$ such that for each state q, a variable x appears at most once in $F(q)$; a state-transition function $\delta : Q \times \Sigma \mapsto Q$; and a variable-update function $\rho : Q \times \Sigma \mapsto \mathcal{A}(X, X, \eta, \Gamma)$. Configurations of such a transducer are of the form (q, α), where $q \in Q$ is a state, and α is a type-consistent valuation for the variables X. The semantics $[\![S]\!]$ of such a transducer is a partial function from Σ^* to $W_0(\Gamma)$. We notice that in this setting the copyless restriction is enough to capture MSO completeness since the model is closed under regular look-ahead (i.e. a reflexive η is enough).

THEOREM 3.16 (COPYLESS SSTT'S CLOSURE 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 3.9 we consider the input to be a copyless SSTT with regular look-ahead. Given a DFA $A = (R, r_0, \delta_A)$ over the alphabet Σ, and a copyless string-to-tree STT $S = (Q, q_0, X, F, \delta, \rho)$ over R, we construct an equivalent copyless multi-parameter STT $S' = (Q', q'_0, X', \Pi, \varphi, F', \delta', \rho')$ over Σ.

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: 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 $0 \leq i \leq m$ such that $s_0 \ldots s_n = s'_i \ldots s'_{i+n}$, and for every $0 \leq j \leq n$, $t_j \in \mathcal{SF}(t'_{i+j})$, then $f \in \mathcal{SF}(f')$; and
— if there exist $i \leq m$ such that $f \in \mathcal{SF}(t'_i)$, then $f \in \mathcal{SF}(f')$.

Given two forests $f_1, f_2 \in \mathcal{F}(U)$, we write $\mathcal{S}(f_1, f_2) \equiv \mathcal{SF}(f_1) \cap \mathcal{SF}(f_2)$ for the set shared sub-forests of f_1 and f_2. Finally the set of maximal shared sub-forests is defined as

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

State Components and Invariants. The transition of the STT S at a given step depends on the state of A after reading the reverse of the suffix. Since the STT S' cannot determine this value based on the prefix, it needs to simulate S for every possible choice. We denote by X_i the type-i variables of X. Every state $q \in Q'$ is a tuple (l, f, g) where

— $l : R \mapsto R$, keeps track, for every possible state $r \in R$, of what would be the state of A after processing the string $\text{REV}(w)$, where w is the string read so far;
— $f : R \mapsto Q$, keeps track, for every possible state $r \in R$, of what would be the state of S after processing the string w_r, where w is the string read so far; and
— $g : (R \times X) \mapsto F(X' \cup \{?\})$ keeps track of how the variables X' of S' need to be combined in order to obtain the value of a variable of S,

State Summarization Invariants. We first discuss the invariants of the first two components l and f of a state, and how they are preserved by the transition function δ' of S'. After reading a word w S' is in state (l, f, g) where

— for every lookahead state $r \in R$, $l(r) = r'$, then $\delta^*_A(r, \text{REV}(w)) = r'$; and
— for every lookahead state $r \in R$, $f(r) = q$, then $\delta^*(q_0, w_r) = q$.

At the beginning S' is in state (l_0, f_0, g_0), where for every $r \in R$, $l_0(r) = r$ and $f_0(r) = q_0$. The component g_0 is discussed later.

Next we describe the transition function δ'. We assume S' to be in state (l, f, g), and to be reading the input symbol $a \in \Sigma$; we denote with l', f' the new values of the state components, and we only write the

parts that change. For every lookahead state $r \in R$, if $\delta_A(r, a) = r'$, and $\delta(q', r') = q$, then $l'(r) = l(r')$ and $f'(r) = q$.

Variable Summarization. Next, we describe how S' keeps track of the variable values. The natural approach for this problem would be that of keeping, for each state $r \in R$ and variable $x \in X$, a variable $g(r, x)$ containing the value of x in S, assuming the prefix read so far, was read by A starting in state r.

This natural approach, however, would cause the machine not to be copyless. Consider, for example the following scenario. Let r, r_1 and r_2 be look-ahead states in R such that, for some $a \in \Sigma$, $\delta(r_1, a) = \delta(r_2, a) = r$. Assume S only has one state $q \in Q$, and one variable $x \in X$. If S updates $\rho(q, r, x)$ to x, in order to perform the corresponding update in S' we would have to assign both $g(r, x)$ to both $g(r_1, x)$ and $g(r_2, x)$, and this assignment is not copyless.

Our solution to this problem relies on a symbolic representation of the update and a careful analysis of sharing. In the previous example, a possible way to represent such update is by storing the content of $g(r, x)$ into a variable z, and then remembering in the state the fact that both $g(r_1, x)$ and $g(r_2, x)$, now contain z as a value. In the construction, the above update is reflected by updating the state, without touching the variable values.

The set of variables X' contains $|R|(4|X_0||R|)$ type-0 variables, and $|R|(4|X_1||R|)$ type-1 variables. The set of parameters Π of S' is $\{\pi_i \mid 0 \leq i \leq 4|X||R|\}$. We will show later how this 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 $g(x)$. We define the unrolling $u : F(X') \mapsto E(X', \Sigma)$, of a symbolic variable representation as follows: given a forest $f = s_0(f_0) \dots s_n(f_n) \in F(X')$, $u(f) \equiv u_t(s_0, f_0) \dots u_t(s_n, f_n)$, where, if $s' \in X'$, and $f' = f'_0 \dots f'_m$, then $u_t(s', f') \equiv s'[\pi_0 \mapsto u(f'_0), \dots, \pi_m \mapsto u(f'_m)]$.

After reading the symbol an input word w S' is in a configuration $((l, f, g), \alpha)$ iff for every lookahead state $r \in R$, and variable $x \in X$, if $\delta^*(q_0, \text{REV}((a_1 \dots a_i)_r)) = (q, \alpha_1)$, and $u(g(r, x)) = s$, then $\alpha_1(x) = \alpha(s)$.

Counting Argument Invariants. Next, we describe how we keep the shape function g compact, allowing us to use a finite number of variables, while updating them in a copyless manner. The shape function g maintains the following invariants.

Single-Use:	each shape $g(r, x)$ is repetition-free: no variable $x' \in X'$ appears twice in $g(r, x)$;
Sharing Bound:	for all states $r, r' \in R$, $\sum_{x,y \in X} \mathcal{M}(g(r, x), g(r', y)) \leq \lvert X \rvert$;
Hole Placement:	for every type-1 variable $x \in X_1$, and state $r \in R$, there exists exactly one occurrence of ? in $g(r, x)$, and it does not have any children;
Horizontal compression:	for every $ff' \in \mathcal{SF}(g(r, x))$, such that $? \notin ff'$, then there must be a shape $g(r', x')$, with $(r', x') \neq (r, x)$, such that either $f \in S\mathcal{F}(g(r', x'))$ and $ff' \notin \mathcal{SF}(g(r', x'))$, or $f' \in \mathcal{SF}(g(r', x'))$ and $ff' \notin \mathcal{SF}(g(r', x'))$; and
Vertical compression:	for every $s(f) \in \mathcal{SF}(g(r, x))$, such that $? \notin s(f)$, then there must be a shape $g(r', x')$, with $(r', x') \neq (r, x)$, such that either $s() \in \mathcal{SF}(g(r', x'))$ and $s(f) \notin \mathcal{SF}(g(r', x'))$, or $f \in \mathcal{SF}(g(r', x'))$ and $s(f) \notin \mathcal{SF}(g(r', x'))$.

The first invariant ensures the bounded size of shapes. Notice that the second invariant implies that for each r, and for each $x \neq y$, $g(r, x)$ and $g(r, y)$ are disjoint. The second invariant implies that for every state r, the tree $g(r, x)$, for all x cumulatively, can have a total of $|X||R|$ maximal shared sub-forests, with respect to all other strings. The compression assured by the third and fourth invariants, then implies 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. The fourth invariant helps us dealing with variable substitution.

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 of each variable in X' is defined as follows: let $X'_0 = \{z_1, \dots, z_k\}$, $X'_1 = \{z'_1, \dots, z'_k\}$, $X_0 = \{x_1, \dots, x_i\}$, $X_1 = \{x'_1, \dots, x'_j\}$. For each type-0 variable $x_i \in X_0$, for each type-1 variable $x'_i \in X_1$, and look-ahead state $r \in R$, we have that $g_0(r, x_i) = z_i$, $z_i = \varepsilon$, $g_0(r, x'_j) = z'_j(?)$, and $z'_j = \pi_0$.

We assume S' to be in state (l, f, g), and to be reading the input symbol $a \in \Sigma$. We denote with g' the new value of g. We assume we are given a look-ahead state $r \in R$, such that $\delta_A(r, a)$ to be equal to r', and x and y are the variables to be the updated.

$\{x := w\}$: where w.l.o.g. $w = \langle a?b\rangle$. We first assume there exists an unused variable, and then show that such a variable must exist. Pick an unused variable z_f, then we update $g'(r,x) = z_f$, and set $z_f := \langle a?b\rangle$. Since the counting invariants are preserved, there must have existed an unused variable z_f.

$\{x := xy, y := \varepsilon\}$: we perform the following update: $g'(r,x) = g(r',x)g(r',y)$. Let $g(r',x) = s_1(f_1)\dots s_n(f_n)$ and $g(r',y) = s'_1(f'_1)\dots s'_m(f'_m)$. We now have two possibilities,
— there exists $l \leq m'$, and $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s_n or s'_1, but not $s_n s'_1$; or
— there does not exists $l \leq m'$, and $(r_1,x_1) \neq (r,x)$ 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; assume $f_n = t_1\dots t_i$ and $f'_1 = t'_1\dots t'_k$, $s_n := s_n s'_1[\pi_0 \mapsto \pi_i,\dots,\pi_k \mapsto \pi_{k+i}]$, and $g'(r,x) = s_1(f_1)\dots s_n(f_n f'_1)s'_2(f'_2)\dots s'_m(f'_m)$
In both cases, due to the preserved counting invariant, we can take an unused variable z_f and use it to update $g(r,y)$: $z_f := \varepsilon$, and $g'(r,y) = z_f$.

$\{x := x[y], y :=?)\}$: without loss of generality let x and y be type-1 variables. Let $s(?)$ be the subtree of $g(r',x)$ containing ?. We perform the following update: $g'(r,x) = g(r',x)\{?/g(r',y)\}$, where $a\{b/c\}$ replaces the node b of a with c. Let $g(r',y) = s'_1(f'_1)\dots s'_m(f'_m)$. For every s_l, we now have two possibilities:
— there exists $l \leq m'$, and $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s or s'_l, but not $s(s'_l)$; or
— there does not exists $l \leq m'$, and $(r_1,x_1) \neq (r,x)$ such that $g(r_1,x_1)$ contains s or s'_l; in this case we can compress: assume $f'_l = t_1\dots t_k$, then $s := s[\pi_i \mapsto s'_l[\pi_0 \mapsto \pi_i,\dots,\pi_k \mapsto \pi_{k+i}], \pi_{i+1} \mapsto \pi_{i+1+k},\dots,\pi_{n+k+1}]$, and $g'(r,x) = g'(r,x)\{s(s'_l)/s\}$.
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 3 shows an example of such an update.

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

We can show by a trivial inspection of the variable assignments that S' is copyless.

The output function F, of S' simply applies the unfolding 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. □

Mapping Ranked Trees. In a ranked tree, each symbol a has a fixed arity k, and an a-labeled node has exactly k children. Ranked trees can encode terms, and existing literature on tree transducers focuses primarily on ranked trees. Ranked trees can be encoded as nested words of a special form, and the definition of an STT can be simplified to use this structure. For simplicity of notation, we assume that there is a single 0-ary symbol $\mathbf{0} \notin \Sigma$, and every symbol in Σ is binary. The set $B(\Sigma)$ of *binary trees* over the alphabet Σ is then a subset of nested words defined by the grammar $T := \mathbf{0} \mid \langle a\, T\, T\, a\rangle$, for $a \in \Sigma$. We will use the more familiar tree notation $a\langle t_l, t_r\rangle$, instead of $\langle a\, t_l\, t_r\, a\rangle$, to denote a binary tree with a-labeled root and subtrees t_l and t_r as children. The definition of an STT can be simplified in the following way if we know that the input is a binary tree. First, we do not need to worry about processing of internal symbols. Second, we restrict to bottom-up STTs due to their similarity to bottom-up tree transducers, where the transducer returns, along with the state, values for variables ranging over output nested words, as a result of processing a subtree. Finally, at a call, we know that there are exactly two subtrees, and hence, the propagation of information across matching calls and returns using a stack can be combined into a unified combinator: the transition function computes the result corresponding to a tree $a\langle t_l, t_r\rangle$ based on the symbol a, and the results of processing the subtrees t_l and t_r.

A *bottom-up ranked-tree transducer* (BRTT) S from binary trees over Σ to nested words over Γ consists of a finite set of states Q; an initial state $q_0 \in Q$; a finite set of typed variables X equipped with a conflict relation η; a partial output function $F : Q \mapsto E_0(X,\Gamma)$ such that for each state q, the expression $F(q)$ is consistent with η; a state-combinator function $\delta : Q \times Q \times \Sigma \mapsto Q$; and a variable-combinator function $\rho : Q \times Q \times \Sigma \mapsto \mathcal{A}(X, X_l \cup X_r, \eta, \Gamma)$, where X_l denotes the set of variables $\{x_l \mid x \in X\}$, X_r denotes the set of variables $\{x_r \mid x \in X\}$, and the conflict relation η extends to these sets naturally: for every $x, y \in X$, if $\eta(x,y)$, then $\eta(x_l,y_l)$, and $\eta(x_r,y_r)$. The state-combinator extends to trees in $B(\Sigma)$: $\delta^*(\mathbf{0}) = q_0$ and $\delta^*(a\langle t_l,t_r\rangle) = \delta(\delta^*(t_l),\delta^*(t_r),a)$. The variable-combinator is used to map trees to valuations for X:

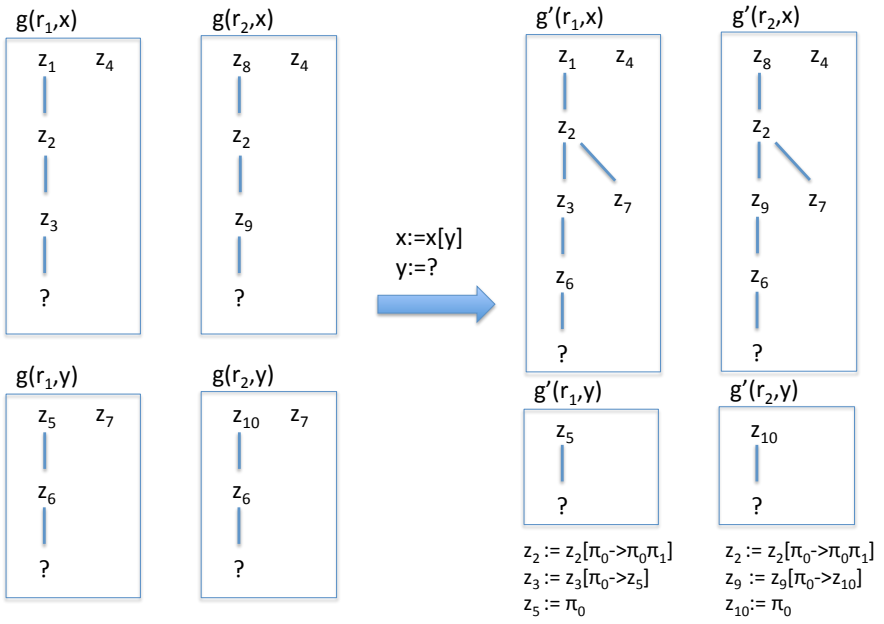

Fig. 3. Example of symbolic variable assignment. Every variable z_i belongs to X'. The depicted update represents the case in which we are reading the symbol a such that $\delta_A(r_1, a) = r_1$ and $\delta_A(r_2, a) = r_1$. Before reading a (on the left), the variables z_2, z_4, z_6, and z_7 are shared between the two representations of the variables at r_1 and r_2. In the new shape g' the hole ? 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_5), and reflect such 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 $g'(r_2, y)$).

$\alpha^*(\mathbf{0}) = \alpha_0$, where α_0 maps each type-0 variable to ε and each type-1 variable to ?, and $\alpha^*(a\langle t_l, t_r\rangle) = \rho(\delta^*(t_l), \delta^*(t_r), a)[X_l \mapsto \alpha^*(t_l)][X_r \mapsto \alpha^*(t_r)]$. That is, to obtain the result of processing the tree t with a-labeled root and subtrees t_l and t_r, consider the states $q_l = \delta^*(t_l)$ and $q_r = \delta^*(t_r)$, and valuations $\alpha_l = \alpha^*(t_l)$ and $\alpha_r = \alpha^*(t_r)$, obtained by processing the subtrees t_l and t_r. The state corresponding to t is given by the state-combinator $\delta(q_l, q_r, a)$. The value $\alpha^*(x)$ of a variable x corresponding to t is obtained from the right-hand side $\rho(q_l, q_r, a)(x)$ by setting variables in X_l to values given by α_l and setting variables in X_r to values given by α_r. Note that the consistency with conflict relation ensures that each value gets used only once. Given a tree $t \in B(\Sigma)$, let $\delta^*(t)$ be q and let $\alpha^*(t)$ be α. Then, if $F(q)$ is undefined then $[\![S]\!](t)$ is undefined, else $[\![S]\!](t)$ equals $\alpha(F(q))$ obtained by evaluating the expression $F(q)$ according to valuation α.

Theorem 3.17 (Expressiveness of Ranked Tree Transducers). *A partial function from $B(\Sigma)$ to $W_0(\Gamma)$ is STT-definable iff it is BRTT-definable.*

Proof Sketch. ***From STT to BRTT*** $\Rightarrow$**.** Using Theorem 3.7, let $S = (Q, P, q_0, X, F, \eta, \delta, \rho)$ be a binary bottom-up STT. We construct a BRTT $S' = (Q', q_0', X', F', \eta', \delta', \rho')$ equivalent to S. We can assume w.l.o.g. that the set of variables X is partitioned into two disjoint sets $X_l = \{x_1^L, \ldots, x_n^L\}$ and $X_r = \{x_1^R, \ldots, x_n^R\}$ such that, given a tree $t = a\langle t_l, t_r\rangle$, 1) after processing the left child t_l, all the variables in X_l depend on the run over t_l, while all the variables in X_r are reset to their initial values, and 2) after processing the second child, the values of X_r only depend on t_r (do not use any variable $x^p \in X^p$), and every variable $x \in X_l$ is exactly assigned the corresponding value x^p stored on the stack. In summary, every variable in X_r only depends on the second child, and every variable in X_L only depends on the first child. Such variables can only be combined at the return $a\rangle$ at the end of t. An STT of this form can be obtained by delaying the variable update at the return at the end t_r to the next step in which $a\rangle$ is read.

Now that we are given a bottom-up STT, the construction of S' is similar to the one showed in Theorem 3.7. Each state $q \in Q'$ contains a function $f : Q \mapsto Q$, that keeps track of the executions of S for every possible starting state. After processing a tree t, S' is in state f, such that for every $q \in Q$, $f(q) = q'$ iff when S reads the tree t starting in state q, it will end up in state q'. Similarly, the BRTT S' uses multiple variables to keep track of all the possible states with which a symbol could have been processed. The BRTT S' has

set of variables $X' = \{x_q \mid x \in X, q \in Q\}$. After processing a tree t, for every $x \in X$, and $q \in Q$: if after S processes t starting in state q, x contains the value s, then x_q also contains the value s.

Next we describe how these components are updated. Let f_l and f_r be the states of S' after processing the children t_l and t_r of a tree $a\langle t_l, t_r\rangle$. We denote with f' the new state after reading a. For every state $q, \in Q$, and variable $x \in X$, if $\delta_c(q, \langle a) = (q_1, p)$, $f_l(q_1) = q_2$, $f_r(q_2) = q_3$, and $\delta_r(q_3, p, \langle a) = q_4$, then $f'(q) = q_4$, and x_q is assigned the value $\rho(q_3, p, \langle a, x)$, in which every variable in X_l and X_r is is replaced with the corresponding variables in X'_l and X'_R. For every state f, the output function of F' of S' is then defined as $F'(f) = F(f(q_0))$. The conflict relation η' of S' has the following rules:

(1) for every $x, y \in X$, and $q \neq q' \in Q$, $\eta'(x_q, y_{q'})$, and
(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 3.7.

From BRTT to STT $\Leftarrow$**.** Given a BRTT $S = (Q, q_0, X, F, \eta, \delta, \rho)$, we construct an STT $S' = (Q', P', q'_0, X', F', \eta', \delta', \rho')$ equivalent to S. The STT S' simulates the execution of S while reading the input nested word. The states and stack states of S' are used keep track of whether the current child is a first or second child. The STT S' has set of states $Q' = Q \times (\{l, r\} \cup Q)$, initial state $q'_0 = (q_0, l)$, and set of stack state $P' = Q \times \{l, r\}$. Given a tree $\langle a\ t_l, t_r\ a\rangle$, the STT S' maintains the following invariant:

— right before processing the first child t_l, S' is in state (q_0, l);
— right before processing the second child t_l, S' is in state (q_1, r); and
— right after processing the second child t_r, S' is in state (q_1, q_2).

We now define the transition relations δ' that preserves the invariant defined above. Let's assume S' is in state $q = (q_1, d)$, and it is processing the symbol a.

a is a call $\langle b$: the STT S' resets the control to the initial state and pushes q_1 on the stack: $\delta'(q_1, b) = (q'_0, q_1)$; and

a is a return $b\rangle$: we first of all observe that since the input is a binary tree d cannot have value r when reading a return symbol. If the state popped from the stack is $p = (q_p, d_p)$, then

End of First Child: if $d_p = l$, then
- Leaf: if $d = l$, and $\delta(q_0, q_0, b) = q_2$ then $\delta'(q, p, b) = (q_2, r)$, and
- 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 $d_p = r$, then
- Leaf: if $d = l$, and $\delta(q_0, q_0, b) = q_2$, then $\delta'(q, p, b) = (q', q_2)$, and
- 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 the combined when processing the parent. The STT S' maintains the following invariant: given a tree $\langle a\ t_l, t_r\ a\rangle$, and $d \in \{l, r\}$, after processing the first child t_d, every variable $x_d \in X'_d$ will contain the value of x computed by S after processing the tree t_d.

We can now describe the variable update function ρ' of S'. Given a variable $x \in X'$, let the function $\textsc{ini}(x)$ be the function that returns the initial value of a variable x, that is defined as: 1) if x is a type-0 variable, then $\textsc{ini}(x) = \varepsilon$, and 2) if x is a type-1 variable, then $\textsc{ini}(x) =?$. Let's assume S' is in state $q = (q_1, d)$, it is processing the symbol a, and updating the variable $x \in X'$:

a is a call $\langle b$: every variable is copied on the stack $\rho'(q, b, x) = x$;

a is a return $b\rangle$: if the state popped from the stack is $p = (q_p, d_p)$, then

End of First Child: if $d_p = l$, then
- if $x \in X_L$, then
  - Leaf: if $d = l$, then $\rho'(q, p, b, x) = \rho(q_0, q_0, b, x)$, and
  - Not a Leaf: if $d = q_2 \in Q$, then $\rho'(q, p, b, x) = \rho(q_1, q_2, b, x)$.
- if $x \in X_R$, then $\rho'(q, p, b, x) = \textsc{ini}(x)$.

End of Second Child: if $d_p = r$, then
- if $x \in X_L$, then $\rho'(q, p, b, x) = x^p$, and
- if $x \in X_R$, then
  - Leaf: if $d = l$, then $\rho'(q, p, b, x) = \rho(q_0, q_0, b, x)$, and
  - Not a Leaf: if $d = q_2 \in Q$, then $\rho'(q, p, b, x) = \rho(q_1, q_2, b, x)$.

The conflict relation η' of S' is defined as: for every $x, y \in X$, and $d \in \{L, R\}$, if $\eta(x, y)$, then $\eta'(x_d, y_d)$. The consistency of η is trivial. Finally, the output function F' is defined as follows. For every state $(q, d) \in Q'$, if $d = r$, $F'(q, i) = F(q)$, otherwise F' is undefined. □

3.8. 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.

Mapping Nested Words to Strings. Each variable of an STT stores a potential output fragment. These fragments get updated by addition of outputs symbols, concatenation, and insertion of a nested word in place of the hole. If we disallow the substitution operation, then the STT cannot manipulate the hierarchical structure in the output. More specifically, if all variables of an STT are type-0 variables, then the STT produces outputs that are strings over Γ. The set of expressions used in the right-hand sides can be simplified to $E_0 := \varepsilon \mid a \mid x_0 \mid E_0 E_0$. That is, each right-hand side is a string over $\Gamma \cup X$. Such a restricted form of STT is called a *streaming tree-to-string transducer* (STST). While less expressive than STTs, this class is adequate to compute all tree-to-string transformations, that is, if the final output of an STT is a string over Γ, then it does not need to use holes and substitution.

THEOREM 3.18 (STST EXPRESSIVENESS). *A partial function from $W_0(\Sigma)$ to Γ^* is STT-definable iff it is STST-definable.*

Proof Sketch. Since STST are also STTs the $\Leftarrow$ direction of the proof is immediate.

We now prove the $\Rightarrow$ direction. Given an STT S that only outputs strings over Γ^*, we can build an equivalent STST S' as follows. The goal of the construction is eliminating all the type-1 variables from S. This can be done by replacing each type-1 variable x in S with two type-0 variables x_l, x_r in S' representing the values to the left and to the right of the ? in x. If after reading a nested word w, the type-1 variable x of S contains the value $w_l?w_r$, then the type-0 variables x_l and x_r of S' respectively contain the values w_l and w_r. Notice that this cannot be done in general for SSTs, because w_l and w_r might not be well-matched nested words. The states, and state transition function of S' are the same as for S, and the variable update function and the output function can be easily derived. If η is the conflict relation of S, then the conflict relation η' of S' is defined as follows: given two type-1 variables x, y of S, $\eta(x, y)$, $x \neq y$, and $d, d' \in \{l, r\}$, then $\eta'(x_d, y_{d'})$. The type-0 case is defined analogously. □

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) [Alur and Cerný 2011].

Mapping Nested Words to Ranked Trees. Suppose we are interested in outputs that are binary trees in $B(\Gamma)$. Then, variables of the transducer can take values that range over such binary trees, possibly with a hole. The internal symbols, and the concatenation operation, are no longer needed in the set of expressions. More specifically, the grammar for the type-0 and type-1 expressions can be modified as:

$$\begin{aligned} E_0 &:= \mathbf{0} \mid x_0 \mid a\langle\, E_0\, E_0 \,\rangle \mid E_1[E_0] \\ E_1 &:= \ ? \mid x_1 \mid a\langle\, E_0\, E_1 \,\rangle \mid a\langle\, E_1\, E_0 \,\rangle \mid E_1[E_1] \end{aligned}$$

where $a \in \Gamma$, $x_0 \in X_0$ and $x_1 \in X_1$. To define transformations from ranked trees to ranked trees, we can use the model of bottom-up ranked-tree transducers with the above grammar.

4. 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). Our proof relies on the known equivalence between MSO and Macro Tree Transducers over *ranked trees*.

4.1. MSO for Nested Word Transductions

Formulas in monadic second-order logic (MSO) can be used to define functions from (labeled) graphs to graphs [Courcelle 1994]. We adapt this general definition for our purpose of defining transductions over nested words. A nested word $w = a_1 \dots a_k$ over Σ is viewed as an edge-labeled graph G_w with $k + 1$ nodes $v_0 \dots v_k$ such that (1) there is a (linear) edge from each v_{j-1} to v_j, for $1 \leq j \leq k$, labeled with the symbol

$a_j \in \Sigma$, and (2) for every pair of matching call-return positions i and j, there is an unlabeled (nesting) edge from v_{i-1} to v_{j-1}. The monadic second-order logic of nested words is given by the syntax:

$$\phi := a(x,y) \mid X(x) \mid x \rightsquigarrow y \mid \phi \vee \phi \mid \neg\phi \mid \exists x.\phi \mid \exists X.\phi$$

where $a \in \Sigma$, x, y are first-order variables, and X is a second-order variable. The semantics is defined over nested words in a natural way. The first-order variables are interpreted over nodes in G_w, while set variables are interpreted over sets of nodes. The formula $a(x,y)$ holds if there an a-labeled edge from the node x to node y (this can happen only when y is interpreted as the linear successor position of x), and $x \rightsquigarrow y$ holds if the nodes x and y are connected by a nesting edge.

An *MSO nested-word transducer* Φ from input alphabet Σ to output alphabet Γ consists of a finite copy set C, node formulas ϕ^c, for each $c \in C$, each of which is an MSO formula over nested words over Σ with one free first-order variable x, and edge formulas $\phi^{c,d}$ and $\phi_a^{c,d}$, for each $a \in \Gamma$ and $c, d \in C$, each of which is an MSO formula over nested words over Σ with two free first-order variables x and y. Given an input nested word w, consider the following output graph: for each node x in G_w and $c \in C$, there is a node x^c in the output if the formula ϕ^c holds over G_w, and for all such nodes x^c and y^d, there is an a-labeled edge from x^c to y^d if the formula $\phi_a^{c,d}$ holds over G_w, and there is a nesting edge from x^c to y^d if the formula $\phi^{c,d}$ holds over G_w. If this graph is the graph corresponding to the nested word u over Γ then $[\![\Phi]\!](w) = u$, and otherwise $[\![\Phi]\!](w)$ is undefined. A nested word transduction f from input alphabet Σ to output alphabet Γ is *MSO-definable* if there exists an MSO nested-word transducer Φ such that $[\![\Phi]\!] = f$.

By adapting the simulation of string transducers by MSO [Engelfriet and Hoogeboom 2001; Alur and Cerný 2010], 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 4.1 (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 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 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 using other 4 copies.

We now explain how the STT variable updates are captured by the MSO transducer. At step 0, x instantiated to ? by adding an ε-labeled edge from the x_i node to the $x_{i?}$ node and from the $x_{o?}$ node to the x_o node. Consider an internal position i, and consider the variable assignment $x := ax[c?]$. This means that the value of x for column i is the value of x in column $i-1$, preceded by the symbol a, where we add a c before the parameter position in $i-1$. To reflect this assignment we insert an a-labeled edge from the x_i node in column i to the x_i node in column $i-1$, a c-labeled edge from the $x_{i?}$ node in column $i-1$ to the $x_{i?}$ node in column i (this reflects adding to the left of the ?), an ε-labeled edge from the $x_{o?}$ node in column i to the $x_{o?}$ node in column $i-1$, and an ε-labeled edge from the x_o node in column $i-1$ to the x_o node in column i.

We again use Figure 4 to show how variables are stored on the stack. At the call step $i+1$, the assignment $x_p := x$ is reflected by the ε-labeled edges between the x_p nodes in column $i+1$ and the x nodes in column i. The edges are added in a similar way as for the previous assignment. The value of x_p in column $i+1$ is preserved unchanged until the corresponding matching return position is found. At the return step j, the value of x can depend on the values of x in column $j-1$, and the value of x_p on the top stack, which is captured by the input/output nodes for x_p in column $j-1$. The $j-1$ position can be identified uniquely using the matching relation $\rightsquigarrow$. Even though it is not shown in figure, at position j we have to add ε edges from the x_p nodes at position j to the x_p nodes at position i to represent the value of x_p that now is on the top of the stack.

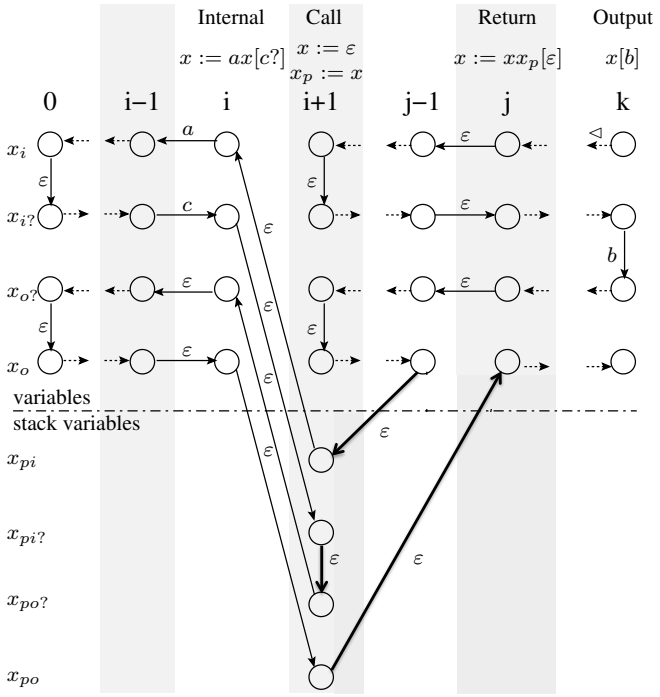

Fig. 4. Encoding STT computation in MSO. The bold lines link the current variable values to the values on top of the stack, and are all added when processing the return position j.

To represent the final output, we use an additional column k. In the example, the output expression is $x[b]$. We mark the first edge from x_i by a special symbol $\triangleleft$ 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 ε edges.

Notice that in this exposition, we have assumed that in the MSO transducer, edges can be labeled with strings over the output alphabet (including ε) instead of single symbols. It is easy to show that allowing strings to label the edges of the output graph does not increase the expressiveness of MSO transducers. Also notice that not every node will appear in the final output string. An MSO transducer able to remove the useless edges and nodes can be defined. Using closure under composition we can then build the final transducer.

Some extra attention must be paid to add the matching edges in the output, but since the output at every point is always a well-matched nested word, the matching relation over the output nested word can be induced by inspection of each assignment (i.e. the matching edges are always between nodes in the same column). □

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 $B(\Sigma)$ is defined by

$$\begin{aligned} nw\_bt(\varepsilon) &= \mathbf{0} \\ nw\_bt(aw) &= a\langle\, nw\_bt(w), \mathbf{0}\,\rangle \\ nw\_bt(\langle a\, w_1\, b\rangle\, w_2) &= a\langle\, nw\_bt(w_1), b\langle\, nw\_bt(w_2), \mathbf{0}\,\rangle\,\rangle \end{aligned}$$

Notice that the tree corresponding to a nested word w has exactly one internal node for each position in w. Observe that nw_bt is a one-to-one function, and in particular, the encodings of the two nested words aaa and $\langle a\, a\, a\rangle$ differ:

— $nw\_bt(aaa) = a\langle\, a\langle\, a\langle \mathbf{0}, \mathbf{0}\,\rangle, \mathbf{0}\,\rangle, \mathbf{0}\,\rangle$, and
— $nw\_bt(\langle aaa\rangle) = a\langle\, a\langle \mathbf{0}, \mathbf{0}\,\rangle, a\langle \mathbf{0}, \mathbf{0}\,\rangle\,\rangle$.

We can define the inverse partial function *bt_nw* from binary trees to nested words as follows: given $t \in B(\Sigma)$, 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 4.2 (NESTED-WORDS BINARY-TREES CORRESPONDENCE). *$nw\_bt : W_0(\Sigma) \mapsto B(\Sigma)$, and $bt\_nw : B(\Sigma) \mapsto W_0(\Sigma)$ are both STT-definable transductions.*

Proof. We prove the two statements in the order.

***STT for** nw_bt.* The transduction *nw_bt* can be performed by an STT S that basically simulates its inductive definition. and only needs one type-1 variable x. When processing the input nested word $w = a_1 \dots a_n$, after reading the symbol a_i, $x[\mathbf{0}]$ contains the value of $nw\_bt(\text{WMS}(w, i))$ (see Theorem 3.7 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 $F(q) = x[\mathbf{0}]$. Next we define the update functions. For every $a, b \in \Sigma$,

— $\delta_i(q, a) = q$, $\delta_c(q, a) = (q, a)$, and $\delta_r(q, a, b) = q$; and
— $\rho_i(q, a, x) = x[a\langle ?, \mathbf{0}\rangle]$, $\rho_c(q, a, x) = x$, and $\rho_r(q, a, b) = x^p[a\langle x[\mathbf{0}], b\langle ?, \mathbf{0}\rangle\rangle]$.

***STT for** bt_nw.* The translation *bt_nw* can be implemented by the following BRTT S'. The BRTT S' has three type-0 variables x_i, x_c, x_r such that after processing the tree t, 1) x_i contains the value of $bt\_nw(t)$, assuming t's root was representing an internal symbol, 2) x_c contains the value of $bt\_nw(t)$, assuming t's root was representing a call, and 3) x_r contains the value of $bt\_nw(t)$, assuming t's root was representing a return. The BRTT S' has set of states $Q' = \{q_{\mathbf{0}}\} \cup (\{q_a \mid a \in \Sigma\} \times \{C, IR\})$, and initial state $q_{\mathbf{0}}$. The $\{C, IR\}$ component is used to remember whether the last symbol S' read was a call, or an internal or return symbol. The BRTT S' is in a state $(q_a, \_)$ after processing a tree rooted with the symbol a. The output function F' is defined as follows: for every $a \in \Sigma$, $F'(q_a, IR) = x_i$, $F'(q_a, C) = x_c$, and F' is undefined otherwise. Next we define the update functions. When a variable is not assigned, we assume it is set to ε. For every symbols $a \in \Sigma$,

Right Child is $\mathbf{0}$: for every $q' \in Q$, $\delta(q', q_{\mathbf{0}}, a) = (q_a, IR)$, and
— if $q' = q_{\mathbf{0}}$, then $\rho(q_{\mathbf{0}}, q_{\mathbf{0}}, a, x_i) = a$;
— $q' = (q_b, C)$ for some $b \in \Sigma$, then $\rho((q_b, C), q_{\mathbf{0}}, a, x_i) = ax_c^l$, $\rho((q_b, IR), q_{\mathbf{0}}, a, x_r) = x_c^l$; and
— $q' = (q_b, IR)$ for some $b \in \Sigma$, then $\rho((q_b, IR), q_{\mathbf{0}}, a, x_i) = ax_i^l$, and $\rho((q_b, IR), q_{\mathbf{0}}, a, x_r) = x_i^l$.

right child is not $\mathbf{0}$: for every $q_l, q_r \in Q$, $\delta(q_l, q_r, a) = (q_a, C)$, and q_r must be of the form (q_b, IR) for some $b \in \Sigma$. The variable update is defined as follows:
— if $q_l = q_{\mathbf{0}}$, then $\rho(q_{\mathbf{0}}, (q_b, IR), a, x_c) = \langle a\rangle b$;
— $q' = (q_c, C)$ for some $c \in \Sigma$, then $\rho((q_c, C), (q_b, IR), a, x_c) = \langle ax_c^l\rangle bx_r^r$; and
— $q' = (q_c, IR)$ for some $c \in \Sigma$, then $\rho((q_c, IR), (q_b, IR), a, x_c) = \langle ax_i^l\rangle b$.

Finally, the conflict relation of S' only contains $\eta'(x_i, x_r)$ □

For a nested-word transduction f from $W_0(\Sigma)$ to $W_0(\Gamma)$, we can define another transduction $\tilde{f}$ that maps binary trees over Σ to binary trees over Γ: given a binary tree $t \in B(\Sigma)$, 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 4.3 (ENCODING NESTED-WORD TRANSDUCTIONS). *If f is an MSO-definable transduction from $W_0(\Sigma)$ to $W_0(\Gamma)$, then the transduction $\tilde{f} : B(\Sigma) \mapsto B(\Gamma)$ 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.

4.2. Macro Tree Transducers

A Macro Tree Transducer (MTT) [Engelfriet and Vogler 1985; Engelfriet and Maneth 1999] is a tree transducer in which the translation of a tree may not only depend 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 and Vogler 1985; Engelfriet and Maneth 1999] 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 $B(\Sigma)$ to $B(\Gamma)$ consists of a finite set Q of ranked states, a list $Y = y_1, \dots y_n$ of parameter symbols, variables $X = \{x_l, x_r\}$ used to refer to input subtrees, an initial state q_0, a finite set R of look-ahead types, an initial look-ahead type r_0, a look-ahead combinator $\theta : \Sigma \times R \times R \mapsto R$, and the transduction function Δ. For every state q and every look-ahead type r, $\Delta(q, r)$ is a ranked tree over the alphabet $(Q \times X) \cup \Gamma \cup Y$, where the rank of a label (q, x) is the same as the rank of q, the rank of an output symbol $a \in \Gamma$ is 2, and the rank of each parameter symbol is 0 (that is, only leaves can be labeled with parameters).

The look-ahead combinator is used to define look-ahead types for trees: $\theta^*(\mathbf{0}) = r_0$ and $\theta^*(a\langle\, s_l, s_r\,\rangle) = \theta(a, \theta^*(s_l), \theta^*(s_r))$. Assume that only the tree $\mathbf{0}$ has the type r_0, and for every state q, $\Delta(q, r_0)$ is a tree over $\Gamma \cup Y$ (the variables X are used to refer to immediate subtrees of the current input tree being processed, and the type r_0 indicates that the input tree has no subtrees).

The MTTR M rewrites the input binary tree s_0, and at every step the output tree is a ranked tree with nodes labeled either with an output symbol, or with a pair consisting of a state of the MTTR along with a subtree of the input tree. Let $\mathcal{T}(s_0)$ denote the set of all subtrees of the input tree s_0. Then, the output t at any step is a ranked tree over $(Q \times \mathcal{T}(s_0)) \cup \Gamma \cup \{\mathbf{0}\}$. The semantics of the MTTR is defined by the derivation relation, denoted by $\Rightarrow$, over such trees. Initially, the output tree is a single node labeled with $[q_0, s_0]$. Consider a subtree of the output of the form $u = [q, s](t_1, \dots t_n)$, that is, the root is labeled with the state q of rank n, with input subtree s, and children of this node are the output subtrees $t_1, \dots t_n$. Suppose the look-ahead type of the input subtree s is r, and let s_l and s_r be the children of the root. Let χ be the tree obtained from the tree $\Delta(q, r)$ by replacing input variables x_l and x_r appearing in a node label with the input subtrees s_l and s_r respectively, and replacing each leaf labeled with a parameter y_l by the output subtree t_l. Then, in one step, the MTTR can replace the subtree u with the tree χ. The rewriting stops when all the nodes in the output tree are labeled only with output symbols. That is, for $s \in B(\Sigma)$ and $t \in B(\Gamma)$, $[\![M]\!](s) = t$ iff $[q_0, s] \Rightarrow^* t$.

In general, MTTs are more expressive than MSO. The restrictions needed to limit the expressiveness rely on the so-called *single-use* and *finite copying*, that enforce an MTT to process every subtree in the input a bounded number of times. Let M be an MTTR.

(1) The MTTR M is single use restricted in the parameters (SURP) if for every state q and every look-ahead type r, each parameter y_j occurs as a node-label at most once in the tree $\Delta(q, r)$.
(2) The MTTR M is finite-copying in the input (FCI) if there exists a constant K such that for every tree s over Σ and subtree s' of s, if the (intermediate) tree t is derivable from $[q_0, s]$, then t contains at most K occurrences of the label $[q, s']$ (and thus, each input subtree is processed at most K times during a derivation).

The following theorem is proved in [Engelfriet and Maneth 1999].

THEOREM 4.4 (REGULARITY FOR MTTS). *A ranked-tree transduction f is MSO-definable iff there exists an MTTR M with SURP/FCI such that $f = [\![M]\!]$.*

4.3. 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 4.5 (MTTS TO BRTTS). *If a ranked-tree transduction $f : B(\Sigma) \mapsto B(\Gamma)$ 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 3.17 we can show that these are equivalent to multi-parameter STTs and therefore using Theorem 3.14 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 and each step uses one of the properties of the MTT.

(1) We construct a BRTT S_1, computing the transduction $f_1 : B(\Sigma) \mapsto B(R)$, where each input element is replaced by its regular look-ahead state;
(2) We construct an STT S_2, computing the transduction $f_2 : B(R) \mapsto B(R')$, 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 $f_3 : B(R') \mapsto B(\Gamma)$. This part relies on the SURP restriction;
(4) Finally, we use the fact that STTs are closed under composition (Theorem 3.15), and the equivalence of STTs and BRTTs (Theorem 3.17), to show that $f = f_1 \cdot f_2 \cdot f_3$ is a BRTT definable transduction.

Step 1. The BRTT S_1 simulates the look-ahead automaton of M by following the transition relation θ. The set of states of S_1 is R, with initial state r_0, and it only has one type-0 variable x. For every symbol $a \in \Sigma$, and states $r_1, r_2 \in R$, the transition function δ_1, and the variable update function ρ_1 of S_1 are as follows: if $\theta(r_1, r_2, a) = r$, then $\delta_1(r_1, r_2, a) = r$, and $\rho_1(r_1, r_2, a, x) = r\langle x_l x_r\rangle$. Finally, for every $r \in R$, $F(r) = x$.

Step 2. STTs can also be viewed as a top down machine, and f_2 is in fact a top-down relabeling. Every subtree s can be processed at most K times, and we can therefore use S_2 to label s with the ordered sequence of states that processes it. So given a tree over $B(R)$, S_2 outputs a tree over $B(R')$, where $R' = R \times Q^{\leq K}$, and $Q^{\leq K} = \bigcup_{0 \leq k \leq K} Q^k$.

The states and stack states of S_2 are defined by the set $Q_2 = P_2 = Q^{\leq K} \cup (Q^{\leq K} \times Q^{\leq K})$. The initial state $q_0^2 = q_0 \in Q^{\leq K}$ means that the root is only processed by q_0). The construction of S_2 then maintains the following invariants: assume t_l and t_r are the left and right children of a node, $m_l \in Q^{\leq K}$ is the sequence of states that processes t_l in M, and $m_r \in Q^{\leq K}$ is the sequence of states that processes t_r in M;

— before processing t_l, S_2 is in state (m_l, m_r); and
— before processing t_r (after processing t_l), S_2 is in state m_r.

The states m_i can be obtained directly from the right hand sides of the rules of the MTT. It's now trivial to do the corresponding labeling using the information stored in the state. Given a state $q \in Q$, and a symbol $r_1 \in R$, let $\text{SEQ}_d(q, r_1)$, with $d \in \{l, r\}$, be the sequence of states processing the child s_l in $\Delta(q, r_1)$, assuming a linearization of such tree. For every sequence $m = q_1 \dots Q^{\leq K}$, and symbol $r_1 \in R$, $\text{SEQ}_d(s, r_1) = \text{SEQ}_d(q_1, r_1) \dots \text{SEQ}_d(q_n, r_1)$. The STT S_2 only has one variable x.

We can now define the transition relation δ_2, and the variable update function ρ_2 of S_2. For every states $m \in Q_2$, $m' \in Q^{\leq K}$, $(m_1, m_2) \in Q^{\leq K} \times Q^{\leq K}$, and for every symbol $r \in R$,

r_1 is a call $\langle r_2$: store the variables on the stack and update the state consistently
— $\delta_2(m', r_2) = (m_1, m')$, where $m_1 = (\text{SEQ}_l(m', r_2), \text{SEQ}_r(m', r_2))$, and $\delta_2((m_1, m_2), r_2) = (m_3, (m_1, m_2))$, where $m_3 = (\text{SEQ}_l(m_1, r_2), \text{SEQ}_r(m_1, r_2))$;
— $\rho_2(m', r_2, x) = x$, and $\rho_2((m_1, m_2), r_2, x) = x$;
r is a return $r_2\rangle$: use the values on the stack to compute the labeling
— $\delta_2(m, (m_1, m_2), r_2) = m_2$, and $\delta_2(m, m', r_2) = m$;
— $\rho_2(m, (m_1, m_2), r_2, x) = \langle (r, m_1)\ x\ (r, m_1)\rangle$, and
$\rho_2(m, m', r_2, x) = x^p\ \langle (r_2, m')\ x\ (r_2, m')\rangle$.

Finally, the output function F_2 of S_2 outputs x for every possible $q \in Q_2$.

Step 3. This last step relies on the SURP property of the MTT M. We notice that, when processing bottom-up the MTT parameter updates behave in a different way as when processing top-down: to perform the top-down parameter update $y_1 := a(y_2)$ in bottom-up manner, we need to use the multi-parameter BRTT parameter substitution $x := x[y_1 \mapsto a(y_2)]$, where now, the new visible parameter is y_2. We now formalize this idea.

We construct a multi-parameter BRTT $S_3 = (Q_3, q_{03}, \Pi_3, \eta_3, X_3, F_3, \delta_3, \rho_3)$ from $B(R') \mapsto B(\Gamma)$, 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 $X_3 = \{x_1, \dots, x_K\}$ contains K variables. After processing a tree t, x_i contains the result of M processing the tree t starting in q_i, with the possible holes given from the parameters. At the beginning all the variable values are set to ε. The parameter set Π_3 is Y.

Next, we define the update functions of S_3. We start from the leaf rules, in which both the $\mathbf{0}$ children are labeled with the empty sequence. Let's assume the current leaf is (r, m), where $m = q_1 \dots q_j$. We assume w.l.o.g., that all the states have exactly K parameters. For every $q_i \in m$, if $\Delta(q_i, r) = t_i(y_1, \dots, y_K)$, we

update $x_i := t_i(y_1, \ldots, y_K)$, where $y_1, \ldots, y_K \in \Pi$. Since the MTT is SURP, there is at most one occurrence of each y_i.

We now analyze the binary node rules. Let's assume the node we are processing the input node (r, m), where $m = q_1 \ldots q_j$. For every $q_i \in m$, $\Delta(q_i, r)$ is of the form

$$t_i(Y, (q^i_{l,1}, s_l), \ldots, (q^i_{l,a_i}, s_l), (q^i_{r,1}, s_r), \ldots, (q^i_{r,b_i}, s_r))$$

where $q^i_{l,1} \ldots q^i_{l,a_i}$ is the sequence of states processing the left subtree, and $q^i_{r,1} \ldots q^i_{r,b_i}$ is the sequence of states processing the right subtree.

When picking the concatenation of all the $\Delta(q_i, r)$, we have that by the construction of S_2, the left child (similarly for the right), must have been labeled with the sequence $m_l = q^1_{l,1} \ldots q^1_{l,a_1} \ldots q^j_{l,1} \ldots q^j_{l,a_j}$ such that $|m_l| \leq K$. Moreover, we have that for all $x_i \in X_l$ (similarly for X_r), x_i contains the output of M when processing the left child of the current node starting in state q_i, where q_i is the i-th element of the sequence m_l and assuming the parameter are not instantiated yet

Now we have all the ingredients to complete the rule. The right hand side of a variable x_i contains the update corresponding to the rule in M where we replace every state with the corresponding variable in the linearization stated above and parameters are updated via substitution. Since ρ is copyless η_3 is trivial. The output function F_S simply outputs x_1, i.e. the transformation of the input tree starting in q_0.

Step 4. We use Theorem 3.15, and Theorem 3.17 to compose all the transformations and build the final BRTT equivalent to the MTT M. □

THEOREM 4.6 (MSO EQUIVALENCE). *A nested-word transduction $f : W_0(\Sigma) \mapsto W_0(\Gamma)$ is STT-definable iff it is MSO-definable.*

Proof. From Theorems 3.15, 3.17, 4.1, 4.2, 4.4, and 4.5. □

5. DECISION PROBLEMS

In this section, we show that a number of analysis problems are decidable for STTs.

5.1. Output Analysis

Given an input nested word w over Σ, and an STT S from Σ to Γ, consider the problem of computing the output $[\![S]\!](w)$. To implement the operations of the STT efficiently, we can store the nested words corresponding to variables in linked lists with reference variables pointing to positions that correspond to holes. Each variable update can be executed by changing only a number of pointers that is proportional to the number of variables.

PROPOSITION 5.1 (COMPUTING OUTPUT). *Given an input nested word w and an STT S with k variables, the output nested word $[\![S]\!](w)$ can be computed in time $O(k|w|)$ in a single pass.*

Proof. A naive implementation of the transducer would cause the running time to be $O(k|w|^2)$ due to the 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 4.1. 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 since at every point in the run the graph only needs to represent the top of the stack. □

The second problem we consider corresponds to *type-checking*: given a regular language L_{pre} of nested words over Σ, a regular language L_{post} of nested words over Γ, and an STT S from Σ to Γ, the type-checking problem is to determine if $[\![S]\!](L_{pre}) \subseteq L_{post}$ (that is, if for every $w \in L_{pre}$, $[\![S]\!](w) \in L_{post}$).

THEOREM 5.2 (TYPE-CHECKING). *Given an STT S from Σ to Γ, an NWA A accepting nested words over Σ, and an NWA B accepting nested words over Γ, checking $[\![S]\!](L(A)) \subseteq L(B)$ is solvable in* EXPTIME,

and more precisely in time $O(|A|^3 \cdot |S|^3 \cdot n^{kn^2})$ where n is the number of states of B, and k is the number of variables in S.

Proof Sketch. The construction is similar to the one of Theorem 3.15, therefore we only present a sketch. Given S, A, and B, we construct an NWA P, that accepts a nested word w iff w is accepted by A and $[\![S]\!](w)$ is not accepted by B. This is achieved by summarizing the possible executions of B on the variable values of S. The states of P are triplets (q_A, q_S, f), such that

— q_A is the current state of A,
— q_S is the current state of S, and
— for every variable x of S, and states q_1, q_2 in B, $f(x, q_1, q_2)$ is a pair of states (q_1', q_2') of B, such that if the value of x is $w_1?w_2$:
  — if B reads w_1 starting in state q_1, then it ends in state q_1' and produces some stack Λ, and
  — if B raads w_2 starting in state q_2, and with stack Λ, then it ends in state q_2'.

The final states of the machine are those where A is final, and the summary of the output function of the current state of Q_S of S leads to a non accepting state in B. □

As noted in Proposition 3.2, the image of an STT is not necessarily regular. However, the pre-image of a given regular language is regular, and can be computed. Given an STT S from input alphabet Σ to output alphabet Γ, and a language $L \subseteq W_0(\Gamma)$ of output nested words, the set $PreImg(L, S)$ consists of input nested words w such that $[\![S]\!](w) \in L$.

THEOREM 5.3 (COMPUTING PRE-IMAGE). *Given an STT S from Σ to Γ, and an NWA B over Γ, there is an* EXPTIME *algorithm to compute an NWA A over Σ such that $L(A) = PreImg(L(B), S)$. The NWA A has size $O(|S|^3 \cdot n^{kn^2})$ 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 5.2, 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. □

5.2. 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', [Alur and Cerný 2010; 2011] shows how to 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. 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. Results in [Esparza 1997; Seidl et al. 2004; Kopczynski and To 2010] can be adopted to check whether this pushdown automaton accepts a string with the same number of 0's and 1's.

THEOREM 5.4 (CHECKING EQUIVALENCE). *Given two STTs S_1 and S_2, it can be decided in* NEXPTIME *whether $[\![S]\!] \neq [\![S']\!]$.*

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

(1) for some input u, only one of $[\![S_1]\!](u)$ and $[\![S_2]\!](u)$ is defined,
(2) for some input u, the lengths of $[\![S_1]\!](u)$ and $[\![S_2]\!](u)$ differ, or
(3) for some input u, there exist two symbols a and b, such that $a \neq b$, $[\![S_1]\!](u) = u_1 a u_2$, $[\![S_2]\!](u) = v_1 b v_2$, and u_1 and v_1 have the same length.

The first case can be solved in PTIME using the techniques in [Alur and Madhusudan 2009]. The second case can be reduced to checking an affine relation over pushdown automata, that in [Müller-Olm and Seidl 2004] is proven to be PTIME. 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 [Müller-Olm and Seidl 2004]. 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 in [Müller-Olm and Seidl 2004] 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 $S_1 = (Q, q_0, P, X, \eta, F, \delta, \rho)$, and a symbol a, we construct a nondeterministic visibly pushdown transducer (VPT) V_1 (a visibly pushdown automaton with output), from Σ to $\{0\}$ such that 0^n is produced by V_1 on input u, if $[\![S_1]\!](u) = u_1 a u_2$ and $|u_1| = n$.

First, we modify S_1, so that we do not have to worry about the output function. We add to S_1 a new variable x_f, and a new state q_f, obtaining a new STT S_1'. We can then add a special end-of-input symbol $\#$, such that, for every state $q \in Q$ for which $F(q)$ is defined, S_1' goes from q to q_f on input $\#$, and updates x_f to $F(q)$. This new STT S_1' has the following property: for every input u, $[\![S_1]\!](w) = [\![S_1']\!](w\#)$.

Each state of V_1 is a pair (q, f), where q is a state of S_1', and f is a partition of the variables X of S_1' into 6 categories $\{l, m1, m?, m2, r, n\}$, such that a variable x is in the set:

l: if x contributes to the final output occurring on the left of a symbol a where a is the symbol we have guessed the two transducers differ in the final output,
m1: if x contributes to the final output and the symbol a appears in this variable on the left of the ?,
m?: if x contributes to the final output and the symbol a appears in this variable in the ? (a future substitution will add a to the ?),
m2: if x contributes to the final output and the symbol a appears in this variable on the right of the ?,
r: if x contributes to the final output occurring on the right of a symbol a, or
n: if x does not contribute to the final output.

At every step, V_1 nondeterministically chooses which of the previous categories each of the variables of S_1 belongs to. In the following, we use f_p to denote a particular partition: for example f_{m1} is the set of variables mapped by f to $m1$. A state (q, f) of V_1 is initial, if $q = q_0$ is the initial state of S_1, and $f_{m1} \cup f_{m2} = \emptyset$. A state (q, f) of V_1 is final, if $q = q_f$, $f_{m1} = \{x_f\}$ (the only variable contributing to the output is x_f), and $f_l \cup f_r \cup f_{m?} \cup f_{m2} = \emptyset$.

We now define the transition relation of V_1. Given a state s, and a symbol b, a transition updates the state to some s', and outputs a sequence in 0^*. We assume V_1 is in state (q, f), and it is processing the input symbol b. Given an expression $\alpha \in E(X, \Sigma)$ (i.e. an assignment right hand side), we use $x \in_a \alpha$ to say that a variable $x \in X$ appears in α.

b is internal: the VPT V_1 goes to state (q', f'), where $\delta_i(q, s) = q'$. For computing f' we have three different possibilities:
(1) the VPT V_1 guesses that in this transition, some variable x is going to contain the position on which the outputs differ. We show the construction with an example. Assume $\rho_i(q, b, x) = \alpha_1 c \alpha_2 ? \alpha_3$, and c is the position on which we guess the output differs. The transition must satisfy the following properties, which ensures that the current labeling is consistent:
— $\forall y \in_a \alpha_1 . y \in f_l$, $\forall y \in_a \alpha_2 \alpha_3 . y \in f_r$, $f'_{m1} = \{x\}$, and $f_m = \emptyset$ (the only variable that contributes in the middle now is x),
— for every $y \neq x$, all the variables appearing in $\rho_i(q, b, y)$, on the left of the ?, belong to the same partition p_l;
— for every $y \neq x$ all the variables appearing in $\rho_i(q, b, y)$, on the right of the ?, belong to the same partition p_r;
— if $p_l = p_r$, then y belongs to f'_{p_l}, and
— if $p_l \neq p_r$, then y belongs to $f'_{m?}$.
If a variable is assigned a constant we nondeterministically choose which category it belongs to in f'. The output of the transition is 0^k, where k is the sum of the number of output symbols in α, and in $\{\rho_i(q, b, y) | y \in f'_l\}$.
(2) the transition is just maintaining the consistency of the partition, and the position on which the output differs has not been guessed yet. Similar to case (1); or
(3) the transition is just maintaining the consistency of the partition, and the position on which the output differs has already been guessed. Similar to case (1).

b is a call: in this case the updates are similar to the internal case. The updated partition is stored on the stack, and a new partition is non-deterministically chosen for the variables that are reset.

b is a return $b'\rangle$: the VPT V_1 uses the partition in the state for the variables in X, and the partition on the stack state for the variables in X_p. We assume (p, f') is the state on top of the stack. The

VPT V_1 steps to (q'', f''), such that $\delta_r(q, p, b) = q'$. The computation of f' is similar to case in which a is internal.

For every transition we also impose the following conditions: the cardinality of $f_{m1} \cup f_{m2}$ is always less or equal than 1, and if a variable x does not appear in the right hand side of any assignment then $x \in f_n$. The first condition ensures that at most one variable contains the position on which the outputs differ.

Then, in the same way as for V_1, given S_2, and a symbol $b \neq a$, we construct a nondeterministic VPT V_2 from Σ to $\{1\}$, such that 1^n is produced by V_2 on input u, if $[\![S_2]\!](u) = u_1 b u_2$ and $|u_1| = n$. Next, we take the product $V = V_1 \times V_2$. After this operation, since 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 $\{0, 1\}^*$. Finally, we check if V accepts a string that contains the same number of 0's and 1's. The existence of such string ensures that there exists an input w on which S_1 outputs a string $\alpha_1 a \beta_1$, and S_2 outputs $\alpha_2 b \beta_2$, such that $|\alpha_1| = |\alpha_2|$. Such a string can be found by constructing the semi-linear set that characterizes the Parikh image of the context-free language of V [Esparza 1997; Seidl et al. 2004]. A solution to such semi-linear set can be found in NP (in the number of states of V) [Kopczynski and To 2010].

The number of states of V is polynomial in the number of states of S_1 and S_2, but exponential in their number of variables. This is due to the partitioning of the variables into the 6 categories. We can then conclude that checking whether two STTs are inequivalent can be solved in NExpTime. □

If the number of variables is bounded, then the size of V is polynomial, and this gives an upper bound of NP. For the transducers that map strings to nested words, that is, for streaming string-to-tree transducers (SSTT), the above construction yields a PSpace bound [Alur and Cerný 2011].

Corollary 5.5 (Equivalence of String-to-Tree Transducers). *Given two SSTTs S and S' that map strings to nested words, the problem of checking whether $[\![S]\!] = [\![S']\!]$ is solvable in* PSpace.

6. DISCUSSION

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.

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) [Raskin and Servais 2009], and multi bottom-up tree transducers (such a transducer computes a bounded number of transformations at each node by combining the transformations of subtrees) [Engelfriet et al. 2008]. 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, and in particular, can compute the transformations such as swap and tag-based sorting.

Regular look-ahead. Finite copying Macro Tree Transducers (MTTs) with regular look-ahead [Engelfriet and Maneth 1999] 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 with "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 [Maneth 2002]. 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 2002] 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, in presence of 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.

From SSTs to STTs. The STT model generalizes our earlier work on streaming string transducers (SST): SST is a copyless STT without a stack [Alur and Cerný 2010; 2011]. 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 approach. 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 [Engelfriet and Hoogeboom 2001], by an SST. The MSO-equivalence proof in this paper 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 [Raskin and Servais 2009]: 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 2.3 we argue that the example conditional swap cannot be expressed by a copyless STT. Proving this result formally is a challenging open problem.

Succinctness. To highlight the differences in how MTTs and STTs compute, we consider two "informal" 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 thus, has inherent parallelism. Thus, it 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 thus, to compute f, one needs the product of the STTs computing f_1 and f_2. This can be generalized to show that MTTs (or top-down tree transducers) can be exponentially more succinct than STTs. If we were to restrict MTT rules so that multiple states processing the same subtree must coincide, then this gap disappears. In the other direction, consider the transformation f' that maps input $u\#v\#a$ to uv if $a = 0$ and vu otherwise. The transduction f' can be easily implemented by an STT using two variables, one of which stores u and one which stores v. The ability of an STT to concatenate variables in any order allows it to output either uv or vu depending on the last symbol. In absence of look-ahead, an MTT for f' must use two parameters, and 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.

Input/output encoding. Most models of tree transducers process ranked trees (exceptions include visibly pushdown transducers [Raskin and Servais 2009] and Macro forest transducers [Perst and Seidl 2004]). 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.

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) [Filiot et al. 2011], which is however less expressive than STT. In particular VPTs cannot guarantee the output to be a well-matched nested word.

Complexity of checking equivalence. The problem of checking functional equivalence of MSO tree transducers is decidable with non-elementary complexity [Engelfriet and Maneth 2006]. Decidability follows for MSO-equivalent models such as MTTs with finite copying, but no complexity bounds have been established. Polynomial-time algorithms for equivalence checking exist for top-down tree transducers (without regular look-ahead) and visibly pushdown transducers [Comon et al. 2002; Raskin and Servais 2009; Engelfriet et al. 2009]. For STTs, we have established an upper bound of NExpTime, while the upper bound for SSTs is PSpace [Alur and Cerný 2011]. 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.

Application to XML processing. 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 [Hosoya 2011] (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. Before we start implementing a tool for XML processing, we want to understand how to integrate data values (that is, tags ranging over a potentially unbounded domain) in our model. A particularly suitable implementation platform for this purpose seems to be the framework of *symbolic automata* and *symbolic transducers* that allows integration of automata-theoretic decision procedures on top of the SMT solver Z3 that allows manipulation of formulas specifying input/output values from a large or unbounded alphabet in a symbolic and succinct manner [Bjorner et al. 2012; D'Antoni et al. 2012].

ACKNOWLEDGMENT

We thank Joost Engelfriet for his valuable feedback: not only he helped us navigate the extensive literature on tree transducers, but also provided detailed comments, including spotting bugs in proofs, on an earlier draft of this paper.

REFERENCES

R. Alur and P. Cerný. 2010. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (LIPIcs 8)*. 1–12.

R. Alur and P. Cerný. 2011. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of 38th ACM Symposium on POPL*. 599–610.

R. Alur and P. Madhusudan. 2009. Adding Nesting Structure to Words. *Journal of the ACM* 56, 3 (2009).

N. Bjorner, P. Hooimeijer, B. Livshits, P. Molner, and M. Veanes. 2012. Symbolic finite state transducers, algorithms, and applications. In *Proc. 39th ACM Symposium on POPL*.

H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. 2002. Tree automata techniques and applications. (2002). Draft, Available at http://www.grappa.univ-lille3.fr/tata/.

B. Courcelle. 1994. Monadic Second-Order Definable Graph Transductions: A Survey. *Theor. Comput. Sci.* 126, 1 (1994), 53–75.

L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. 2012. *FAST: A Transducer-Based Language for Tree Manipulation.* Technical Report MSR-TR-2012-123. Microsoft Research.

J. Engelfriet. 1975. Bottom-up and top-down tree transformations? a comparison. *Mathematical systems theory* 9, 2 (1975), 198–231. DOI:http://dx.doi.org/10.1007/BF01704020

J. Engelfriet and H.J. Hoogeboom. 2001. MSO definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Log.* 2, 2 (2001), 216–254.

J. Engelfriet, E. Lilin, and A. Maletti. 2008. Extended Multi Bottom-Up Tree Transducers. In *Developments in Language Theory (LNCS 5257)*. 289–300.

J. Engelfriet and S. Maneth. 1999. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation* 154 (1999), 34–91.

J. Engelfriet and S. Maneth. 2006. The equivalence problem for deterministic MSO tree transducers is decidable. *Inf. Process. Lett.* 100, 5 (2006), 206–212.

J. Engelfriet, S. Maneth, and H. Seidl. 2009. Deciding equivalence of top-down XML transformations in polynomial time. *J. Comput. Syst. Sci.* 75, 5 (2009), 271–286.

J. Engelfriet and H. Vogler. 1985. Macro tree transducers. *J. Comput. System Sci.* 31 (1985), 71–146.

J. Esparza. 1997. Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes. *Fundam. Inform.* 31, 1 (1997), 13–25.

E. Filiot, O. Gauwin, P. Reynier, and F. Servais. 2011. Streamability of Nested Word Transductions.. In *FSTTCS (LIPIcs)*, Supratik Chakraborty and Amit Kumar (Eds.), Vol. 13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 312–324. http://dblp.uni-trier.de/db/conf/fsttcs/fsttcs2011.html#FiliotGRS11

H. Hosoya. 2011. *Foundations of XML Processing: The Tree-Automata Approach.* Cambridge University Press.

H. Hosoya and B. C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.* 3, 2 (2003), 117–148.

E. Kopczynski and A.W. To. 2010. Parikh Images of Grammars: Complexity and Applications. In *Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on*. 80–89. DOI:http://dx.doi.org/10.1109/LICS.2010.21

P. Madhusudan and M. Viswanathan. 2009. Query Automata for Nested Words. In *Mathematical Foundations of Computer Science 2009, 34th International Symposium (LNCS 5734)*. 561–573.

S. Maneth. 2002. The Complexity of Compositions of Deterministic Tree Transducers. In *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference (LNCS 2556)*. 265–276.

W. Martens and F. Neven. 2005. On the complexity of typechecking top-down XML transformations. *Theor. Comput. Sci.* 336, 1 (2005), 153–180.

T. Milo, D. Suciu, and V. Vianu. 2000. Typechecking for XML Transformers. In *Proceedings of the 19th ACM Symposium on PODS*. 11–22.

M. Müller-Olm and H. Seidl. 2004. Precise interprocedural analysis through linear algebra. *SIGPLAN Not.* 39 (2004), 330–341. Issue 1.

F. Neven and T. Schwentick. 2002. Query automata over finite trees. *Theor. Comput. Sci.* 275, 1-2 (2002), 633–674.

T. Perst and H. Seidl. 2004. Macro forest transducers. *Inf. Process. Lett.* 89, 3 (2004), 141–149.

J. Raskin and F. Servais. 2009. Visibly pushdown transducers. In *Automata, Languages and Programming: Proceedings of the 35th ICALP (LNCS 5126)*. 386–397.

L. Segoufin and V. Vianu. 2002. Validating streaming XML documents. In *Proceedings of the 21st ACM Symposium on PODS*. 53–64.

H. Seidl, T. Schwentick, A. Muscholl, and P. Habermehl. 2004. Counting in Trees for Free. In *Automata, Languages and Programming: 31st International Colloquium (LNCS 3142)*. 1136–1149.