Pebble Scope and the Power of Pebble Tree Transducers

Master's Thesis of

Bart Samwel

LIACS, Leiden University

Thesis advisor: Joost Engelfriet

Second Reader: Hendrik Jan Hoogeboom

I do not know what I seem to the world, but to myself I appear to have been like a boy playing upon the seashore and diverting myself and then finding a smoother pebble or prettier shell than ordinary, while the great ocean of truth lay before me all undiscovered.

– *Sir Isaac Newton*

Men stumble over pebbles, never over mountains.

– *Emilie Cady*

Abstract

The *pebble tree transducer* (ptt) is a formal model of tree transformation, introduced by Milo et al. **[17]** as a model of the capabilities of XML transformation languages such as XSLT and XQuery. An n-ptt is essentially a tree-walking tree transducer (twtt) that can remember up to n locations by marking them using one of n pebbles, which it must drop and lift in a nested order, i.e., as if they were placed upon a stack. This thesis provides three contributions.

Firstly, this thesis provides an improvement upon the decomposition result of Engelfriet and Maneth **[13]**, which states that an n-ptt can be decomposed into $n + 1$ ptts without pebbles (i.e., twtts). We introduce the concept of the *local pebble*, defined as a pebble whose location is not queried except when the pebble is at the top of the "pebble stack". We then show that if an n-ptt has m local pebbles, then the ptt can be decomposed into $n - m + 1$ twtts. Furthermore, if the original ptt is deterministic, then all of the twtts in the decomposition are deterministic; if the original ptt is nondeterministic, then the last twtt in the decomposition is nondeterministic, while the other $n - m$ twtts are deterministic.

The second contribution has to do with the recent proof by Bojanczyk et al. **[5]** that the pebble tree transducer is not capable of recognizing all MSO-definable input patterns. In this thesis, we define and investigate an extension to the basic tree-transducer model with the capability to evaluate MSO predicates in the left-hand sides of rules. The improved decomposition result still holds for ptts that use MSO predicates.

Thirdly, this thesis provides an in-depth analysis of how a ptt can model the various features of XSLT and XQuery. To date, there has been no proper verification of the claim that ptts are a good model for XML transformation languages. In this thesis, we perform this verification by describing implementation techniques for a large subset of the features of XSLT and XQuery. We show that in practice, local pebbles are a common occurrence, and that the implementation techniques enabled by the use of MSO logic increase the number of local pebbles significantly. Surprisingly, the fact that a ptt has a bounded number of pebbles turns out to be a major obstacle for modeling XSLT fully, which is inherently recursive in nature. In contrast, XQuery can be modeled to a much larger extent.

Contents

Acknowledgements

First of all, I owe a large amount of gratitude to my thesis advisor Joost Engelfriet, who has guided me, encouraged me, taught me how to think logically, and who has shown me the error of my ways on more occasions than I dare to remember. Above all, I thank him for his *patience*.

This thesis would not have been possible without my wife Mieke, who has not only supported me throughout the entire process, but who has also lovingly donated her time by taking care of more than her fair share of household chores.

A big thanks should go to my parents, without whom there would be neither a me nor a Master's thesis of my making.

The remaining kudos should be divided among the following people: my secondary thesis advisor, Hendrik Jan Hoogeboom; my parents-in-law Jan and Jos Ketelaars, for their encouragement; Jeroen Vermeulen, for introducing me to theoretical computer science; Jeroen Kruis, for listening to my complaints; and my colleagues at Centric, who had to live with my thesis-related exhaustion.

CHAPTER 1

Introduction

1.1. Pebble Tree Transducers

Tree transducers are tree automata that process an input tree and produce an output tree as a result. The concept of tree transducers has been around for a long time (see, for example, **[8, 15]**). Historically, they have been useful as a formal model of compilers, among other things. In recent years, tree transducers are also being used as a formal model for transformation and query languages for XML documents **[23]**, such as XSLT **[16]** and XQuery **[22]**. XML documents can be represented as trees, and therefore an XML transformation can be modeled as a transformation of one tree into another. To model the transformations that can be described by XML queries, Milo et al. **[17]** introduced a new kind of tree transducer, the *pebble tree transducer* (ptt). The ptt is an extension of the *tree-walking tree transducer* (twtt), which is the transducer version of the tree-walking automaton **[3]**, and which is closely related to the attribute grammar (cf. Section 3.2 of **[13]**). The extension is analogous to the one that Engelfriet and Hoogeboom **[11]** applied to the tree-walking automaton to create the *pebble tree-walking automaton* (pta). To the tree-walking tree transducer model, the pebble tree transducer adds so-called *pebbles*, markers for remembering locations in the tree that are similar to those used in marker automata **[4]**, but that are subject to an additional nesting or "stack order" requirement. The pebbles are used as follows. In addition to the tree-walking and output actions that one expects from a tree-walking tree transducer, the ptt provides the facility to either *drop* or *lift* a pebble at the current tree node. Two restrictions are imposed on the use of pebbles: firstly, the number of pebbles is bounded, and secondly, pebbles are "stacked", which means that pebbles are always lifted in the reverse order in which they were dropped. In addition to the usual tree-walking tests, the ptt adds the ability to check for the presence or absence of pebbles at the current tree node.

1.2. XML Type Checking

Milo et al. **[17]** introduced the pebble tree transducer to solve a specific problem: type checking of XML transformations. In XML, every document may specify that it is of a certain "type", which means that it satisfies the constraints posed by a Document Type Definition (DTD). The type checking of a transformation consists of checking

whether the transformation, when given an input document conforming to a specific type, always produces output that conforms to a specific output type.

Milo et al. have shown that the type checking problem is decidable for pebble tree transducers when document types are modeled by regular tree languages – a stronger model than XML's DTDs. Their type checking method is based on the principle of inverse type inference, i.e., inferring the input document type from the output document type. The time complexity of the resulting algorithm is hyperexponential in the number of pebbles, but the precise height of the tower of exponentials is not specified in **[17]**. An alternative algorithm was given by Engelfriet and Maneth **[13]**, who showed that a pebble tree transducer using n pebbles can be decomposed into $n+1$ pebble tree transducers using 0 pebbles (which are really just twtts), thus reducing the type checking algorithm to $n+1$ inverse type inferences of twtts. Since the inverse type inference for twtts takes exponential time **[10]**, this leads to an $(n+2)$-fold exponential time algorithm (i.e., a tower of height $n+2$).[1] This does assume that the decomposition process has subexponential time complexity, which is in fact the case.

1.3. Extending the Pebble Tree Transducer

1.3.1. Introduction. In this thesis, we introduce several extensions to the pebble tree transducer model, which improve the type checking complexity and which will make it a more useful tool. The extensions include the concept of *pebble scope*, a concept comparable to variable scope in programs, and the use of *MSO logic* in rules in order to simplify pattern matching. The former addition serves to improve type checking complexity, while the latter addition is intended to improve the correspondence between the pebble tree transducer and the XML transformation languages that it is intended to model. In addition, for technical reasons, we have adopted a different tree model. The earlier ptt models are defined using a tree model based on terms over ranked alphabets, while we will be using a tree model based on graphs. We will now discuss each of these extensions in more detail.

1.3.2. Pebble Scope. A pebble tree transducer can be viewed as a computer program. Pebbles can only be dropped and lifted according to a stack discipline, which means that the dropping and lifting of a given pebble always fully encloses the dropping and lifting of any pebbles higher on the stack. Thus, a dropping and subsequent lifting of a pebble can be viewed as an invocation of a subroutine, which at the beginning drops a pebble to remember its starting location, and which lifts the pebble when it is done. The pebble that remembers the subroutine's starting location can then be regarded as a local variable of the subroutine. But in pebble tree transducers, the

[1]The extra exponential is needed to construct the complement of the output type.

pebbles can be accessed not only by the subroutine that dropped it, but also by any subroutine *called by* such a subroutine – e.g. the subroutine that dropped the fifth pebble on the stack can also see the location of the pebbles numbered one to four. So, pebbles in pebble tree transducers are not really local variables: they are actually globals.

We are interested to see the effects of reducing the scope of pebbles, and therefore we introduce the concept of *local pebbles*. Local pebbles can be detected *only* by the subroutine that dropped them, or, in pebble tree transducer terms, they can be detected only when they are located at the top of the pebble stack. Interestingly, it turns out that global pebble scope is not really needed all the time: often, ptt transformations that require a given number of pebbles turn out to already use some local pebbles, or they can be adapted to use the same number of pebbles but a smaller number of global pebbles.

As we will show in Chapter 4, pebble scope influences the power of pebble tree transducers: pebble tree transducers that use local pebbles are *weaker* than pebble tree transducers that do not use them. As we mentioned earlier, Engelfriet and Maneth **[13]** showed that any pebble tree transducer that uses n pebbles (an n-*ptt*[2]) can be decomposed into $n + 1$ twtts. They also proved that this is the smallest possible decomposition that works for all ptts in general.[3] In Chapter 4, we will show that if an n-ptt has m local pebbles, it can be decomposed into $n - m + 1$ twtts: only the global pebbles count for the number of twtts in the decomposition. Like Engelfriet and Maneth's algorithm, the new decomposition algorithm has subexponential time complexity.[4] This has obvious implications for the complexity of type checking pebble tree transducers using the algorithm described by Engelfriet and Maneth: if there are $n - m + 1$ twtts in the decomposition, then the time complexity of the type checking operation is a tower of $n-m+2$ exponentials. For $m > 0$, this is an improvement upon the time complexity of Engelfriet and Maneth's earlier algorithm, which is a tower of $n + 2$ exponentials.

1.3.3. Pattern Matching using MSO Logic. In general, pattern matching can be described as a boolean test on an input tree in which a bounded number of nodes are marked; if the test result is positive, then the marked nodes constitute a "match".

[2]Milo et al. [**17**], who originally proposed the pebble tree transducer model, count the automaton's reading head position in the tree as a pebble as well, whereas Engelfriet and Maneth [**13**] do not. In this thesis, we will count pebbles like Engelfriet and Maneth: an n-ptt means a ptt which can *drop* n pebbles at most. The differences between the various pebble tree transducer models are discussed in more detail in Section 3.3.

[3]To be precise, they proved this for deterministic ptts only. Whether nondeterministic ptts can be decomposed into less twtts remains an open problem.

[4]A detailed proof of this assertion is beyond the scope of this thesis and will not be provided.

Almost all XML transformation languages support some sort of pattern matching facility, so pattern matching support is crucial for any theoretical model that purports to be a model for XML transformation. Indeed, Milo et al. [**17**] use pattern matching as their prime example of the utility of pebble tree transducers. The definition of pattern matching used by Milo et al. is relatively weak: if we compare it to the formalisms discussed in the survey by Neven and Schwentick [**19**], most of which are equivalent to monadic second-order logic (MSO), we find that the strength of Milo et al.'s pattern matching mechanism is somewhat comparable to, but strictly weaker than that of FOREG, which in turn is strictly weaker than MSO. It has recently been shown that there is a good reason for this: pebble tree-walking automata cannot recognize all regular tree languages [**5**], which implies that pebble tree transducers cannot evaluate all MSO-definable patterns.

In the current thesis, we will investigate an extension to pebble tree transducers that allows them to use the full power of MSO logic. In our model, the left hand side of a rule consists only of a state and an MSO predicate ψ over the domain of nodes in the input tree. The MSO predicate has atomic formulas at its disposal to check whether a node is the node that the reading head currently points at, whether a pebble with a certain number has been placed on a node, whether a node has a certain label, whether an *edge* has a certain label (more on this subject in the next section), and whether two nodes are, in fact, the same node. A single MSO predicate is powerful enough to perform analysis of the relationship between any number of nodes, and includes the ability to test for the presence of a node having certain relationships with all of the other nodes *without* remembering its location and *without* requiring that a pebble be placed on the node. As a result of this ability, the number of pebbles (and especially the number of *global* pebbles) needed to implement XML queries is reduced significantly, as we will show in Chapter 5.

Unfortunately, even though MSO logic is a well-understood tool, it is also known that efficient evaluation procedures for MSO logic formulas (i.e., evaluation procedures with less than hyperexponential time complexity) do not exist. However, every MSO predicate can be represented by a bottom-up tree automaton, which *can* be evaluated in linear time. This makes the use of MSO logic feasible for usage in pebble tree transducers.[5]

It should be noted that since rules with MSO predicates provide a strict superset of the previous models' detection capabilities, not all previous results on ptts may transfer to ptts with MSO logic. We have reason to believe, however, that most results on ptts

[5]Of course, the overall complexity *does* increase, because rule conditions are evaluated in linear instead of constant time. Furthermore, we should mention that our earlier unproven assertion that our decomposition algorithm has subexponential time complexity is predicated on the fact that MSO logic formulas can be represented in this way.

transfer to our ptt model. In Chapter 4 we will show that Engelfriet and Maneth's decomposition [**13**] also applies to our MSO-extended ptt model. Our decomposition algorithm decomposes an n-ptt that uses MSO extensions into $n - m + 1$ twtts (for some m), of which only the *last* twtt actually uses the extra power provided by the MSO extensions: the others are equivalent to (and easily converted into) regular twtts. For regular twtts, inverse type inference is possible in exponential time, while for MSO-extended twtts, inverse type inference, the cornerstone of the type checking algorithm, is possible in double exponential time [**9**]. Given the fact that exactly one of the twtts generated by our decomposition algorithm uses the MSO extensions, the use of MSO increases the time complexity of type checking by one extra exponential, yielding a tower of $n-m+3$ exponentials instead of $n-m+2$. However, by using MSO one can often implement algorithms using less global pebbles, which implies that $n - m$ can often be smaller when a ptt's algorithm makes good use of MSO extensions. This can tip the scale in favour of MSO-extended ptts.

1.3.4. Graph-Based Tree Model. Both the Engelfriet and Maneth [**13**] decomposition and our extended version of that decomposition require the "uprooting" of trees, i.e., reorganising the tree so that a different node is at the top. Unfortunately, the term-based tree model used by Engelfriet and Maneth does not represent this operation very well: an uprooted tree is completely different from the original tree, and the process that is used to construct the uprooted tree is rather complicated. For that reason, we use a different tree model: we represent trees as undirected rooted acyclic connected graphs. This makes the uprooting operation extremely intuitive: it simply consists of selecting a different root. Our tree model differs in some other aspects as well: we use labeled edges, and no fixed relationship between a node label and the labels of the incident edges. An edge can have a different label in either direction, which is similar to the fact that in a term-based tree, the edge number going from a node to its parent is unlabeled while the same edge in reverse direction would be labeled with a positive integer. The only restriction on the labeling of the tree is that all outgoing edge labels of a node must be unique.

1.4. XML Transformation with Pebble Tree Transducers

Milo et al. [**17**] assert that pebble tree transducers can be used to model the capabilities of XML transformation languages. In their article, they include a small example of how a pebble tree transducer would be able to perform a pattern matching operation. However, this is the full extent to which their assertion has been verified: to date, an actual analysis of the techniques by which pebble tree transducers can model transformation languages such as XSL Transformations (XSLT) [**16**] and XQuery [**22**] is lacking. In Chapter 5, we perform such an analysis, considering implementations

using both MSO-less and MSO-extended ptt models. Our analysis shows that pebble tree transducers are able to model a large number of constructs that are available in XSLT and XQuery, regardless of whether the MSO extensions are used. However, the analysis also reveals some unanticipated limits. In particular, it turns out that the recursive nature of XSLT is problematic for pebble tree transducers, because XSLT's recursion depth is unbounded, while the number of pebbles in a pebble tree transducer is bounded.

Throughout Chapter 5, when we describe implementations for XSLT and XQuery constructs, we consider the implications of these implementations for the number of global pebbles in the resulting pebble tree transducers. Our analysis shows that in practice, using the implementation techniques that we describe, it is not uncommon for pebbles to be local. In addition, we find that when the MSO-extended ptt model is used, the MSO-based rules not only help to reduce the total number of pebbles in general, but they generally allow for a reduction in the number of *global* pebbles as well.

1.5. Outline

The remainder of this thesis is built up as follows. Chapter 2 lays out the concepts, definitions and notations that are used in this thesis. Chapter 3 defines our pebble tree transducer model. Chapter 4 reproduces Engelfriet and Maneth's decomposition result for our pebble tree transducer model, and shows that pebble tree transducers with n pebbles of which m are local can be decomposed into $n - m + 1$ twtts. Chapter 5 analyzes how we can use pebble tree transducers to implement the features of XSLT and XQuery, and how the described implementation techniques influence the number of global pebbles that are required to execute a transformation. Chapter 6 discusses the conclusions that can be drawn from the research presented in this thesis.

CHAPTER 2

Preliminaries

2.1. Introduction

This chapter contains the mathematical definitions that are going to be used throughout this thesis but that are not new material. The basics may be skipped by readers who are already familiar with commonly used mathematical notations. The other sections cannot be skipped.

2.2. Basics

Natural numbers. The set of natural numbers $\{0, 1, \dots\}$ is denoted by $\mathbb{N}$. For $k, l \in \mathbb{N}$, $[k, l]$ denotes the set $\{i \in \mathbb{N} \mid k \leq i \leq l\}$.

Sets. The empty set is denoted by $\emptyset$. For a set A, the cardinality of A is denoted by $|A|$, $\mathcal{P}(A)$ is the powerset (i.e., the set of all possible subsets) of A. Set membership of an element a in a set A is denoted by $a \in A$, the union of two sets A, B is denoted by $A \cup B$, the difference by $A - B$ and the intersection by $A \cap B$. For sets A and B, we define the *disjoint-sets-union*, written $A \cup_{\not\emptyset} B$, as $A \cup B$ when $A \cap B = \emptyset$ and undefined otherwise. (This operator allows us to express partitioning of a set in a single statement; e.g., $A = X \cup_{\not\emptyset} Y \cup_{\not\emptyset} Z$ expresses the fact that A can be split into blocks X, Y and Z.)

Alphabets, strings and languages. An *alphabet* is a finite set. For an alphabet A, a string w over A is written as $a_1 a_2 \cdots a_m$ with $a_i \in A$ (for $i \in [1, m]$), where $m \geq 0$ is called the *length of* w, denoted $|w|$. In a string $w = a_1 a_2 \cdots a_m$, the symbol a_i (the ith symbol) is denoted by $w(i)$. The empty string over any alphabet is denoted by λ. For any string $w = a_1 a_2 \cdots a_m$, we define $\text{left}(w, l) = a_1 a_2 \cdots a_l$ (for $0 \leq l \leq m$) and $\text{left}(w, l) = w$ (for $l > m$). We use A^* to denote the set of all strings over A. For $n \in \mathbb{N}$, $A^{\leq n} = \{w \in A^* \mid |w| \leq n\}$ is the set of all strings over A with length at most n, $A^{<n} = A^{\leq n-1}$ is the set of all strings over A with length less than n, and $A^n = \{w \in A^* \mid |w| = n\}$ is the set of all strings over A with length exactly equal to n. For an alphabet A and two strings $v, w \in A^*$, where $v = v_1 v_2 \cdots v_{|v|}$ and $w = w_1 w_2 \cdots w_{|w|}$, the *concatenation of* v *and* w, denoted $v \cdot w$ or vw, is defined as $vw = v_1 v_2 \cdots v_{|v|} w_1 w_2 \cdots w_{|w|}$.

Cartesian product. For n sets $A_1, A_2, \dots, A_n$, the Cartesian product is defined as $A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \dots, a_n) \mid a_i \in A_i \ (i \in [1, n])\}$. As a notational convenience,

we also define $\langle A_1, A_2, \ldots, A_n \rangle = \{\langle a_1, a_2, \ldots, a_n \rangle \mid a_i \in A_i \ (i \in [1, n])\}$. An element $(a_1, a_2, \ldots, a_n)$ of a Cartesian product of n sets is called an n-tuple; 2-tuples are called pairs.

Relations. A set $R \subseteq A \times B$ is called a *(binary) relation over A and B*. For a relation R over A and B, and a set $A' \subseteq A$, $R(A') = \{y \mid x \in A' \text{ and } (x, y) \in R\}$; note that $R(A') \subseteq B$. For $R \subseteq A \times B$, $R^{-1} = \{(y, x) \mid (x, y) \in R\} \subseteq B \times A$ is the *inverse* of relation R. The composition of two binary relations $R \subseteq A \times B$ and $S \subseteq B \times C$, denoted by $R \circ S$ or RS, is a relation over A and C defined as $\{(x, z) \mid (x, y) \in R \text{ and } (y, z) \in S \text{ for some } y \in B\}$. The n-fold composition of a binary relation $R \subseteq A \times A$ with itself is denoted as R^n, where $n \geq 1$; R^0 denotes the identity relation $I_A = \{(x, x) \mid x \in A\}$. The reflexive transitive closure of a binary relation $R \subseteq A \times A$ is $R^* = R^0 \cup R^1 \cup R^2 \cup \ldots$.

Functions. A *function f from A to B* (denoted $f : A \to B$) is a relation over A and B in which, for any $a \in A$, $f(\{a\})$ has at most one element. If $f(\{a\})$ has exactly one element, we denote this element as $f(a)$. If $f(\{a\})$ is empty, then $f(a)$ is undefined. A function f is *total* if $f(a)$ is defined for all $a \in A$, it is *injective* if for every $a_1, a_2 \in A$ for which $f(a_1)$ and $f(a_2)$ are defined, if $a_1 \neq a_2$ then $f(a_1) \neq f(a_2)$, and it is *surjective* if $f(A) = B$. If a function is both surjective and injective, it is called *bijective*.

Computation relations. For a binary relation $\Rightarrow$ over A and A, relation $\Rightarrow^*$ is called the *computation relation of* $\Rightarrow$. A sequence $a_1, a_2, \ldots, a_n$ with $a_i \in A$ (for $i \in [1, n]$) and $a_i \Rightarrow a_{i+1}$ (for $i \in [1, n-1]$) is called a *finite computation of length n by $\Rightarrow$ starting with a_1*. A sequence $a_1, a_2, \ldots$ with $a_i \in A$ and $a_i \Rightarrow a_{i+1}$ (for all $i \geq 1$) is called an *infinite computation by $\Rightarrow$ starting with a_1*. A computation is *complete* if it is either infinite or if there is no $a_{n+1} \in A$ such that $a_n \Rightarrow a_{n+1}$. In the latter case, a_n is called the *result* of the computation.

2.3. Graphs and Trees

2.3.1. Graphs. A *graph* is a finite, nonempty, undirected graph with labeled nodes and edges. The undirected edges $\{u, v\}$ of a graph are represented as a pair of directed edges (u, v) and (v, u). Edge labels are directed, and the labels of the outgoing edges of a node must be distinct. Formally, a *graph* is a pair $g = (\nu_g, \epsilon_g)$ consisting of a finite total node labeling function $\nu_g : V_g \to \Sigma_g$ and a finite total edge labeling function $\epsilon_g : E_g \to \Phi_g$, with $\Sigma_g = \nu_g(V_g)$, $\Phi_g = \epsilon_g(E_g)$ (in other words, ν_g and ϵ_g are surjective) and $E_g \subseteq V_g \times V_g$, having $(u, v) \in E_g$ if and only if $(v, u) \in E_g$, and for all $(u, v_1), (u, v_2) \in E_g$, $v_1 \neq v_2$ implies $\epsilon_g(u, v_1) \neq \epsilon_g(u, v_2)$. A graph g is a *graph over Σ and Φ* for all Σ, Φ for which $\Sigma \supseteq \Sigma_g$ and $\Phi \supseteq \Phi_g$. The set of all graphs

over Σ and Φ is written $G_{\Sigma,\Phi}$. For a graph g over Σ and Φ and a subset $\Sigma' \subseteq \Sigma$, $V_{g,\Sigma'} = \{u \mid u \in V_g \text{ and } \nu_g(u) \in \Sigma'\}$ is the set of all nodes in g that are labeled in Σ'.

For a graph g, V_g is the set of *nodes* (or *vertices*) of g, Σ_g is the node label alphabet of g, Φ_g is the edge label alphabet of g, and E_g is the set of *edges* of g. For notational convenience, if a graph has a name with a subscript, e.g. g_1, then we will write V_1 etc. instead of V_{g_1}. A graph g is a *supergraph* of another graph g', written $g \supseteq g'$, when $\nu_g \supseteq \nu_{g'}$ and $\epsilon_g \supseteq \epsilon_{g'}$. The *subgraph* relation ($\subseteq$) is defined analogously. For a graph g and a set $V \subseteq V_g$, the *V-induced subgraph* of g is defined as the graph $g_V = (\nu_V, \epsilon_V)$, where $\nu_V(w) = \nu_g(w)$ for all $w \in V$, $E_V = \{(u,w) \mid (u,w) \in E_g \text{ and } u,w \in V\}$ and $\epsilon_V(u,w) = \epsilon_g(u,w)$ for all $(u,w) \in E_V$. Graph g' is an induced subgraph of g if $V_{g'} \subseteq V_g$ and g' is the $V_{g'}$-induced subgraph of g.

For a node $u \in V_g$, $\text{out}_g(u) = \{\epsilon_g(u,v) \mid (u,v) \in E_g\}$ is the set of labels of outgoing edges of u. For $u \in V_g$ and $\phi \in \text{out}_g(u)$, $\phi_{\langle g \rangle}(u)$ denotes the unique $v \in V_g$ such that $\epsilon_g(u,v) = \phi$. If the intended graph g is clear from the context, we will simply write $\phi(u)$. A *path* is a sequence $v_1 \cdots v_n$, $n \geq 1$, of distinct nodes $v_i \in V_g$ such that $(v_i, v_{i+1}) \in E_g$ for all $i \in [1, n-1]$. A *cycle* is a path $v_1 \cdots v_n$ with $(v_n, v_1) \in E_g$ and $n \neq 2$; its length is n. A graph is *acyclic* if it contains no cycles. A graph g is *connected* if for all pairs of nodes $u, v \in V_g$, there exists a path of which u is the first node and v is the last.

A graph *homomorphism* from graph g_1 to g_2 is a total function $h : V_1 \to V_2$ such that (1) for all $u \in V_1$, $\nu_2(h(u)) = \nu_1(u)$, and (2) for all $u, v \in V_1$, if $(u,v) \in E_1$, then $(h(u), h(v)) \in E_2$ and $\epsilon_2(h(u), h(v)) = \epsilon_1(u,v)$. A graph *isomorphism* is a bijection $h : V_1 \to V_2$ such that both h and h^{-1} are graph homomorphisms. When there exists an isomorphism from a graph g_1 to a graph g_2, the graphs are *isomorphic*, which is written $g_1 \cong g_2$. An *edge* homomorphism is like a homomorphism, except that it does not have to satisfy condition (1). The existence of an edge homomorphism from a graph g_1 to a graph g_2 is written $g_1 \cong_{\text{edge}} g_2$, and the graphs are called *edge isomorphic*. For a homomorphism h and nodes $u \in V_1$ and $v \in V_2$ such that $h(u) = v$, we say that node v *corresponds to* node u by homomorphism h. As a naming convention for homomorphisms and isomorphisms, we will use $h_{g_1 \to g_2}$ for an (edge) homo/isomorphism from a graph g_1 to a graph g_2. We will leave out the source graph g_1 if g_2 is a mathematical expression that uses graph g_1 as its only graph input. For instance, with node relabeling (defined below), $g_2 = g_1[s]$ for a graph g_1 and some function s, and we simply write $h_{g_1[s]}$ to indicate the edge isomorphism from g_1 to $g_1[s]$.

Informally, the *gluing* of a graph g_2 at a node $u_2 \in V_2$ onto a graph g_1 at a node $u_1 \in V_1$ is obtained by taking the union of copies of the two graphs, where u_1 and u_2 are merged into a single node having the label of u_2. For this operation to be defined, u_1 and u_2 must have disjoint outgoing edge labels, because if they do

not, the result will not be a graph: it will have duplicate outgoing edge labels on the node that merges u_1 and u_2. For the formal definition of gluing we require the following definition. Let V be the set consisting of all nodes in all graphs. Then $h_{u_1,u_2,1} : V \to V$ and $h_{u_1,u_2,2} : V \to V$ are total injections, uniquely determined by $u_1 \in V$ and $u_2 \in V$, that satisfy for any pair of graphs g_1, g_2 with $u_1 \in V_1$ and $u_2 \in V_2$, $h_{u_1,u_2,1}(V_1) \cap h_{u_1,u_2,2}(V_2) = \{h_{u_1,u_2,1}(u_1)\} = \{h_{u_1,u_2,2}(u_2)\}$. In order to show that such $h_{u_1,u_2,1}$ and $h_{u_1,u_2,2}$ exist, we will now give a possible definition for them. Assume that V is countable, and without loss of generality, that $V = \mathbb{N}$. Then we can define $h_{u_1,u_2,1} = \{(u_1, 0)\} \cup \{(u, 2u+1) \mid u \in V \text{ and } u \neq u_1\}$ and $h_{u_1,u_2,2} = \{(u_2, 0)\} \cup \{(u, 2u+2) \mid u \in V \text{ and } u \neq u_2\}$. It is easy to verify that these functions satisfy $h_{u_1,u_2,1}(V_1) \cap h_{u_1,u_2,2}(V_2) = \{h_{u_1,u_2,1}(u_1)\} = \{h_{u_1,u_2,2}(u_2)\} = \{0\}$. Having this definition, we can now continue with the formal definition of gluing. For graphs $g_1, g_2 \in G_{\Sigma,\Phi}$ and nodes $u_1 \in V_1$ and $u_2 \in V_2$ having $\text{out}_1(u_1) \cap \text{out}_2(u_2) = \emptyset$, the *gluing of* (g_2, u_2) *onto* g_1 *at* u_1, written $g_1[u_1 \leftarrow (g_2, u_2)]$, is then defined as follows. Let $g_1', g_2' \in G_{\Sigma,\Phi}$ so that $g_1' \cong g_1$ and $g_2' \cong g_2$ by respective isomorphisms $h_1 \subset h_{u_1,u_2,1}$ and $h_2 \subset h_{u_1,u_2,2}$. We then have $V_1' = h_1(V_1)$ and $V_2' = h_2(V_2)$, and $V_1' \cap V_2' = \{h_1(u_1)\} = \{h_2(u_2)\}$. Now let $g_1'' = (\nu_1'', \epsilon_1'')$ with $V_1'' = V_1'$, $\epsilon_1'' = \epsilon_1'$, $\nu_1''(h_1(u_1)) = \nu_2(u_2)$ and $\nu_1''(u) = \nu_1'(u)$ for all $u \neq h_1(u_1)$. The gluing $g_1[u_1 \leftarrow (g_2, u_2)]$ is the coordinatewise union of g_1'' and g_2'. The overlap between g_1'' and g_2' is only one node $h_1(u_1) = h_2(u_2)$, and as those nodes share their node labels and have disjoint outgoing edge labels, $g_1[u_1 \leftarrow (g_2, u_2)]$ is a graph. As $g_1[u_1 \leftarrow (g_2, u_2)]$ fully contains g_2', the isomorphism h_2 from graph g_2 to g_2' is an injective homomorphism from g_2 to $g_1[u_1 \leftarrow (g_2, u_2)]$. For a gluing $g_1[u_1 \leftarrow (g_2, u_2)]$, we will indicate this homomorphism by $h_{g_2 \to g_1[u_1 \leftarrow (g_2,u_2)]}$. In addition, the isomorphism h_1 from g_1 to g_1' is an injective *edge* isomorphism from g_1 to g_1'', and as $g_1[u_1 \leftarrow (g_2, u_2)]$ contains g_1'', it is an edge *homomorphism* from g_1 to $g_1[u_1 \leftarrow (g_2, u_2)]$. For a gluing $g_1[u_1 \leftarrow (g_2, u_2)]$, we will use $h_{g_1 \to g_1[u_1 \leftarrow (g_2,u_2)]}$ to indicate this homomorphism. Note that gluing is monotonic w.r.t. g_1 and g_2, i.e., if we have supergraphs $g_3 \supseteq g_1$ and $g_4 \supseteq g_2$, then we have $g_3[u_1 \leftarrow (g_4, u_2)] \supseteq g_1[u_1 \leftarrow (g_2, u_2)]$ for any $u_1 \in V_1$ and $u_2 \in V_2$. This is ensured by the fact that $h_{u_1,u_2,1}$ and $h_{u_1,u_2,2}$ are uniquely determined by u_1 and u_2, and that the chosen homomorphisms $h_{g_1 \to g_1[u_1 \leftarrow (g_2,u_2)]}$ and $h_{g_2 \to g_1[u_1 \leftarrow (g_2,u_2)]}$ are always subsets of $h_{u_1,u_2,1}$ and $h_{u_1,u_2,2}$, respectively.

For a graph $g \in G_{\Sigma,\Phi}$ and a total function $s : \Sigma' \to \Delta$ defined on some domain $\Sigma' \subseteq \Sigma$, the *node relabeling* $g[s] \in G_{(\Sigma - \Sigma') \cup \Delta, \Phi}$ is defined as the graph obtained as follows. Let $s' : \Sigma \to (\Sigma - \Sigma') \cup \Delta$ so that $s'(\sigma) = s(\sigma)$ for $\sigma \in \Sigma'$ and $s'(\sigma) = \sigma$ otherwise. Now $g[s] = \left(\nu_{g[s]}, \epsilon_{g[s]}\right)$ with $\epsilon_{g[s]} = \epsilon_g$ and $\nu_{g[s]} = \nu_g s'$. Obviously, $g[s]$ is edge isomorphic with g. Note that, like gluing, node relabeling is monotonic w.r.t. the graph parameter as well: if $g' \supseteq g$, then $g'[s] \supseteq g[s]$. There is a difference though:

for node relabeling, the monotonicity is ensured by the fact that the edge isomorphism from the original graph to the label-substituted graph is an identity function.

2.3.2. Trees. A *tree* is a connected acyclic graph. For label alphabets Σ and Φ, the set of all trees over Σ and Φ is denoted $T_{\Sigma,\Phi}$. Because trees are connected and acyclic, for a tree t and nodes $u, v \in V_t$, there is a unique path from u to v in t, which we denote $\mathrm{path}_t(u, v)$. The gluing of trees obviously yields a tree, as (1) the gluing is connected, because the subgraphs representing the original trees are themselves connected, and every pair of nodes from different trees are connected through the glue node, and (2) the gluing is acyclic: the original trees are acyclic and no edges were added, so any cycle must run across both subgraphs; and a cycle that runs across both subgraphs would have to pass the glue node twice, which is not allowed, as the nodes in a path (and therefore in a cycle) must be distinct.

A *rooted tree* t is a pair $(\mathrm{tree}_t, \mathrm{root}_t)$ where $\mathrm{tree}_t \in T_{\Sigma,\Phi}$ is a tree and $\mathrm{root}_t \in V_{\mathrm{tree}_t}$ is the *root*. For a rooted tree t, "t" normally refers to the pair $t = (\mathrm{tree}_t, \mathrm{root}_t)$, but when it is used in a context that unambiguously requires a tree, it is understood to refer to tree_t. For instance, for a rooted tree t, we will write V_t for V_{tree_t}. A homomorphism h from a rooted tree t to a rooted tree t' is a homomorphism from tree_t to $\mathrm{tree}_{t'}$ having $\mathrm{root}_{t'} = h(\mathrm{root}_t)$, and analogous definitions apply for isomorphism, edge homomorphism and edge isomorphism. The notations $t \cong t'$ and $t \cong_{\mathrm{edge}} t'$ are used to indicate isomorphism and edge isomorphism with rooted trees as they are with graphs. The set of all rooted trees over Σ and Φ is $R_{\Sigma,\Phi} = \{(t, u) \mid t \in T_{\Sigma,\Phi} \text{ and } u \in V_t\}$.

For a rooted tree t, a node $w \in V_t$ is a *leaf* when either $w = \mathrm{root}_t$ and w has no outgoing edges, or $w \neq \mathrm{root}_t$ and the only outgoing edge of w is $(w, \mathrm{path}_t(w, \mathrm{root}_t)\,(2))$. If a node is not a leaf, it is an *internal node*. For a rooted tree t, an edge $(w, w') \in E_t$ is *upward* when $\mathrm{path}_t(w, \mathrm{root}_t) = w \cdot \mathrm{path}_t(w', \mathrm{root}_t)$, and *downward* otherwise. For a node label σ, we use $\mathrm{snt}(\sigma)$ (pronounced: single-node tree) to indicate a rooted tree consisting of a single node having σ as its label.

For a rooted tree $t \in R_{\Sigma,\Phi}$, the node relabeling $t[s]$ is defined as $(\mathrm{tree}_t[s], \mathrm{root}_t)$. For rooted trees $t_1, t_2 \in R_{\Sigma,\Phi}$ and a node $u_1 \in V_1$, the *gluing of* t_2 *onto* t_1 at u_1, written $t_1[u_1 \leftarrow t_2]$ is defined as the rooted tree $\left(\mathrm{tree}_1[u_1 \leftarrow t_2],\ h_{\mathrm{tree}_1 \to \mathrm{tree}_1[u_1 \leftarrow t_2]}(\mathrm{root}_1)\right)$. We define $h_{t_1 \to t_1[u_1 \leftarrow t_2]} = h_{\mathrm{tree}_1 \to \mathrm{tree}_1[u_1 \leftarrow t_2]}$, the edge homomorphism from t_1 to $t_1[u_1 \leftarrow t_2]$. It should be noted that the homomorphism $h_{\mathrm{tree}_2 \to \mathrm{tree}_1[u_1 \leftarrow t_2]}$ is *not* a homomorphism from t_2 to $t_1[u_1 \leftarrow t_2]$, as $h_{\mathrm{tree}_2 \to \mathrm{tree}_1[u_1 \leftarrow t_2]}(\mathrm{root}_2) \neq \mathrm{root}_{t_1[u_1 \leftarrow t_2]}$.

Given alphabets Σ, Φ and Δ, $T_{\Sigma,\Phi}(\Delta)$ denotes the set of all *trees over* Σ *and* Φ *augmented by* Δ. Formally, $T_{\Sigma,\Phi}(\Delta) = T_{\Sigma\cup\Delta,\Phi}$. Note that, in contrast with the customary definition, this definition of augmented trees allows the labels in Δ to appear on internal nodes as well as leaf nodes. Similarly, $R_{\Sigma,\Phi}(\Delta) = R_{\Sigma\cup\Delta,\phi}$ is the set of all

rooted trees over Σ and Φ augmented by Δ. Note that for a tree $t \in T_{\Sigma,\Phi}(\Delta)$, $\Sigma \cap \Delta = \emptyset$ implies $V_t = V_{t,\Sigma} \cup_{\not\emptyset} V_{t,\Delta}$.

2.4. Monadic Second-Order Tree Logic

Let A be a set of *relation symbols*, where all relation symbols $\alpha \in A$ are associated with an arity $\text{arity}(\alpha) \geq 0$ and a set of atomic formulas

$$\text{AF}_\alpha = \left\{ \alpha\left(x_1, \ldots, x_{\text{arity}(\alpha)}\right) \mid x_1, \ldots, x_{\text{arity}(\alpha)} \text{ are node variables} \right\}.$$

A *monadic second-order logic graph predicate* or *MSO predicate* ψ *on* A (an A-MSO predicate, in short) is a logical formula on graphs g, that can use existential and universal quantification over both nodes and sets of nodes from V_g, and that can use atomic formulas in AF_α for all $\alpha \in A$, plus the atomic formula $x \in X$ for any node variable x and node set variable X. We will first provide a formal definition of well-formed MSO predicates, and then we will define the semantics of the formulas.

2.4.1. Well-Formed MSO Predicates. A well-formed A-MSO predicate (or simply A-MSO predicate – well-formedness is normally implied) and its free variables are defined as follows. The free variables of an MSO predicate are a pair $(F_{\text{node}}, F_{\text{set}})$, where F_{node} contains the free node variables, and F_{set} contains the free set variables. Lower-case variable names (such as x) indicate node variables, upper-case variable names (such as X) indicate node set variables.

- For node variables $x_1, x_2, \ldots, x_n$ and a relation symbol $\alpha \in A$, $\alpha(x_1, x_2, \ldots, x_n) \in \text{AF}_\alpha$ is a well-formed MSO predicate having free variables $(\{x_1, \ldots, x_n\}, \emptyset)$. Note that atomic formulas for relation symbols in A only take node variables as arguments, never node set variables.
- If x is a node variable and X is a node set variable, $x \in X$ is a well-formed MSO predicate that has free variables $(\{x\}, \{X\})$.
- If ψ is a well-formed MSO predicate, $(F_{\text{node}}, F_{\text{set}})$ is the set of free variables of ψ, and x and X are variables, then $\neg\psi$, $\forall x : \psi$ and $\forall X : \psi$ are well-formed MSO predicates having free variables $(F_{\text{node}}, F_{\text{set}})$, $(F_{\text{node}} - \{x\}, F_{\text{set}})$ and $(F_{\text{node}}, F_{\text{set}} - \{X\})$, respectively.
- If ψ_1 and ψ_2 are well-formed MSO predicates having free variable sets $(F_{\text{node},1}, F_{\text{set},1})$ and $(F_{\text{node},2}, F_{\text{set},2})$, respectively, then $(\psi_1 \vee \psi_2)$ is a well-formed MSO predicates that has free variables $(F_{\text{node},1} \cup F_{\text{node},2}, F_{\text{set},1} \cup F_{\text{set},2})$.

The set of all well-formed A-MSO predicates is written MSO_A. A *closed* A-MSO predicate is an A-MSO predicate that has free variables $(\emptyset, \emptyset)$. The set of all well-formed closed A-MSO predicates is written CMSO_A. Note that the definition implies that for a set of relation symbols $A' \supseteq A$, any well-formed A-MSO predicate is also a

well-formed A'-MSO predicate. Also note that our definition of MSO predicates does not include any redundant constructs. We will use the following notations, but only as convenient shorthands:

- $(\psi_1 \wedge \psi_2)$ stands for $\neg(\neg\psi_1 \vee \neg\psi_2)$.
- $(x \implies y)$ stands for $(\neg x \vee y)$.
- $\exists x : \psi$ and $\exists X : \psi$ stand for $\neg\forall x : \neg\psi$ and $\neg\forall X : \neg\psi$, respectively.
- $\forall x, y : \psi$ stands for $\forall x : \forall y : \psi$; defined analogously for node set variables, $\exists$ clauses, and more than two variables.
- $\forall x \in X : \psi$ stands for $\forall x : (x \in X \implies \psi)$.
- $\exists x \in X : \psi$ stands for $\exists x : (x \in X \wedge \psi)$.

2.4.2. Binding and Truth Values. For a well-formed A-MSO predicate ψ with free variables $(F_{\text{node}}, F_{\text{set}})$, and a graph g, a *binding of ψ on g* is a triple $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b_A)$, where $b_{F_{\text{node}}} : F'_{\text{node}} \to V_g$ (with $F'_{\text{node}} \supseteq F_{\text{node}}$) is a total variable binding function assigning nodes to ψ's free node variables, $b_{F_{\text{set}}} : F'_{\text{set}} \to \mathcal{P}(V_g)$ (with $F'_{\text{set}} \supseteq F_{\text{set}}$) is a total variable binding function assigning node sets to ψ's free node set variables, and relation binding $b_A : A \to \mathcal{P}\left(V_g^*\right)$ is a total function having $b_A(\alpha) \in \mathcal{P}\left(V_g^{\text{arity}(\alpha)}\right)$ for every $\alpha \in A$, binding an actual relation to every relation symbol in A. Given a binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b_A)$, the *truth value* (or simply *value*) of ψ is a value in $\{\text{true}, \text{false}\}$. For node variable binding $b_{F_{\text{node}}}$, a node variable x, and a node $w \in V_g$, $b_{F_{\text{node}}} \oplus (x, w)$ is defined as the node variable binding $b'_{F_{\text{node}}}$ having $b'_{F_{\text{node}}}(x) = w$ and $b'_{F_{\text{node}}}(x') = b_{F_{\text{node}}}(x')$ for all $x' \neq x$ in the domain of $b_{F_{\text{node}}}$. The $\oplus$ operator is defined analogously for node set variable bindings.

The truth value of an MSO predicate ψ, given a binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b_A)$ on a graph g, is defined recursively as follows:

- If $\psi \equiv \alpha(x_1, \ldots, x_n)$, then ψ is true iff $(b_{F_{\text{node}}}(x_1), \ldots, b_{F_{\text{node}}}(x_n)) \in b_A(\alpha)$.
- if $\psi \equiv x \in X$, then ψ is true iff $b_{F_{\text{node}}}(x) \in b_{F_{\text{set}}}(X)$.
- If $\psi \equiv (\psi_1 \vee \psi_2)$, then ψ is true iff at least one of ψ_1 and ψ_2 is true for binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b_A)$.
- If $\psi \equiv (\psi_1 \wedge \psi_2)$, then ψ is true iff both ψ_1 and ψ_2 are true for binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b_A)$.
- If $\psi \equiv \neg\psi_1$, then ψ is true iff ψ_1 is false for binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b_A)$.
- If $\psi \equiv \forall x : \psi_1$ for node variable x, then ψ is true iff for every $w \in V_g$, ψ_1 is true for binding $(b_{F_{\text{node}}} \oplus (x, w), b_{F_{\text{set}}}, b_A)$.
- If $\psi \equiv \forall X : \psi_1$ for node set variable X, then ψ is true iff for every $V \in \mathcal{P}(V_g)$, ψ_1 is true for binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}} \oplus (X, V), b_A)$.

2.4.3. Common Relation Symbols. These are a number of relation symbols that are commonly made available in MSO predicates on graphs.

- *equality*: eq is a binary relation symbol that tests for node equality. Its relation for a graph g is $b_{\text{eq},g} = \{(x, y) \mid x, y \in V_g \text{ and } x = y\}$. Instead of $\text{eq}(x, y)$ we may also write $x = y$.
- *node label test*: lab_σ is a unary relation symbol that tests the label of a node. Its relation for a graph g is $b_{\text{lab}_\sigma,g} = \{(x) \mid x \in V_g \text{ and } \nu_g(x) = \sigma\}$.
- *edge label test*: edge_ϕ is a binary relation symbol that tests the label of the edge going from the first node to the second node. Its relation for a graph g is $b_{\text{edge}_\phi,g} = \{(x, y) \mid x, y \in V_g \text{ and } \epsilon_g(x, y) = \phi\}$.
- *true/false:* true and false are nullary relation symbols, which indicate an always-true or always-false predicate. The relation for true is $b_{\text{true}} = \{()\}$, where $()$ is a 0-tuple of nodes. The relation for false is $b_{\text{false}} = \emptyset$. The effect of these relations is that the zero-tuple (which is always the argument list of true and false) is in b_{true} but not in b_{false}.

The set $A_{\Sigma,\Phi}$ of these common relation symbols for graphs over Σ and Φ is defined as $A_{\Sigma,\Phi} = \{\text{eq}, \text{true}, \text{false}\} \cup \{\text{lab}_\sigma \mid \sigma \in \Sigma\} \cup \left\{\text{edge}_\phi \mid \phi \in \Phi\right\}$. The relation binding for $A_{\Sigma,\Phi}$ on a graph g is

$$\begin{aligned} b_{A_{\Sigma,\Phi},g} = &\{(\text{eq}, b_{\text{eq},g}), (\text{true}, b_{\text{true}}), (\text{false}, b_{\text{false}})\} \cup \\ &\{(\text{lab}_\sigma, b_{\text{lab}_\sigma,g}) \mid \sigma \in \Sigma\} \cup \\ &\left\{\left(\text{edge}_\phi, b_{\text{edge}_\phi,g}\right) \mid \phi \in \Phi\right\}. \end{aligned}$$

We will use $\text{edge}_{\Phi'}(x, y) \equiv \bigvee_{\phi \in \Phi'} \text{edge}_\phi(x, y)$ to indicate the existence of an edge having any edge label that occurs in a set Φ'.

CHAPTER 3

Pebble Tree Transducers

3.1. Introduction

In this chapter we define an extended pebble tree transducer model based on our graph-based tree model. The extensions include rules with conditions expressed in MSO logic, and configuration instructions on internal nodes. Section 3.2 gives the formal definition of a pebble tree transducer, an introduction into the ptt computation model, and formal definitions of all the concepts involved in the ptt's computation model. Section 3.3 then contrasts our model to some of the models that have been proposed in related work.

3.2. Definition

We will begin by giving a formal definition of the syntax of our pebble tree transducer model. We will then provide an overview of both the syntax and the semantics of a pebble tree transducer, in order to provide the reader with a general idea of the roles played by the various ptt components. After that, we will provide formal definitions of the ptt semantics.

DEFINITION 3.2.1. A *pebble tree transducer* (or *ptt*) is a tuple

$$M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$$

where:

$n \in \mathbb{N}$ is the *number of pebbles*,

(Σ, Φ) is a pair of *input alphabets* (Σ a node label alphabet, Φ an edge label alphabet),

(Δ, Γ) is a pair of *output alphabets* (Δ a node label alphabet, Γ an edge label alphabet),

Q is a finite set of *states*,

$q_0 \in Q$ is the *initial state*, and

R is a finite set of *rules*. In order to fully define rules we require the following definitions (that we will explain in more detail later):

(1) For a tree $t \in T_{\Sigma,\Phi}$, $\mathrm{IC}_{n,t} = V_t \times V_t^{\leq n}$ is the set of *n-pebble input configurations on t*.

(2) A *condition* of M is a closed MSO predicate on relation symbols set $A_M = A_{\Sigma,\Phi} \cup \{\text{head}\} \cup \{\text{peb}_i \mid 1 \leq i \leq n\}$. The relation symbols head and peb_i ($1 \leq i \leq n$) are all unary. The truth value of a condition ψ for an n-pebble input configuration $(u, \pi) \in \text{IC}_{n,t}$ on a tree $t \in T_{\Sigma,\Phi}$ is defined to be the truth value of ψ with binding $\left(\emptyset, \emptyset, b_{A_M,t,(u,\pi)}\right)$ where $b_{A_M,t,(u,\pi)} = b_{A_{\Sigma,\Phi},t} \cup \{(\text{head}, \{u\})\} \cup \{(\text{peb}_i, \{\pi(i)\}) \mid 1 \leq i \leq |\pi|\}$.

(3) For a condition ψ, the set of *input instructions of M on ψ* is defined as

$$\begin{aligned}\text{II}_{M,\psi} \;=\; & \{\text{stay}\} \cup \\ & \left\{\text{go}_\phi \mid \phi \in \Phi \text{ and } M \models \left(\psi \implies \exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right)\right)\right\} \cup \\ & \{\text{drop} \mid M \models (\psi \implies \neg\exists x : \text{peb}_n(x))\} \cup \\ & \{\text{lift} \mid M \models (\psi \implies \exists x : (\text{head}(x) \wedge \text{toppebble}(x)))\}\end{aligned}$$

where $M \models \chi$ stands for "for all trees $t \in T_{\Sigma,\Phi}$ and all input configurations (u, π) in $\text{IC}_{n,t}$, χ is true", and where

$$\begin{aligned}\text{toppebble}(x) &\equiv \bigvee_{k \in [1,n]} (\text{peb}_k(x) \wedge \text{pebcount}_k) \\ \text{pebcount}_i &\equiv \begin{cases} (\exists x : \text{peb}_i(x)) \wedge \text{free}_{i+1} & (\text{when } i > 0) \\ \text{free}_{i+1} & (\text{when } i = 0) \end{cases} \\ \text{free}_i &\equiv \left(\bigwedge_{k \in [i,n]} \neg\exists y : \text{peb}_k(y)\right)\end{aligned}$$

(4) For a condition ψ, the set of *configuration instructions of M on ψ* is defined as $\text{CI}_{M,\psi} = \langle Q, \text{II}_{M,\psi}\rangle$.

(5) For a condition ψ, the set of *instructions of M on ψ* is *defined as* $I_{M,\psi} = R_{\Delta,\Gamma}(\text{CI}_{M,\psi})$.

(6) A rule $r \in R$ is of the form $\langle q, \psi\rangle \to \iota$ with $q \in Q$, ψ a condition, and $\iota \in I_{M,\psi}$ an instruction. There must exist a total function $\text{out}_R : Q \to \mathcal{P}(\Gamma)$ such that for each rule $r : \langle q, \psi\rangle \to \iota$ in R:
   (a) if $\nu_\iota(\text{root}_\iota) = \langle q', \omega\rangle$ then $\text{out}_R(q') \supseteq \text{out}_R(q)$,
   (b) if $\nu_\iota(u) = \langle q', \omega\rangle$ for $u \in V_\iota$, then $\text{out}_R(q') \supseteq \text{out}_\iota(u)$, and
   (c) $\text{out}_\iota(\text{root}_\iota) \cap \text{out}_R(q) = \emptyset$.

The respective components of a ptt M may be referred to as n_M, Σ_M, Φ_M etc. An n-ptt is a ptt M with $n_M = n$ pebbles. A 0-ptt is also called a tree-walking tree transducer or *twtt*.

3.2.1. Overview. Before we give the formal definitions of all the concepts of the ptt's semantics, we start with a high-level overview of the way a ptt works.

Tree Walking. First, we will look at the way it walks over its input tree. A pebble tree transducer $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ walks over an input tree $t \in R_{\Sigma,\Phi}$, much like a tree-walking automaton would. During the walk, it drops and lifts pebbles to temporarily "bookmark" certain locations that it has visited. At every step the ptt is in a *configuration* c, which consists of a state in Q and of an input configuration in $\mathrm{IC}_{n,t}$: information about the whereabouts of the reading head and the pebbles. The ptt starts its tree-walk in the *initial configuration*, which has state q_0 (the *initial state*), no pebbles, and the reading head at the root of the tree. For every configuration c, there is a set of *rules* $R_c \subseteq R$ that are *applicable to* c. Whether a rule $r : \langle q_r, \psi_r \rangle \to \iota_r$ is applicable to c depends on the rule's *condition* ψ_r, which is specified as a closed MSO predicate on the input configuration, and the rule's *state* q_r, which must match the state of configuration c. A rule r prescribes the action that is to be taken in a configuration c to which it is applicable, by way of an *instruction* ι_r.

An instruction ι_r contains zero or more *configuration instructions,* instructions that specify what step to take in order to get to a next configuration starting from c, i.e., these instructions specify a step in the tree walk. A configuration instruction can be one of the following: drop a pebble, lift a pebble, move along an edge with a specific label, or stay where you are. In addition, a configuration instruction specifies a new state for the next configuration. The reason that an instruction may contain multiple configuration instructions is found in how a ptt generates output, which is something that we will explain in the next paragraph. The important thing to understand at this point is that the configuration instructions are not executed serially but *in parallel*: every configuration instruction represents a different single-step continuation of the ptt's tree walk starting from the same configuration, and the tree walk "splits up" into as many parallel tree walks as there are configuration instructions in the rule's instruction ι_r.

Generating Output. Given an input rooted tree $t \in R_{\Sigma,\Phi}$, a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ defines *intermediate output trees on* t as trees in $R_{\Delta,\Gamma}(C_{M,t})$, i.e., rooted trees over output alphabets Δ and Γ, which may also have configurations of M on t as node labels. The nodes in the tree that have labels in Δ are the already-produced output, while the configuration-labeled nodes represent the fact that those nodes still need to be completed, with output that is yet to be produced. As such, a configuration that labels a node represents the beginning of a line of computation that, when calculated to completion, will produce all of the remaining output that should appear at that node.

The computation of a ptt M on an input rooted tree t begins with the *initial intermediate output tree,* a singleton tree in which the node is labeled with the ptt's *initial configuration:* the initial state q_0, the reading head at root_t, and no pebbles on

the stack. As we said earlier, a node labeled with a configuration represents output that is yet to be produced; and the initial configuration represents *all* the output that the ptt will produce. But how does the ptt transform this initial tree into its final output? It does so by performing *computation steps* on the intermediate output tree, for as long as it still contains configurations. A ptt's computation step takes an intermediate output tree, and transforms it by performing *one* step in the tree walk of *one* configuration that is present in the tree. In short, it takes a node u in the intermediate output tree that is labeled with some configuration c, and on this configuration it executes the instruction from a rule that is applicable to this configuration c. The execution of the instruction yields a new segment of intermediate output tree (which may include nodes with labels in Δ, but also nodes labeled with configurations in $C_{M,t}$), and then the new intermediate output tree is created by *gluing* the new segment onto the old intermediate output tree at u. Note that the gluing operation *replaces* node u and the configuration that labeled it – but not its edges: the edges that u already had remain in place.

Now, how does an instruction ι create an intermediate output tree segment? Well, the instruction is basically a *template* for such a tree segment: it has the same nodes and shape as the tree segment that it outputs, but the nodes that are to be labeled with configurations in the instruction's output are labeled with *configuration instructions* instead. This is how, as we noted when we discussed the ptt tree walking mechanism, an instruction can contain zero or more configuration instructions. As we explained at that point, the configuration instructions specify how to transform a configuration into another configuration by traversing an edge or dropping/lifting a pebble, and by modifying the configuration's state. The new segment of intermediate output tree, the result of the instruction's execution, is generated from the template ι and the configuration c on which ι is executed, by executing the individual configuration instructions on c, and by replacing the configuration instructions by their results.

While a generic instruction output may contain several nodes, some labeled with configurations, some with output node labels, there are a number of instruction output forms that are noteworthy because they fulfill a special function. First of all, an instruction can output a tree segment that contains exactly one node, labeled with a configuration. In effect, this instruction output achieves nothing except replacing the prevous configuration in the intermediate output tree by a new one. Using this type of instruction output, the ptt can walk the input tree and perform large calculations without generating any output while it is doing so. Another noteworthy instruction output form is one that contains no configurations *at all*. Such an instruction output terminates the generation of output at the node in the intermediate output tree at which the instruction's output is glued: no configurations means no more new output.

This type of instruction output is the only type of instruction output that *reduces* the number of configurations in the intermediate output tree. For a ptt's computation to succeed, at one point or another all configurations must be removed by an instruction that produces an output of this kind, yielding an intermediate output tree containing no configurations at all: this is the output of the ptt's computation. If a computation eventually yields a tree with no configurations, then we call the ptt's computation *successful*.

To conclude this overview, we still need to discuss a small detail that we have ignored earlier: gluing arbitrary trees onto each other does not necessarily yield valid trees. As we explained, an instruction's output is glued onto the intermediate output tree, at an existing node that is labeled with a configuration. That node may already have a number of outgoing edges, and those edges may conflict with the outgoing edges of the root node of the instruction tree. In order to prevent this statically, for any $q \in Q$ a ptt demands the existence of an upper bound $\mathrm{out}_R(q)$ to the set of the labels of outgoing edges that an intermediate output node labeled with a configuration in that state may have, and it demands that if there is a rule $\langle q, \psi \rangle \to \iota$, then the outgoing edge labels of the root node of ι do not occur in $\mathrm{out}_R(q)$. Fortunately, whether out_R exists is decidable. The smallest $\mathrm{out}_R(q)$ that satisfy the first two conditions of Definition 3.2.1(6) can be calculated as follows. First of all, for every rule that generates an output node labeled with a configuration having state q, the outgoing edge labels of that output node are in $\mathrm{out}_R(q)$, as stated in the second condition of Definition 3.2.1(6). We therefore initialize $\mathrm{out}_R(q)$ with all edge labels found using this method. Secondly, if the output node labeled with a configuration having state q is the *root node* of the instruction's output, then the rule's execution will glue this root node onto a node that already had a configuration on it, and that may already have a set of outgoing edge labels. We also know which edge labels that may be: those associated with the state p of the previous configuration, which is the state named on the left hand side of the rule. So, the elements of $\mathrm{out}_R(p)$ must be in $\mathrm{out}_R(q)$ as well. This is expressed in the first condition of Definition 3.2.1(6). By repeatedly iterating over all rules and adding $\mathrm{out}_R(p)$ to $\mathrm{out}_R(q)$ until out_R is stable (which must happen, as $\mathrm{out}_R(q)$ is bounded), one eventually obtains an out_R that satisfies the first two conditions of Definition 3.2.1(6). As the algorithm only adds elements to the sets that must necessarily be present according to the first two conditions in Definition 3.2.1(6), the resulting sets are the smallest ones that satisfy these two conditions; any other out'_R that also satisfies these conditions must have $\mathrm{out}'_R(q) \supseteq \mathrm{out}_R(q)$ for all q. The third condition of Definition 3.2.1(6), which states that for all rules $\langle q, \psi \rangle \to \iota$, $\mathrm{out}_\iota(\mathrm{root}_\iota) \cap \mathrm{out}_R(q) = \emptyset$, is not necessarily satisfied by the algorithm's result, however. Fortunately, whether the condition is satisfied can be easily verified. When

the condition is satisfied, then we have shown that there exists an out_R that satisfies all of the given conditions. And when the third condition is *not* satisfied by the calculated out_R, then there cannot exist an other out'_R that *does* satisfy the third condition: that would imply that $\mathrm{out}_\iota(\mathrm{root}_\iota) \cap \mathrm{out}'_R(q) = \emptyset$ while $\mathrm{out}_\iota(\mathrm{root}_\iota) \cap \mathrm{out}_R(q) \neq \emptyset$ for some rule $\langle q, \psi\rangle \to \iota$ in R, which cannot be the case given that $\mathrm{out}'_R(q) \supseteq \mathrm{out}_R(q)$. This means that if out_R does not satisfy the third condition, then there are no sets that satisfy all three conditions of Definition 3.2.1(6).

We will continue by giving detailed definitions of all the concepts we just introduced.

3.2.2. Configurations. A *configuration* describes the full state of a ptt at one position in a tree walk. It consists of two parts: an *input configuration*, which describes the locations of the pebbles and the location of the reading head on the input tree, and a *state*.

Input configuration. For a pebble count n and a tree $t \in T_{\Sigma,\Phi}$, an *n-pebble input configuration on t* is a pair $h = (u, \pi)$, where $u \in V_t$ is a *reading head configuration* and $\pi \in V_t^{\leq n}$ is a *pebble configuration*. The reading head configuration indicates the location of the reading head of the ptt, on a node which we call the *current node* for that configuration. The pebble configuration indicates that there are $l = |\pi|$ pebbles on the tree, and that the pebbles numbered $1, \ldots, l$ are present at nodes $\pi(1), \ldots, \pi(l)$, respectively. We will also refer to pebble configurations as pebble *stacks*, because the dropping and lifting of pebbles must conform to a *stack discipline*, i.e., it is only possible to lift the pebble with number l (specifically, it is only possible to lift it *from node* $\pi(l)$), and it is only possible to drop a new pebble with number $l + 1$. This is ensured by the restrictions on the available *input instructions*, transformations on input configurations that will be defined shortly.

As defined earlier in Definition 3.2.1, the set of all n-pebble input configurations on t is denoted by $\mathrm{IC}_{n,t} = V_t \times V_t^{\leq n}$. The set of all input configurations of a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ is $\mathrm{IC}_M = \bigcup_{t \in T_{\Sigma,\Phi}} \mathrm{IC}_{n,t}$.

Configuration. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and a tree $t \in T_{\Sigma,\Phi}$, a *configuration of M on t* is a pair $\langle q, h\rangle \in \langle Q, \mathrm{IC}_{n,t}\rangle$. The meaning of configuration $\langle q, h\rangle$ is that q is the current state, and h is the current input configuration. The set $\langle Q, \mathrm{IC}_{n,t}\rangle$ of all configurations of M on t is denoted $C_{M,t}$. For a configuration $\langle q, h\rangle$, $\mathrm{ic}(\langle q, h\rangle)$ denotes the input configuration h.

3.2.3. Rules. As defined earlier in Definition 3.2.1, a *rule* $r \in R$ of a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ is of the form $\langle q, \psi\rangle \to \iota$, where the left hand side $\langle q, \psi\rangle$ consists of a state $q \in Q$ and a condition $\psi \in \mathrm{CMSO}_{A_M}$, and the right hand side $\iota \in I_{M,\psi}$ is an *instruction*. The state and condition control whether the rule is *applicable* to a given configuration of M. For a configuration $\langle q, (u, \pi)\rangle$ of M on a tree $t \in T_{\Sigma,\Phi}$, a

rule $r \in R : \langle q_r, \psi_r \rangle \to \iota_r$ is *applicable to* $\langle q, (u, \pi) \rangle$ when $q = q_r$ and ψ_r is true for (u, π). The subset of R of rules that are applicable to a given configuration $c \in C_{M,t}$ is denoted by R_c.

3.2.4. Instructions. Transformations on input configurations and configurations are identified by *input instructions* and *configuration instructions*, respectively. *Instructions* are templates for a segment of intermediate output tree.

Input instruction. An *input instruction* identifies a transformation of an input configuration. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and a condition $\psi \in \mathrm{CMSO}_{A_M}$, $\mathrm{II}_{M,\psi}$ is the set of *input instructions on* ψ, as defined earlier in Definition 3.2.1. The set of *all* input instructions that may occur in M is $\mathrm{II}_M = \{\mathrm{stay}, \mathrm{drop}, \mathrm{lift}\} \cup \{\mathrm{go}_\phi \mid \phi \in \Phi\}$. The interpretation of the instructions is as follows. *Stay* instructs the ptt to leave the reading head and the pebbles where they are, go_ϕ moves the reading head in the direction of outgoing edge label ϕ, *drop* drops a pebble on the stack (only available when the stack is not full), and *lift* lifts the topmost pebble from the stack (only available when the topmost pebble is at the reading head location). For a tree $t \in T_{\Sigma,\Phi}$, the *execution* of an input instruction $\omega \in \mathrm{II}_{M,\psi}$ on an input configuration $(u, \pi) \in V_t \times V_t^{\leq n}$ for which condition ψ is true, denoted by $\omega((u, \pi))$, applies the instruction to an input configuration, resulting in a new input configuration in $\mathrm{IC}_{n,t}$. It is defined as follows:

$$\omega((u,\pi)) = \begin{cases} (u, \pi) & \text{if } \omega = \mathrm{stay} \\ (\phi(u), \pi) & \text{if } \omega = \mathrm{go}_\phi \text{ with } \phi \in \Phi \\ (u, \pi u) & \text{if } \omega = \mathrm{drop} \\ (u, \mathrm{left}(\pi, |\pi| - 1)) & \text{if } \omega = \mathrm{lift} \end{cases}$$

Note that because ψ is true for (u, π) and $\omega \in \mathrm{II}_{M,\psi}$, we must have $\omega((u, \pi)) \in \mathrm{IC}_{n,t}$. For instance, go_ϕ is in $\mathrm{II}_{M,\psi}$ if and only if ψ implies $\exists x : \exists y : \big(\mathrm{head}(x) \wedge \mathrm{edge}_\phi(x, y)\big)$, which implies that for the reading head node u (bound to x), there is a node v (bound to y) for which $\epsilon_t(u, v) = \phi$, i.e., $\mathrm{go}_\phi((u, \pi)) = (e(u), \pi)$ is defined. Similarly, the conditions on the *drop* and *lift* instructions ensure that the *drop* instruction is not available when the last pebble has been dropped ($\exists x : \mathrm{peb}_n(x)$), and that the *lift* instruction is not available when there is not at least one node that has a pebble on it.

Configuration instruction. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and a condition $\psi \in \mathrm{CMSO}_{A_M}$, a *configuration instruction* is a pair $\langle q', \omega \rangle \in \langle Q, \mathrm{II}_{M,\psi} \rangle$. The set of all configuration instructions of M is denoted by $\mathrm{CI}_M = \langle Q, \mathrm{II}_M \rangle$, while the set of all configuration instructions of M available for a condition ψ is $\mathrm{CI}_{M,\psi} = \langle Q, \mathrm{II}_{M,\psi} \rangle$. For a tree $t \in T_{\Sigma,\Phi}$, the execution of a configuration instruction $\langle q', \omega \rangle$ on a configuration $\langle q, h \rangle \in C_{M,t}$ is defined as $\langle q', \omega(h) \rangle$.

Instruction. As we explained earlier, a computation step of a pebble tree transducer takes a node of an intermediate output tree that is labeled with a configuration, and

then glues a newly generated segment of intermediate output tree onto the old intermediate output tree at that node. An *instruction* ι is a template for such a tree segment. It consists of a tree over the output alphabets, Δ and Γ, augmented by configuration instructions in $\mathrm{CI}_{M,\psi}$, where ψ is the condition of the rule of which ι is the right hand side.[1] For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and a condition $\psi \in \mathrm{CMSO}_{A_M}$, the set of *instructions of* M *on* ψ is defined as

$$I_{M,\psi} \quad = \quad R_{\Delta,\Gamma}(\mathrm{CI}_{M,\psi})$$

The complete set of *instructions of* M is defined as $I_M = R_{\Delta,\Gamma}(\mathrm{CI}_M)$.

Execution of an instruction. By *executing* an instruction, we create the to-be-glued intermediate output tree segment from the template given by the instruction. Conceptually, a configuration instruction that labels a node of the instruction tree instructs us how to calculate the configuration that is to be placed at that node, from the configuration that the instruction tree is going to replace. The execution of an instruction simply consists of executing all the configuration instructions present as node labels, and replacing the nodes by nodes labeled with the configurations that result from the configuration instruction executions. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$, a tree $t \in T_{\Sigma,\Phi}$, an input configuration $h = (u, \pi) \in \mathrm{IC}_{n,t}$, a condition $\psi \in \mathrm{CMSO}_{A_M}$ that is true for h, and an instruction $\iota \in I_{M,\psi}$, the *execution* $\iota(h)$ *of* ι *on* h , is defined as $\iota(h) = \iota[s_h]$ where the node relabeling $s_h : \mathrm{CI}_{M,\psi} \to C_{M,t}$ is defined as $s_h(\langle q, \omega\rangle) = \langle q, \omega(h)\rangle$. Note that because $\iota \in R_{\Delta,\Gamma}(\mathrm{CI}_{M,\psi})$, and because all node labels in $\mathrm{CI}_{M,\psi}$ are replaced by node labels in $C_{M,t}$, $\iota(h) \in R_{\Delta,\Gamma}(C_{M,t})$. For a configuration $c = \langle q, h\rangle \in C_{M,t}$, we define the *execution* $\iota(c)$ *of* ι *on* c as $\iota(h)$.

3.2.5. Computation.

Intermediate output tree. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and a tree $t \in T_{\Sigma,\Phi}$, an *intermediate output tree of* M *on* t is a rooted tree $d \in R_{\Delta,\Gamma}(C_{M,t})$.[2] Intermediate output trees represent the global state of a pebble tree transducer during its computation of an output tree (as opposed to the local state represented by an individual configuration). The nodes with labels in Δ are the output that has already been produced by the transducer, while each node with a label in $C_{M,t}$ represents unfinished output: the label is the configuration from which the remaining output attached to that node should be computed.

[1] Note that our definition of 'instruction' does not correspond with the one used by Engelfriet and Maneth [**13**]; their definition of 'instruction' corresponds with our definition of 'input instruction'. (For a more extensive discussion of the differences between the various models, see Section 3.3.)

[2] Note that we use d to denote an (intermediate) output tree, which might not be intuitive at first. The following might help the reader to correctly associate d with the concept of a tree: (a) d is a tree that may have node labels in Δ, and (b) d stands for *dendron*, which is the word for tree in ancient Greek.

Initial configuration and initial intermediate output tree. A computation of a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ on an input rooted tree $t \in R_{\Sigma,\Phi}$ always starts in the initial state at root_t, and with no pebbles present. We call this configuration, $\langle q_0, (\mathrm{root}_t, \lambda)\rangle$, the *initial configuration of M on t*. The tree $\mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle)$, which consists of only a root node labeled with the initial configuration of M, is called the *initial intermediate output tree of M on t*.

Computation relation. A computation of a pebble tree transducer M is a transformation from one intermediate output tree of M to another. In a computation step, a node u labeled with a configuration is taken, a rule $r \in R$ is selected that is applicable to that configuration, the rule's instruction is executed on the node's label, and then the resulting intermediate output tree segment is glued onto the intermediate output tree at node u, resulting in a new intermediate output tree. Formally, for a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and a tree $t \in T_{\Sigma,\Phi}$, the *computation step relation of M on t*, denoted by $\Rightarrow_{M,t}$, is the binary relation over intermediate output trees such that for intermediate output trees $d, d' \in R_{\Delta,\Gamma}(C_{M,t})$, the relation $d \Rightarrow_{M,t} d'$ holds if:

- for all $w \in V_d$ labeled in $C_{M,t}$, with $\nu_d(w) = \langle q', c\rangle$, the relationship $\mathrm{out}_d(w) \subseteq \mathrm{out}_R(q')$ holds,
- there is a node $u \in V_d$ where $\nu_d(u) \in C_{M,t}$, and there is a rule $\langle q, \psi\rangle \to \iota$ in R that is applicable to $\nu_d(u)$, such that

$$d' \cong d\left[u \leftarrow \iota\left(\nu_d(u)\right)\right]$$

The node u is called the *computation step node* of computation step $d \Rightarrow_{M,t} d'$. The relation $\Rightarrow^*_{M,t}$ is the *computation relation of M on t*. Note that because the first of the above conditions holds for d, it also holds for d'.

3.2.6. Various Properties. A ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ may have the following properties:

- *Total*: If for every tree $t \in T_{\Sigma,\Phi}$, every configuration $\langle q, h\rangle \in C_{M,t}$ has at least one applicable rule in R, then M is *total*.
- *Deterministic*: If for every tree $t \in T_{\Sigma,\Phi}$, every configuration $\langle q, h\rangle \in C_{M,t}$ has at most one applicable rule in R, then M is *deterministic*, and we call it a **d**ptt or an n-**d**ptt.

A pebble $i \in [1, n]$ of M may be either *global* or *local*. Conceptually, if a pebble is local, then its position can have no influence on the automaton's behaviour when the pebble is not at the top of the pebble stack. Formally, a pebble $i \in [1, n]$ is local iff $i < n$ and for all rules $\langle q, \psi\rangle \to \iota$ that use an atomic formula in $\mathrm{AF}_{\mathrm{peb}_i}$, $M \models \psi \implies \mathrm{free}_{i+1}$. If a pebble is not local, then it is global. Note that this definition implies that pebble n is always global (provided that $n > 0$, of course).

3.2.7. Translation.

Translation. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$, the *translation realized by* M, denoted by τ_M, is defined as

$$\tau_M = \left\{(t, d) \in R_{\Sigma,\Phi} \times R_{\Delta,\Gamma} \mid \mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow^*_{M,t} d\right\}.$$

Two transducers (of any type) M_1 and M_2 are *equivalent* if $\tau_{M_1} = \tau_{M_2}$.

Translation classes. The class of all translations realized by n-ptts is denoted by n-PTT, and the class of all translations realized by n-dptts is denoted by n-DPTT. Furthermore, we define PTT as $\bigcup_{n\geq 0} n$-PTT, and DPTT as $\bigcup_{n\geq 0} n$-DPTT . The class 0-PTT is also referred to as TWTT, the class of all translations realized by tree-walking tree transducers.

Pebble tree transducer types. The translation class names n-PTT, PTT etc. correspond with pebble tree transducers of a certain type. The set of all possible n-ptts is denoted n-ptt, and so on.

3.3. Comparison with Other Pebble Tree Transducer Models

Both Engelfriet and Maneth **[13]** and Milo, Suciu and Vianu **[17]** define a model for pebble tree transducers, and these models are both different from our model and from each other. The main differences are:

(1) Our ptt model is based on graphs with labeled edges, while both other models use trees that are based on terms over ranked alphabets. It is possible to map between a term-based tree model and our graph model, so this difference does not affect the relative strength of our ptt model.

(2) We allow generalized instructions, while Engelfriet and Maneth only allow simplified instructions. As simplified instructions are a normal form, this does not lead to a difference in model strength.

(3) We allow internal nodes of intermediate output trees to be labeled with configurations. In our graph-based tree model this makes sense: any extra output is simply glued onto the node in question, an operation which is no different for leaves and internal nodes except for the fact that the internal nodes may have more than one existing outgoing edge, where a leaf has one at most. We have reason to believe that the ptt that does not allow configurations on internal nodes is a normal form of our generic ptt model: we expect that it is possible to use MSO logic to predict *all* output that will be generated at a node. However, converting a ptt to such a normal form does come at a price: the MSO-based prediction may have to look at pebbles that are not at the top of the stack, forcing them to be global, while these pebbles may have been local in the denormalized ptt. As we will show in Chapter 4, the number of

global pebbles influences the number of twtts into which the ptt can be decomposed. We therefore believe that allowing configurations at internal nodes of intermediate output trees makes the ptt model strictly stronger.

(4) Where configuration tests in both other ptt models are relatively limited, we allow conditions to be specified as MSO predicates on the configuration. The MSO predicates can express all configuration tests that the other models are able to perform. This makes our model considerably more powerful.

(5) There is one thing we have not allowed our MSO predicates to do, which is to perform a *direct root test*: the MSO predicate has no built-in way of determining which node is the root node. The literature is divided on whether to provide a root check: the original ptt model described by Milo et al. **[17]** did not provide a root check, but Engelfriet and Maneth **[13]** did provide it. We considered adding a relation symbol to test whether a node was the root, but we thought this would be rather arbitrary: the physical root node of a tree is not always the *logical* root node, and there is no reason to give the root such a special place. A primary example for this is that in the encoded trees we use for our decomposition proof in Chapter 4, we never use the root at all. Fortunately, when one *does* want to know the (logical) root node, there are ways to provide that information. For instance, one can encode the "upward direction" in the labeling. In that case, a node that has no outgoing "upward" edges is the root. In fact, this is the way the root is encoded in tree models based on terms over ranked alphabets: in those models, the upward edge is always labeled with number -1. As the term-based ptt models already use this encoding, it can be argued that our ptt model is not weakened by the lack of a built-in root check: when provided with the same input as the other ptt models, having the same edge labels, our ptt model *can* detect the root node.

(6) In the model by Milo et al. there is no concept of a reading head: instead, they use an extra pebble (which they call the *current pebble*) which is always at the top of the stack, and which is moved around as if it were a reading head. The *drop* operation drops a new 'current' pebble at the root of the tree (leaving the old 'current' pebble where it was, the equivalent of 'dropping' it in our model), while the *lift* operation simply picks up the current pebble and makes the next pebble on the stack the current pebble, wherever it is located. Translated to our model, this means that *drop* drops a pebble and moves the reading head to the root, and *lift* moves the reading head to the location of the pebble at the top of the stack and then picks it up. While the ptts of Engelfriet and Maneth can emulate the behaviour of the ptts of Milo et al.

Our Model	**Engelfriet and Maneth**
Input Configuration	Input Configuration
Configuration	Configuration
Condition	Test
Input Instruction	Instruction
Configuration Instruction	*Not an explicit concept.*
Instruction	Right-hand side of a rule

TABLE 1. Mapping between the concepts of the model of Engelfriet and Maneth **[13]** and our model.

and vice versa **[13**, section 3.1], our ptts cannot emulate the behaviour of the ptts by Milo et al. because our model lacks a "root check".

(7) Both we and Engelfriet and Maneth consider both deterministic and nondeterministic ptts, while Milo et al. consider only nondeterministic ptts.

Table 1 contains a convenient overview of how the concepts of Engelfriet and Maneth's model relate to our concepts.

CHAPTER 4

Decomposition

4.1. Introduction

Engelfriet and Maneth showed that an n-ptt can be decomposed into $n+1$ tree-walking tree transducers (twtts) **[13]**. The method that they used to perform this decomposition is just one example of a larger class of decomposition methods. In this class of methods, a decomposition is obtained by repeatedly decomposing an n-ptt M into a twtt A and an m-ptt M' such that $\tau_M = \tau_A \tau_{M'}$ and $m < n$. The end result of such a decomposition is a sequence $A_1, \ldots, A_k$ of twtts and a 0-ptt $M^{(k)}$, for some $k \leq n$, such that $\tau_M = \tau_{A_1} \cdots \tau_{A_k} \tau_{M^{(k)}}$. The 0-ptt $M^{(k)}$ is of course a twtt as well, so the result is a decomposition into $k+1$ twtts with $k \leq n$.

In this chapter, we first define a general framework for all decompositions of ptts into twtts using this incremental decomposition method. Within this framework, we then identify a subclass of simple decomposition methods that are based on *stepwise simulation*. This subclass covers all the decomposition methods we will discuss in this thesis. After defining the framework we use it to create an extended version of Engelfriet and Maneth's method that supports our extended ptt model. Using this extended method, we can decompose an n-ptt M into $k+1$ twtts, where k is the number of global pebbles in M.

4.2. A Framework for Incremental Decomposition

In the remainder of this thesis, we will conveniently use the term *decomposition* to mean a decomposition using the incremental decomposition framework defined here, unless we indicate otherwise.

DEFINITION 4.2.1. For an n-ptt M, an m-ptt M' and a twtt A, the pair $(M, (A, M'))$ is a *decomposition step* if $m < n$ and $\tau_M = \tau_A \tau_{M'}$.

For an n-ptt M, $k \geq 1$, twtts A_i $(1 \leq i \leq k)$, and an m-ptt M', a sequence $A_1, \ldots, A_k, M'$ is a *decomposition of* M if $m \leq n - k$ and $\tau_M = \tau_{A_1} \cdots \tau_{A_k} \tau_{M'}$. If $m = 0$, then the decomposition is *complete*, otherwise it is *partial*.

It is clear that if we have a partial decomposition $A_1, \ldots, A_k, M'$ and a decomposition step $\langle M', (A_{k+1}, M'') \rangle$, then $A_1, \ldots, A_{k+1}, M''$ is a decomposition as well, where M'' is a ptt with less pebbles than M'. By repeatedly applying decompositon steps to

the rightmost ptt in the decomposition we will always arrive at a complete decomposition. The only requirement for this to work is that we can find decomposition steps that can be applied. So, we need to have a method for finding decomposition steps.

DEFINITION 4.2.2. A *decomposition method* is a function $\delta : (\text{ptt} - \text{twtt}) \to (\text{twtt} \times \text{ptt})$ such that all $(M, (A, M')) \in \delta$ are decomposition steps. The decomposition method is *total* if the function δ is total. If for a ptt subtype xptt $\subseteq$ ptt, $\delta(M) = (A, M')$ and $M \in$ xptt imply $M' \in$ xptt and $A \in$ xptt, then xptt is *closed under decomposition by* δ. If $\delta(M)$ exists for every $M \in \text{xptt} - \text{twtt}$, then δ is *total on xptt*.

It is easy to see that when we have a total decomposition method δ, we can find a complete decomposition for any n-ptt M by repeatedly applying the decomposition method. When a decomposition method δ is not total on ptt, but when we do have a subset xptt $\subseteq$ ptt on which δ *is* total, and when this subset xptt is also closed under decomposition by δ, then we can still find a complete decomposition for any ptt $M \in$ xptt. We will use this fact later to show that the decomposition method that we will develop also applies to some restricted ptt subtypes, including subtypes that are similar in operation to Engelfriet and Maneth's ptts.

4.3. Decomposition Methods using Stepwise Simulation

4.3.1. Introduction. To find a decomposition step $(M, (A, M'))$, we need to create an M' that uses less pebbles than M, and that works on a transformed tree that can be generated by a twtt A, with $\tau_A \tau_{M'} = \tau_M$. Speaking in terms of configurations, we want to build a ptt M' that has a smaller *pebble configuration* space than M. However, M' must be able to represent every M-configuration by a configuration of its own, so M' needs to have a combined configuration space that is at least as large as that of M. So, the obvious course of action is to enlarge the spaces of the other configuration components, the *reading head* and the *state*: in other words, we need to encode the pebble locations of M in the state and in the reading head of M'. Observe that the pebble configuration space that needs to be encoded is unbounded: the amount of information that needs to be encoded elsewhere if we remove even a single pebble from the pebble configuration space is proportional to the size of the input tree, which is unbounded. Unfortunately, the state space can only be enlarged by a finite amount, because the number of states of a ptt is finite. So, the only way we can encode a pebble position is by encoding some amount of it in the reading head position. And in order to do that, we need to have a larger tree, with enough nodes to yield a configuration space for M' that is at least as large as that of M. If we want to encode a reading head position and m pebble positions in the reading head without

enlarging the state space, we will need to use a tree that has a node for every combination (u, p) of a reading head position u and a string of pebble locations p ($|p| \leq m$): the size of this enlarged tree t' in terms of the size of the original tree t will therefore satisfy $|V_{t'}| \geq |V_t| (1 + |V_t| + |V_t|^2 + \cdots + |V_t|^m)$. This equation follows from the fact that u has $|V_t|$ possible values, which must be multiplied by the number of different configurations for p, which can represent zero pebbles (1 possible configuration), one pebble ($|V_t|$ possible configurations), two pebbles ($|V_t|^2$ possible configurations), and so on for up to m pebbles.

If we perform decomposition by transforming the tree, changing the configuration space, and transforming the ptt accordingly, we must have a method to prove that the tree transformation combined with the transformed ptt results in the same translation as the original ptt. We do this by requiring that the new ptt together with the transformed tree perform a *stepwise simulation* of the original ptt. In the next paragraphs, we will define a very restricted form of stepwise simulation, and we will then show that any original ptt, alternative ptt and tree transformation that satisfy this definition yield the transformation equality that we require.

4.3.2. Stepwise Simulation.

DEFINITION 4.3.1. For ptts $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and $M' = (n', (\Sigma', \Phi'), (\Delta, \Gamma), Q', q_0', R')$ and rooted trees $t \in R_{\Sigma,\Phi}$ and $t' \in R_{\Sigma',\Phi'}$, $\mathrm{CC}_{M,t,M',t'}$ is the set of all total injective functions $\mathrm{cc} : C_{M,t} \to C_{M',t'}$ (which should be read as "corresponding configurations") that map the configurations of M on t to the configurations of M' on t', in such a way that:

(1) The initial configurations are mapped onto each other:
$\mathrm{cc}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) = \langle q_0', (\mathrm{root}_{t'}, \lambda)\rangle$.

(2) For all $c \in C_{M,t}$ and for all rules $r : \langle q, \psi\rangle \to \iota$ in R that are applicable to c, there is a rule $r' : \langle q', \psi'\rangle \to \iota'$ in R' that is applicable to $\mathrm{cc}(c)$ for which $\iota'(\mathrm{cc}(c)) \cong \iota(c)\,[\mathrm{cc}]$.

(3) For all $c \in C_{M,t}$ and for all rules $r' : \langle q', \psi'\rangle \to \iota'$ in R' that are applicable to $\mathrm{cc}(c)$, there is a rule $r : \langle q, \psi\rangle \to \iota$ in R that is applicable to c for which $\iota'(\mathrm{cc}(c)) \cong \iota(c)\,[\mathrm{cc}]$.

For ptts $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and $M' = (n', (\Sigma', \Phi'), (\Delta, \Gamma), Q', q_0', R')$, and a total function $\tau : R_{\Sigma,\Phi} \to R_{\Sigma',\Phi'}$, *$M'$ performs a stepwise simulation of M using a tree transformed by τ* if, for all trees $t \in R_{\Sigma,\Phi}$, $\mathrm{CC}_{M,t,M',\tau(t)}$ is nonempty.[1]

[1]Note that our definition of stepwise simulation requires one-on-one correspondence of steps, which is not required for stepwise simulation in general. The reason we included this restriction is that we do not actually require anything more than this for the things we are going to prove, and it simplifies the proofs significantly.

We will now show that for any M, M' and τ that satisfy our definition of stepwise simulation, the translation realized by M' applied to a tree transformed by τ is equal to the one realized by M. In order to prove this, we require the following technical lemmas.

LEMMA 4.3.2. *Let Σ and Δ be node label alphabets, let Φ be an edge label alphabet, let $\Sigma' \subseteq \Sigma$ and $\Delta' \subseteq \Delta$ be subsets of the node label alphabets so that $(\Sigma - \Sigma') \cup \Delta' = \Delta$, and let $s : \Sigma' \to \Delta'$ be a total function. Furthermore, let $t, d \in R_{\Sigma,\Phi}$, and let $u \in V_t$ be a node of t. Then $t\,[s]\,[u \leftarrow d\,[s]] = t\,[u \leftarrow d]\,[s]$.*

PROOF. For any edge labeling function ϵ, node labeling function ν and (edge) homomorphism h, let $h(\epsilon) = \{((h(u), h(v)), \phi) \mid ((u, v), \phi) \in \epsilon\}$ and $h(\nu) = \{(h(w), \sigma) \mid (w, \sigma) \in \nu\}$. In general, for any rooted trees t, d and node $u \in V_t$, the following equalities hold for gluing $t\,[u \leftarrow d]$:

$$\begin{aligned}
\nu_{t[u\leftarrow d]} &= h_{u,\mathrm{root}_d,1}\left(\nu_t - \{(u, \nu_t(u))\}\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_d,2}\left(\nu_d\right) \\
\epsilon_{t[u\leftarrow d]} &= h_{u,\mathrm{root}_d,1}\left(\epsilon_t\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_d,2}\left(\epsilon_d\right) \\
\mathrm{root}_{t[u\leftarrow d]} &= h_{u,\mathrm{root}_d,1}\left(\mathrm{root}_t\right)
\end{aligned}$$

Let $s' : \Sigma \to \Delta$ be defined as in the definition of node relabeling, and for any node labeling function let $s'(\nu) = \nu s'$. In general, for any tree t and a node relabeling $t\,[s]$, the equalities $\nu_{t[s]} = s'(\nu)$, $\epsilon_{t[s]} = \epsilon_t$ and $\mathrm{root}_{t[s]} = \mathrm{root}_t$ hold. Note that for any (edge) homomorphism h, $s'(h(\nu)) = h(s'(\nu)) = \{(h(w), s'(\sigma)) \mid (w, \sigma) \in \nu\}$, as h only substitutes node identities, while s' only substitutes labels. Then:

$$\begin{aligned}
\epsilon_{t[u\leftarrow d][s]} &= \epsilon_{t[u\leftarrow d]} \\
&= h_{u,\mathrm{root}_d,1}\left(\epsilon_t\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_d,2}\left(\epsilon_d\right) \\
&= h_{u,\mathrm{root}_d,1}\left(\epsilon_{t[s]}\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_d,2}\left(\epsilon_{d[s]}\right) \\
&= \epsilon_{t[s][u\leftarrow d[s]]}
\end{aligned}$$

$$\begin{aligned}
\nu_{t[u\leftarrow d][s]} &= s'\left(\nu_{t[u\leftarrow d]}\right) \\
&= s'\left(h_{u,\mathrm{root}_d,1}\left(\nu_t - \{(u, \nu_t(u))\}\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_d,2}\left(\nu_d\right)\right) \\
&= h_{u,\mathrm{root}_d,1}\left(s'\left(\nu_t - \{(u, \nu_t(u))\}\right)\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_d,2}\left(s'\left(\nu_d\right)\right) \\
&= h_{u,\mathrm{root}_{d[s]},1}\left(\nu_{t[s]} - \left\{\left(u, \nu_{t[s]}(u)\right)\right\}\right) \cup_{\not\emptyset} h_{u,\mathrm{root}_{d[s]},2}\left(\nu_{d[s]}\right) \\
&= \nu_{t[s][u\leftarrow d[s]]}
\end{aligned}$$

$$\begin{aligned}\mathrm{root}_{t[u\leftarrow d][s]} &= \mathrm{root}_{t[u\leftarrow d]} \\ &= h_{u,\mathrm{root}_d,1}\left(\mathrm{root}_t\right) \\ &= h_{u,\mathrm{root}_{d[s]},1}\left(\mathrm{root}_{t[s]}\right) \\ &= \mathrm{root}_{t[s][u\leftarrow d[s]]}\end{aligned}$$

□

LEMMA 4.3.3. *Let $t, \tilde{t}, d, \tilde{d} \in R_{\Sigma,\Phi}$ be rooted trees over some node and edge label alphabets Σ and Φ, and let $u \in V_t$. If $t \cong \tilde{t}$ and $d \cong \tilde{d}$ with isomorphisms $h_{t\to\tilde{t}}$ and $h_{d\to\tilde{d}}$, then $t[u \leftarrow d] \cong \tilde{t}[h_{t\to\tilde{t}}(u) \leftarrow \tilde{d}]$.*

PROOF. This lemma is provided without proof. □

LEMMA 4.3.4. *Let Σ and Δ be node label alphabets, let Φ be an edge label alphabet, let $\Sigma' \subseteq \Sigma$ and $\Delta' \subseteq \Delta$ be subsets of the node label alphabets so that $(\Sigma - \Sigma') \cup \Delta' = \Delta$, and let $s : \Sigma' \to \Delta'$ be a total function. Furthermore, let $t, \tilde{t} \in R_{\Sigma,\Phi}$ be rooted trees. If $t \cong \tilde{t}$, then $t[s] \cong \tilde{t}[s]$.*

PROOF. This lemma is provided without proof. □

The next two lemmas show that when our definition of stepwise simulation applies, for every intermediate output tree of the simulated automaton there is a corresponding intermediate output tree in the simulating automaton, and every intermediate output tree in the simulating automaton corresponds to an intermediate output tree in the simulated automaton.

LEMMA 4.3.5. *Let $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and $M' = (n', (\Sigma', \Phi'), (\Delta, \Gamma), Q', q_0', R')$ be ptts, and let $\tau : R_{\Sigma,\Phi} \to R_{\Sigma',\Phi'}$ be a total function. Let M' perform a stepwise simulation of M using a tree transformed by τ, and let $t \in R_{\Sigma,\Phi}$, $cc_t \in CC_{M,t,M',\tau(t)}$ and $k \geq 0$. Then for all $d \in R_{\Delta,\Gamma}(C_{M,t})$, if $snt(\langle q_0, (root_t, \lambda)\rangle) \Rightarrow^k_{M,t} d$, then $snt\left(\left\langle q_0', \left(root_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d\,[cc_t]$.*

PROOF. We will prove this by induction on k, the length of the computations.

Property: For a given $k \geq 0$, for all $d \in R_{\Delta,\Gamma}(C_{M,t})$, if $\mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) \Rightarrow^k_{M,t} d$, then $\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d\,[\mathrm{cc}_t]$.

Base ($k = 0$): For $k = 0$ we must have $d = \mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle)$. By Definition 4.3.1, $d\,[\mathrm{cc}_t] = \mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right)$, so $\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^0_{M',\tau(t)} d\,[\mathrm{cc}_t]$.

Induction: Assume the property holds for some $k \geq 0$. Now let $d \in R_{\Delta,\Gamma}(C_{M,t})$ so that $\mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) \Rightarrow^{k+1}_{M,t} d$. Then there must be a $d_0 \in R_{\Delta,\Gamma}(C_{M,t})$ so that $d_0 \Rightarrow_{M,t} d$ and $\mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) \Rightarrow^k_{M,t} d_0$. By the induction assumption,

$\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d_0\left[\mathrm{cc}_t\right]$. By the definition of $d_0 \Rightarrow_{M,t} d$, there are a node $u \in V_{d_0}$ where $c = \nu_{d_0}(u) \in C_{M,t}$ and a rule $\langle q, \psi\rangle \to \iota$ in R that is applicable to c so that $d \cong d_0\left[u \leftarrow \iota(c)\right]$. Then $c' = \nu_{d_0[\mathrm{cc}_t]}(u) = \mathrm{cc}_t(c)$, and by Definition 4.3.1 there then exists a rule $\langle q', \psi'\rangle \to \iota'$ in R' that is applicable to c', so that $\iota'(c') \cong \iota(c)\left[\mathrm{cc}_t\right]$. Then $d_0\left[\mathrm{cc}_t\right] \Rightarrow_{M',\tau(t)} d_0\left[\mathrm{cc}_t\right]\left[u \leftarrow \iota'(c')\right]$, and because (using Lemmas 4.3.3, 4.3.2 and 4.3.4) $d_0\left[\mathrm{cc}_t\right]\left[u \leftarrow \iota'(c')\right] \cong d_0\left[\mathrm{cc}_t\right]\left[u \leftarrow \iota(c)\left[\mathrm{cc}_t\right]\right] = d_0\left[u \leftarrow \iota(c)\right]\left[\mathrm{cc}_t\right] \cong d\left[\mathrm{cc}_t\right]$, we have $\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d_0\left[\mathrm{cc}_t\right] \Rightarrow_{M',\tau(t)} d\left[\mathrm{cc}_t\right]$, which proves that the property holds for $k+1$ and therefore for all k. □

LEMMA 4.3.6. *Let $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and $M' = (n', (\Sigma', \Phi'), (\Delta, \Gamma), Q', q_0', R')$ be ptts, and let $\tau : R_{\Sigma,\Phi} \to R_{\Sigma',\Phi'}$ be a total function. Let M' perform a stepwise simulation of M using a tree transformed by τ, and let $t \in R_{\Sigma,\Phi}$, $cc_t \in CC_{M,t,M',\tau(t)}$ and $k \geq 0$. Then for all $d' \in R_{\Delta,\Gamma}\left(C_{M',\tau(t)}\right)$, if $snt\left(\left\langle q_0', \left(root_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d'$ then there exists a $d \in R_{\Delta,\Gamma}(C_{M,t})$ such that $snt(\langle q_0, (root_t, \lambda)\rangle) \Rightarrow^k_{M,t} d$ and $d' \cong d\left[cc_t\right]$.*

PROOF. We will prove this by induction on k, the length of the computations.

Property: For a given $k \geq 0$, for all $d' \in R_{\Delta,\Gamma}\left(C_{M',\tau(t)}\right)$, if

$$\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d',$$

then there exists a $d \in R_{\Delta,\Gamma}(C_{M,t})$, such that $\mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) \Rightarrow^k_{M,t} d$ and $d' \cong d\left[\mathrm{cc}_t\right]$.

Base ($k = 0$): The only computation of length 0 is $\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d'$ with $d' = \mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right)$. We have $d' \cong d\left[\mathrm{cc}_t\right]$ where

$$d = \mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right),$$

because $\mathrm{cc}_t(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) = \left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle$ by Definition 4.3.1. We also have

$$\mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow^0_{M,t} \mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) = d.$$

This proves that the property holds for $k = 0$.

Induction: Assume the property holds for k ($k \geq 0$). Now let $d', d_0' \in R_{\Delta,\Gamma}\left(C_{M',\tau(t)}\right)$ so that $\mathrm{snt}\left(\left\langle q_0', \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow^k_{M',\tau(t)} d_0' \Rightarrow_{M',\tau(t)} d'$. Since the property holds for k, there exists a tree $d_0 \in R_{\Delta,\Gamma}(C_{M,t})$, so that $\mathrm{snt}(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) \Rightarrow^k_{M,t} d_0$ and $d_0\left[\mathrm{cc}_t\right] \cong d_0'$ by some isomorphism $h_{d_0[\mathrm{cc}_t] \to d_0'}$. As $d_0' \Rightarrow_{M',\tau(t)} d'$, there is a node $v' \in V_{d_0'}$ labeled with a configuration $c' \in C_{M',\tau(t)}$, and there is rule $r' : \langle q', \psi'\rangle \to \iota'$ in R' which is applicable to c', such that $d' \cong d_0'\left[v' \leftarrow \iota'(c')\right]$. Let $v = h^{-1}_{d_0[\mathrm{cc}_t] \to d_0'}(v') \in V_{d_0}$ be the corresponding node in d_0. Then $c = \nu_{d_0}(v) = \mathrm{cc}_t^{-1}\left(\nu_{d_0'}(v')\right) = \mathrm{cc}_t^{-1}(c')$. By condition 3 of Definition 4.3.1, there is also a rule $r : \langle q, \psi\rangle \to \iota$ in R which is applicable to c, having $\iota(c)\left[\mathrm{cc}_t\right] \cong \iota'(c')$. Now let $d = d_0\left[v \leftarrow \iota(c)\right]$. We then have $d_0 \Rightarrow_{M,t} d$ and

therefore $\mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow_{M,t}^{k+1} d$. Now Lemma 4.3.3 and Lemma 4.3.2 give us:

$$\begin{aligned} d' &\cong d'_0\left[v' \leftarrow \iota'\left(c'\right)\right] \\ &\cong d_0\left[\mathrm{cc}_t\right]\left[v \leftarrow \iota\left(c\right)\left[\mathrm{cc}_t\right]\right] \\ &= d_0\left[v \leftarrow \iota\left(c\right)\right]\left[\mathrm{cc}_t\right] \\ &= d\left[\mathrm{cc}_t\right] \end{aligned}$$

This completes the proof that the property also holds for $k+1$, and therefore for all k. □

LEMMA 4.3.7. *Let* $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ *and* $M' = (n', (\Sigma', \Phi'), (\Delta, \Gamma), Q', q'_0, R')$ *be ptts and let* $\tau : R_{\Sigma,\Phi} \to R_{\Sigma',\Phi'}$ *be a total function so that* M' *performs a stepwise simulation of* M *using a tree transformed by* τ. *Then for all* $(t, d) \in R_{\Sigma,\Phi} \times R_{\Delta,\Gamma}$,

$$\mathit{snt}\left(\langle q_0, (\mathit{root}_t, \lambda)\rangle\right) \Rightarrow_{M,t}^{*} d$$

if and only if

$$\mathit{snt}\left(\left\langle q'_0, \left(\mathit{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow_{M',\tau(t)}^{*} d.$$

PROOF. By Definition 4.3.1, $\mathrm{CC}_{M,t,M',\tau(t)}$ is nonempty. Let $\mathrm{cc}_t \in \mathrm{CC}_{M,t,M',\tau(t)}$.

($\Rightarrow$) $\mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow_{M,t}^{*} d$ implies $\mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow_{M,t}^{k} d$ for some k, and by Lemma 4.3.5 this implies $\mathrm{snt}\left(\left\langle q'_0, \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow_{M',\tau(t)}^{k} d'$ for $d' = d\left[\mathrm{cc}_t\right]$, and as $d \in R_{\Delta,\Gamma}$, this means $d' = d$.

($\Leftarrow$) $\mathrm{snt}\left(\left\langle q'_0, \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow_{M',\tau(t)}^{*} d$ implies

$$\mathrm{snt}\left(\left\langle q'_0, \left(\mathrm{root}_{\tau(t)}, \lambda\right)\right\rangle\right) \Rightarrow_{M',\tau(t)}^{k} d$$

for some k, and by Lemma 4.3.6 this implies that there exists a $d' \in R_{\Delta,\Gamma}\left(C_{M,t}\right)$ so that $\mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow_{M,t}^{k} d'$ with $d \cong d'\left[\mathrm{cc}_t\right]$. As $d \in R_{\Delta,\Gamma}$ and $\mathrm{cc}_t : C_{M,t} \to C_{M',\tau(t)}$, we must have $d' \cong d$, and as the computation relation $\Rightarrow_{M,t}$ is defined modulo isomorphism, this implies $\mathrm{snt}\left(\langle q_0, (\mathrm{root}_t, \lambda)\rangle\right) \Rightarrow_{M,t}^{k} d$. □

THEOREM 4.3.8. *Let* $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ *and* $M' = (n', (\Sigma', \Phi'), (\Delta, \Gamma), Q', q'_0, R')$ *be ptts and let* $\tau : R_{\Sigma,\Phi} \to R_{\Sigma',\Phi'}$ *be a total function so that* M' *performs a stepwise simulation of* M *using a tree transformed by* τ. *Then* $\tau_M = \tau\tau_{M'}$.

4.4. The Decomposition Method

In this section, we will define a decomposition method δ. Given a ptt

$$M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$$

with $n > 0$, we will specify a ptt $M' = (n - m, (\Sigma, \Phi'), (\Delta, \Gamma), Q, q_0, R')$ and a total function $\tau_{m,\Sigma,\Phi} : R_{\Sigma,\Phi} \to R_{\Sigma,\Phi'}$ so that M' performs a stepwise simulation

of M using a tree transformed by $\tau_{m,\Sigma,\Phi}$. In addition, we will specify twtt $A = (0, (\Sigma, \Phi), (\Sigma, \Phi'), Q_A, q_{0,A}, R_A)$ so that $\tau_A = \tau_{m,\Sigma,\Phi}$. For brevity, we will write $\tau = \tau_{m,\Sigma,\Phi}$.

DEFINITION 4.4.1. For a ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$, $\mathrm{fg}(M)$ is the number of the lowest-numbered or *first global pebble* in M.

The variable m signifies the number of encoded pebbles, its value must be chosen for every decomposition. It may be chosen anywhere in the range $[1, \mathrm{fg}(M)]$; choosing $m = 1$ yields a decomposition method equivalent to Engelfriet and Maneth's method, while $m = \mathrm{fg}(M)$ yields the shortest decomposition that is possible with this method. The decomposition method works as follows. Transformation τ first encodes the pebbles up to and including m in its transformed tree by adding extra nodes. Then M' makes use of the extra nodes (and therefore reading head configurations) in the τ-transformed tree to keep track of the location of the first m pebbles. Effectively, the decomposition M' performs a stepwise simulation of M using transformation τ. To arrive at an M' and τ that satisfy the stepwise simulation conditions, we will start by defining for a rooted tree $t \in R_{\Sigma,\Phi}$ an input configuration mapping $\mathrm{cic}_t : \mathrm{IC}_{n,t} \to \mathrm{IC}_{n-m,\tau(t)}$, and we will use that mapping to provide a complete definition of τ and M'. We then construct a twtt A with $\tau_A = \tau$. Finally, we prove that $\mathrm{cc}_t(\langle q, (u, \pi)\rangle) = \langle q, \mathrm{cic}_t(u, \pi)\rangle$ is a member of $\mathrm{CC}_{M,t,M',\tau(t)}$, which by the definition of stepwise simulation implies that M' performs a stepwise simulation of M using tranformation τ. As $\tau_A = \tau$, Theorem 4.3.8 then implies that $(M, (A, M'))$ is a decomposition step. Furthermore, the method by which we construct (A, M') from M constitutes a total decomposition method, because the process of constructing A where $\tau_A = \tau_{m,\Sigma,\Phi}$ is defined for all Σ, Φ and $m \geq 1$, and the construction of M' from M is defined for every $M \in \mathrm{ptt} - \mathrm{twtt}$.

4.4.1. The Configuration Mapping. Consider trees $t \in R_{\Sigma,\Phi}$ and $\tau(t) \in R_{\Sigma,\Phi'}$ for a to-be-defined transformation $\tau : R_{\Sigma,\Phi} \to R_{\Sigma,\Phi'}$, an input configuration $h = (u, \pi)$ in $\mathrm{IC}_{n,t}$, and the corresponding input configuration $h' = \mathrm{cic}_t(h) = (u', \pi')$ according to a to-be-defined mapping $\mathrm{cic}_t : \mathrm{IC}_{n,t} \to \mathrm{IC}_{n-m,\tau(t)}$.

First, look at the reading head configuration in M'. Encoding m pebbles in the tree means that we are going to expand the number of possible configurations of the reading head compared to M. Without committing to a specific tree transformation, we can state that $u' = \mathrm{pebenc}_t(u, \mathrm{left}(\pi, m))$ for some injective total function $\mathrm{pebenc}_t : V_t \times V_t^{\leq m} \to V_{\tau(t)}$. We will choose a specific function pebenc_t in the next section, where we will discuss the tree transformation.

Next, look at the pebble configuration π'. Because we encode the first m pebbles of M in the tree, we are going to skip the leftmost m elements of π when we construct π' from it: π' only needs to represent the pebbles $\pi(m+1)\,\pi(m+2)\cdots\pi(n)$. The

pebble locations in π' will, of course, be nodes from the transformed tree $\tau(t)$, not from the original tree t. But which nodes? The encoding function pebenc_t requires not only a node, but also the leftmost m pebbles on the stack, so it seems logical that we should encode the stack pebbles as $\pi'(k) = \text{pebenc}_t(\pi(m+k), \text{left}(\pi, m))$. Note that the fact that pebbles are stacked will ensure that the first m pebbles, $\text{left}(\pi, m)$, will not change as long as there is a pebble $m+k > m$ on the stack, and that will ensure that the encoding of such a pebble $m+k$ will not change during its "lifetime".

Finally, a word about states. Because we will encode everything in the tree, we have chosen to keep the states the same in the simulated and the simulating ptts. This is not a necessity: it is very well possible to define more complex encodings where the state space is transformed as well. However, in our case, transforming the state space would serve no other purpose than to complicate things; as we discussed in Section 4.3.1, a state space transformation can only change the total configuration space size by a constant factor, while we need an increase linear in the size of the input tree to encode even a single pebble location.

Summarizing this mapping:

DEFINITION 4.4.2. Given ptts $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and $M' = (n-m, (\Sigma, \Phi'), (\Delta, \Gamma), Q, q_0, R')$ where $m \in [1, \text{fg}(M)]$, a total function $\tau : R_{\Sigma,\Phi} \to R_{\Sigma,\Phi'}$, a tree $t \in R_{\Sigma,\Phi}$, and a total injective function $\text{pebenc}_t : V_t \times V_t^{\leq m} \to V_{\tau(t)}$, the input configuration mapping $\text{cic}_t : \text{IC}_{n,t} \to \text{IC}_{n-m,\tau(t)}$ is defined as

$$\text{cic}_t((u, \pi)) = (\text{pebenc}_t(u, \text{left}(\pi, m)), \text{stackenc}_t(\pi))$$

with $|\text{stackenc}_t(\pi)| = \max(|\pi| - m, 0)$ and $\text{stackenc}_t(\pi)(k) = \text{pebenc}_t(\pi(m+k), \text{left}(\pi, m))$ (for all $k \in [1, |\text{stackenc}_t(\pi)|]$). Furthermore, the configuration mapping $\text{cc}_t : C_{M,t} \to C_{M',\tau(t)}$ is defined as

$$\text{cc}_t(\langle q, (u, \pi)\rangle) = \langle q, \text{cic}_t((u, \pi))\rangle.$$

4.4.2. The Transformed Tree. We are now ready to define what the transformed tree $\tau(t)$ will look like. Let us begin by defining $\text{pebenc}_t : V_t \times V_t^{\leq m} \to V_{\tau(t)}$ as

$$\text{pebenc}_t(u, \pi) = u_\pi$$

where u_π is simply an alternative way of writing (u, π). This definition for pebenc_t is extremely simple, and it makes it immediately clear that pebenc_t, cic_t, and cc_t are total and injective. In the definition of $\tau(t)$, the encoding of the nodes is not really an issue. The only other things that we can state about the nodes of $\tau(t)$ is that $V_{\tau(t)} = \{u_\pi \mid u \in V_t \text{ and } \pi \in V_t^{\leq m}\}$, and that $\nu_{\tau(t)}(u_\pi) = \nu_t(u)$; this means that pebenc_t is even a bijection. It is the *edges* between the nodes that matter, because

they determine whether the transformed tree can be used for stepwise simulation.[2] The main constraint that the edges of $\tau(t)$ have to satisfy is that they must support *input instruction mapping*: we must be able to simulate every input instruction that can be executed by M on t on the transformed tree with a single input instruction. We choose the following encodings for the input instructions, immediately specifying the edges that these encodings require in $\tau(t)$:

- The go_ϕ ($\phi \in \Phi$) instruction is mapped to the same instruction. This means that we must have $\Phi' \supseteq \Phi$, and if $\text{go}_\phi(u, \pi) = (\phi(u), \pi)$ (with $\pi \in V_t^{\leq m}$), then we must have $\text{go}_\phi(u_\pi, \text{stackenc}_t(\pi)) = (\phi(u_\pi), \text{stackenc}_t(\pi))$. In other words, if $\phi(u) = u'$ in t, then we must have $\phi(u_\pi) = u'_\pi$ in $\tau(t)$. The set of required labeled edges in $\epsilon_{\tau(t)}$ is $\{((u_\pi, u'_\pi), \phi) \mid ((u, u'), \phi) \in \epsilon_t \text{ and } \pi \in V_t^{\leq m}\}$.
- Dropping/lifting a pebble with number $i \geq m + 1$ is encoded as the same drop/lift instruction, but for the pebble with number $i - m$. This requires no edges in $\epsilon_{\tau(t)}$.
- Dropping/lifting a pebble with a number $i \leq m$ requires moving from a node u_π to a node $u_{\pi u}$ (for dropping) and back (for lifting). Because we can only use a single instruction to simulate the dropping/lifting instruction, the encoded nodes before and after the instruction need to be directly connected. We introduce two new edge selectors $\text{lift}, \text{drop} \in \Phi'$ to connect them. The drop and lift instructions are then translated as go_{drop} and go_{lift}, respectively. The set of labeled edges that this encoding requires in $\epsilon_{\tau(t)}$ is $\{((u_\pi, u_{\pi u}), \text{drop}) \mid u \in V_t \text{ and } \pi \in V_t^{<m}\} \cup \{((u_{\pi u}, u_\pi), \text{lift}) \mid u \in V_t \text{ and } \pi \in V_t^{<m}\}$.
- The stay instruction maps to itself. No edges in $\epsilon_{\tau(t)}$ are required for this.

These edges are all that are required, so we specify that $\tau(t)$ contains exactly these required edges. This leaves us with the following definition:

DEFINITION 4.4.3. For given node and edge label alphabets Σ, Φ and encoded pebble count $m \geq 1$, $\tau : R_{\Sigma,\Phi} \to R_{\Sigma,\Phi'}$ where $\Phi' = \Phi \cup \{\text{drop}, \text{lift}\}$, is a total function.

[2]This is very different from Engelfriet and Maneth's decomposition proof [**13**], where the node encoding *was* the major issue. The reason for this difference is that we have adopted a graph-based tree model, which naturally includes the concept of node identity, whereas Engelfriet and Maneth used a tree model based on terms over ranked alphabets. In such a tree model, nodes have no separate identity, and instead are identified by their paths from the root; this makes node encoding naturally cumbersome, as it tightly binds node encoding together with the creation of the complete structure of the encoded tree. The graph-based tree model that we use is much better suited for this proof.

It is defined for $t \in R_{\Sigma,\Phi}$ as $\tau(t) = \left(\left(\nu_{\tau(t)}, \epsilon_{\tau(t)}\right), \text{root}_{\tau(t)}\right)$, where

$$\begin{aligned}
\nu_{\tau(t)} &= \bigcup_{\pi \in V_t^{\leq m}} \nu_{\tau_\pi(t)} \\
\epsilon_{\tau(t)} &= \left(\bigcup_{\pi \in V_t^{\leq m}} \epsilon_{\tau_\pi(t)} \right) \cup \epsilon_{\tau(t),\text{droplift}} \\
\nu_{\tau_\pi(t)} &= \{(u_\pi, \nu_t(u)) \mid u \in V_t\} \\
\epsilon_{\tau_\pi(t)} &= \{((u_\pi, u'_\pi), \phi) \mid ((u, u'), \phi) \in \epsilon_t\} \\
\epsilon_{\tau(t),\text{droplift}} &= \{((u_\pi, u_{\pi u}), \text{drop}) \mid u \in V_t \text{ and } \pi \in V_t^{<m}\} \cup \\
&\quad \{((u_{\pi u}, u_\pi), \text{lift}) \mid u \in V_t \text{ and } \pi \in V_t^{<m}\} \\
\text{root}_{\tau(t)} &= (\text{root}_t)_\lambda
\end{aligned}$$

Note that this choice of $\text{root}_{\tau(t)}$ ensures that cic_t maps the initial input configuration of M on t, (root_t, λ), to the initial input configuration of M' on $\tau(t)$, $\left(\text{root}_{\tau(t)}, \lambda\right)$. $\tau(t)$ is comprised of disjoint trees $\tau_\pi(t) = \left(\nu_{\tau_\pi(t)}, \epsilon_{\tau_\pi(t)}\right)$, plus the edges in $\epsilon_{\tau(t),\text{droplift}}$. Note that each of the component trees $\tau_\pi(t)$ is isomorphic with t (in the non-rooted sense) by an isomorphism $h_{t\to\tau_\pi(t)} : V_t \to V_{\tau_\pi(t)}$, $h_{t\to\tau_\pi(t)}(w) = w_\pi$; clearly, $h_{t\to\tau_\pi(t)}$ is also an injective homomorphism from t to $\tau(t)$. Note also that all of the edges in $\epsilon_{\tau(t),\text{droplift}}$ run between nodes from distinct components $\tau_\pi(t)$ and $\tau_{\pi u}(t)$, which implies that for each $\pi \in V_t^{\leq m}$, $\tau_\pi(t)$ is an induced subgraph of $\tau(t)$.

LEMMA 4.4.4. *$\tau(t)$ is a rooted tree.*

PROOF. We need to prove that $\tau(t)$ is a connected, acyclic graph. As for proving that it is a graph, it is easy to see that $\tau(t)$ is finite and nonempty if t is finite and nonempty, which it is. What remains to be shown is that $\tau(t)$ is connected, acyclic, that all edges have a reverse, and that the outgoing edge labels of a node are distinct. For starters, it is clear that each of the components $\tau_\pi(t)$ is acyclic and connected, that all of their internal edges have a reverse and that all of their nodes have distinct outgoing edges to nodes within the same component. We now need to prove that the edges that run over component boundaries do not create cycles, that they make the whole of $\tau(t)$ connected, that they all have reverses, and that they do not add duplicate outgoing edges. These are the proofs for these properties:

- *Reverse edges.* It is clear from the definition of $\epsilon_{\tau(t),\text{droplift}}$ that there is always a *drop* edge in the opposite direction of a *lift* edge and vice versa, so these edges always have a reverse.
- *No duplicate edges.* The *lift* and *drop* edges do not occur within the component trees, so only duplicates within $\epsilon_{\tau(t),\text{droplift}}$ need to be considered. Assume a

node u_π has two outgoing *drop* edges. Then both those edges run to $u_{\pi u}$, as all *drop* edges generated by $\epsilon_{\tau(t),\text{droplift}}$ run from a node u_π to a node $u_{\pi u}$. But $\epsilon_{\tau(t),\text{droplift}}$ is a set, so there cannot be two such edges. A similar argument can be made for *lift* edges.

- *Connectedness.* Any component tree $\tau_\pi(t)$ with $\pi \neq \lambda$ is connected to component tree $\tau_{\text{left}(\pi,|\pi|-1)}$ by edge $\left(\left(\pi(|\pi|)_\pi, \pi(|\pi|)_{\text{left}(\pi,|\pi|-1)}\right), \text{lift}\right) \in \epsilon_{\tau(t),\text{droplift}}$. Note that if $\pi \neq \lambda$ then $\pi = \text{left}(\pi, |\pi|-1) \cdot \pi(|\pi|)$. By induction, any component tree $\tau_\pi(t)$ is therefore connected to the root component $\tau_\lambda(t)$, and as all of the edges in $\tau(t)$ have reverses, any component tree $\tau_\pi(t)$ is connected to any other component tree $\tau_{\pi'}(t)$.
- *Acyclicity.* For $\pi \in V_t^{\leq m}$, let $\tau_{\pi+}(t)$ be defined as the $\left\{u_{\pi\pi'} \mid u \in V_t,\ \pi\pi' \in V_t^{\leq m}\right\}$-induced subgraph of $\tau(t)$. Obviously, $\tau_{\lambda+}(t) = \tau(t)$. We use induction on the length of π to prove that $\tau_{\lambda+}(t)$ is acyclic.
  Base ($|\pi| = m$): $\tau_{\pi+}(t) = \tau_\pi(t)$, which is a tree and therefore acyclic.
  Induction: Let $\pi \in V_t^{<m}$, and assume $\tau_{\pi u+}(t)$ is acyclic for all $u \in V_t$. Now consider $\tau_{\pi+}(t)$. Clearly,

$$\tau_{\pi+}(t) \quad = \quad \left(\bigcup_{u \in V_t} \tau_{\pi u+}(t)\right) \cup \tau_\pi(t) \cup (\emptyset, E)$$

  where $\cup$ indicates coordinatewise union, and where

$$E = \{((u_\pi, u_{\pi u}), \text{drop}) \mid u \in V_t\} \cup \{((u_{\pi u}, u_\pi), \text{lift}) \mid u \in V_t\}.$$

  It is clear that E adds exactly *one* edge to each component $\tau_{\pi u+}(t)$, thus connecting each of the components to $\tau_\pi(t)$. Now assume that there is a cycle in $\tau_{\pi+}(t)$, and that it contains a node in a component $\tau_{\pi u+}(t)$. By assumption, $\tau_{\pi u+}(t)$ is acyclic, so the cycle has to use an edge that leaves $\tau_{\pi u+}(t)$ *and* a different edge to get back inside $\tau_{\pi u+}(t)$. However, $\tau_{\pi u+}(t)$ has only one new edge attached to it in $\tau_{\pi+}(t)$, so that is not possible. A cycle in $\tau_{\pi+}(t)$ can therefore not contain a node in any of the components $\tau_{\pi u+}(t)$. But then the cycle should contain only nodes from $\tau_\pi(t)$, which is also acyclic, and $\tau_{\pi+}(t)$ does not add any edges between nodes of $\tau_\pi(t)$. In conclusion, such a cycle cannot exist, and by induction, $\tau_{\lambda+}(t) = \tau(t)$ is acyclic.

□

4.4.3. The Encoding TWTT. As we now have a definition for $\tau(t)$, we can define a twtt A so that $\tau_A = \tau$. We will start by defining A, after which we will show that $\tau_A = \tau$. The general principle of A will be very simple: it walks outward along the edges of the tree, copying nodes as it goes along. At every node it passes, it also generates a "drop" edge for every node along the way. At the other end of the "drop"

edge, it simply starts walking outward again, copying the whole tree as it goes along. When the twtt reaches a subtree and m "drop" edges have already been generated, no more drop edges are generated.

DEFINITION 4.4.5. For given node label alphabet Σ, edge label alphabet Φ, and encoded pebble count $m \geq 1$, $A = (0, (\Sigma, \Phi), (\Sigma, \Phi'), Q_A, q_{0,A}, R_A)$ with

$$
\begin{aligned}
\Phi' &= \Phi \cup \{\text{drop}, \text{lift}\} \\
Q_A &= \{q_{\phi,i} \mid \phi \in \Phi \cup \{\text{lift}\} \text{ and } i \in [0,m]\} \\
q_{0,A} &= q_{\text{lift},0} \\
R_A &= \{\langle q_{\phi,i}, \psi_{\sigma,F,r}\rangle \to \iota_{\phi,i,\sigma,F,r} \mid \\
&\qquad \phi \in \Phi \cup \{\text{lift}\},\ i \in [0,m],\ \sigma \in \Sigma, \\
&\qquad F \subseteq \Phi, \text{ total function } r:\ F \to \Phi\} \\
\psi_{\sigma,F,r} &\equiv \exists x : \text{head}(x) \wedge (\text{lab}_\sigma(x) \wedge \\
&\qquad \left(\bigwedge_{\hat{\phi} \in F} \exists y : \left(\text{edge}_{\hat{\phi}}(x,y) \wedge \text{edge}_{r(\hat{\phi})}(y,x)\right)\right) \wedge \\
&\qquad \left(\bigwedge_{\hat{\phi} \in \Phi - F} \neg\exists y : \text{edge}_{\hat{\phi}}(x,y)\right)) \\
\iota_{\phi,i,\sigma,F,r} &= ((\nu_{\phi,i,\sigma,F,r},\ \epsilon_{\phi,i,\sigma,F,r}),\ \text{root}_{\phi,i,\sigma,F,r})
\end{aligned}
$$

where

$$
\begin{aligned}
V_{\phi,i,\sigma,F,r} &= \{\text{root}_{\phi,i,\sigma,F,r}\} \cup \left\{u_{\hat{\phi}} \mid \hat{\phi} \in F - \{\phi\}\right\} \cup \{u_{\text{drop}} \mid i < m\} \\
\nu_{\phi,i,\sigma,F,r} &= \{(\text{root}_{\phi,i,\sigma,F,r}, \sigma)\} \cup \\
&\quad \left\{\left(u_{\hat{\phi}},\ \left\langle q_{r(\hat{\phi}),i},\ \text{go}_{\hat{\phi}}\right\rangle\right) \mid \hat{\phi} \in F - \{\phi\}\right\} \cup \\
&\quad \{(u_{\text{drop}},\ \langle q_{\text{lift},i+1},\ \text{stay}\rangle) \mid i < m\} \\
\epsilon_{\phi,i,\sigma,F,r} &= \left\{\left(\left(\text{root}_{\phi,i,\sigma,F,r},\ u_{\hat{\phi}}\right),\ \hat{\phi}\right) \mid \hat{\phi} \in F - \{\phi\}\right\} \cup \\
&\quad \left\{\left(\left(u_{\hat{\phi}},\ \text{root}_{\phi,i,\sigma,F,r}\right),\ r(\hat{\phi})\right) \mid \hat{\phi} \in F - \{\phi\}\right\} \cup \\
&\quad \{((\text{root}_{\phi,i,\sigma,F,r},\ u_{\text{drop}}),\ \text{drop}) \mid i < m\} \cup \\
&\quad \{((u_{\text{drop}},\ \text{root}_{\phi,i,\sigma,F,r}),\ \text{lift}) \mid i < m\}
\end{aligned}
$$

The parameters of the rule set may be interpreted as follows. σ, F and r convey the tree structure at the node that is to be copied: σ is its node label, F is its set of outgoing edge labels, $r : F \to \Phi$ is a function that associates a reverse edge label with each of the outgoing edges. For every state $q_{\phi,i}$, there is exactly one rule for every possible input situation σ, F, r. Every such rule contains an instruction that creates an exact

copy of the input situation in the output tree, plus a *drop* edge. The parameters ϕ and i, stored in the state $q_{\phi,i}$, are used to remember what part of the tree copying process has already been done by the twtt. First of all, the i parameter remembers the number of *drop* edges that have already been crossed at the current location in the output tree. Basically, the invariant is maintained that if a node u_π is being generated, then it is labeled with a configuration having state $q_{\phi,i}$ with $i = |\pi|$. When the automaton generates another *drop* edge, i.e., when it generates a node $u_{\pi u}$ below a node u_π, the configuration labeling node $u_{\pi u}$ has an i that is one higher than the configuration that labeled u_π. Lastly, the ϕ parameter represents the "way back". When the automaton generates a new node u_π, labeling it with a configuration, then it already generates one of the outgoing edges of that node – the edge leading toward the part of the tree that was already copied. Of course, the subsequent actions of the automaton on u_π should not generate that edge *again*. This is where ϕ plays its part: it records in the configuration that labels u_π which of the outgoing edges of u_π has already been generated. This is reflected in the instruction tree's node set $V_{\phi,i,\sigma,F,r}$: it only contains nodes $u_{\hat{\phi}}$ for $\hat{\phi} \in F - \{\phi\}$, i.e., it does not generate an output node for the node behind the edge labeled ϕ, even though it exists in the input tree.

The definition of A clearly makes it look like a ptt, but it is important that we verify that it actually satisfies the conditions of Definition 3.2.1. We will do so in the following Lemma.

LEMMA 4.4.6. *A is a ptt.*

PROOF. The alphabets, states and pebble count are fine for any ptt; what we need to verify is that the rules satisfy Definition 3.2.1(6).

The first condition specified by item (6) is that a rule $r \in R_A$ should be of the form $\langle q, \psi\rangle \to \iota$ with $q \in Q_A$, ψ a condition of A, and $\iota \in I_{A,\psi}$ an instruction. It is easy to verify that in a rule $r \in R_A$, $r : \langle q_{\phi,i}, \psi_{\sigma,F,r}\rangle \to \iota_{\phi,i,\sigma,F,r}$, $\psi_{\sigma,F,r}$ is in fact a condition of A, as it is a closed MSO predicate that uses relation symbols $\left\{\text{edge}_{\hat{\phi}} \mid \hat{\phi} \in F \subseteq \Phi \text{ or } \hat{\phi} \in \Phi - F \subseteq \Phi\right\} \cup \{\text{head}, \text{lab}_\sigma\} \subseteq A_A$. Furthermore, $q_{\phi,i} \in Q_A = \{q_{\phi,i} \mid \phi \in \Phi \cup \{\text{lift}\} \text{ and } i \in [0,m]\}$, as the definition of R_A specifies $\phi \in \Phi \cup \{\text{lift}\}$ and $i \in [0,m]$. The condition $\iota \in I_{A,\psi} = R_{\Sigma,\Phi'}\left(\text{CI}_{A,\psi_{\sigma,F,r}}\right)$ is also satisfied, because ι is obviously a rooted tree, the edge labels used in ι are $\{\text{drop}, \text{lift}\} \cup \left\{\hat{\phi} \mid \hat{\phi} \in F - \{\phi\}\right\} \cup \left\{r\left(\hat{\phi}\right) \mid \hat{\phi} \in F - \{\phi\}\right\} \subseteq \Phi'$, root_ι is labeled $\sigma \in \Sigma$, there may be a node that is labeled $\langle q_{\text{lift},i+1}, \text{stay}\rangle$ which is clearly in $\text{CI}_{A,\psi_{\sigma,F,r}}$, and the remaining nodes are labeled $\left\langle q_{r(\hat{\phi}),i}, \text{go}_{\hat{\phi}}\right\rangle$ for a $\hat{\phi} \in F$, which is also in $\text{CI}_{A,\psi_{\sigma,F,r}}$, as we will show. First of all, $q_{r(\hat{\phi}),i} \in Q_A$ because $r\left(\hat{\phi}\right) \in \Phi \subseteq \Phi \cup \{\text{lift}\}$ and $i \in [0,m]$ by the definition

of R_A. Secondly, $\text{go}_{\hat{\phi}}$ is in $\text{II}_{A,\psi_{\sigma,F,r}}$, as $\psi_{\sigma,F,r}$ clearly implies that for any $\hat{\phi} \in F$, $\exists x : \exists y : \text{head}(x) \wedge \text{edge}_{\hat{\phi}}(x,y)$.

The second condition specified by Definition 3.2.1(6) is that there must exist a total function $\text{out}_{R_A} : Q_A \to \mathcal{P}(\Phi')$ that satisfies a number of conditions for each rule in R_A. Let $\text{out}_{R_A}(q_{\phi,i}) = \{\phi\}$. For each rule $r : \langle q_{\phi,i}, \psi_{\sigma,F,r}\rangle \to \iota_{\phi,i,\sigma,F,r}$ in R_A, out_{R_A} satisfies the conditions given by Definition 3.2.1(6) as follows :

(1) $\nu_{\iota_{\phi,i,\sigma,F,r}}\left(\text{root}_{\iota_{\phi,i,\sigma,F,r}}\right) \in \Sigma$, so this condition is satisfied trivially.
(2) For any $u \in V_{\iota_{\phi,i,\sigma,F,r}}$ with $\nu_{\iota_{\phi,i,\sigma,F,r}}(u) = \langle q_{\phi',i'}, \omega\rangle$, we have $\text{out}_{\iota_{\phi,i,\sigma,F,r}}(u) = \{\phi'\} \subseteq \text{out}_{R_A}(q_{\phi',i'}) = \{\phi'\}$.
(3) $\text{out}_{\iota_{\phi,i,\sigma,F,r}}\left(\text{root}_{\iota_{\phi,i,\sigma,F,r}}\right) \cap \text{out}_{R_A}(q_{\phi,i}) = ((F - \{\phi\}) \cup \{\text{drop} \mid i < m\}) \cap \{\phi\}$, and as $\phi \in \Phi \cup \{\text{lift}\}$ and therefore $\phi \neq \text{drop}$, and as $\phi \notin F - \{\phi\}$, this intersection is empty.

□

LEMMA 4.4.7. *Let $d \in R_{\Sigma,\Phi'}(C_{A,t})$. If $snt(\langle q_{lift,0}, (root_t, \lambda)\rangle) \Rightarrow^k_{A,t} d$, then $|V_{d,\Sigma}| = k$.*

PROOF. We will prove this by induction on k. For $k = 0$, $|V_{d,\Sigma}| = 0 = k$, as $\nu_d = \{(\text{root}_d, \langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle)\}$. Now assume the property holds for a given k, and that $\text{snt}(\langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle) \Rightarrow^k_{A,t} d_k \Rightarrow_{A,t} d_{k+1}$. Let $w \in V_k$ be the computation step node of computation step $d_k \Rightarrow_{A,t} d_{k+1}$. Then w is labeled with a configuration in $C_{A,t}$, and it is replaced in d_{k+1} by the output of an instruction execution. As all instructions occurring in rules of A have only the root labeled in Σ, $\left|V_{d_{k+1},\Sigma}\right| = |V_{d_k,\Sigma}| + 1$. By induction, the lemma holds for all k. □

LEMMA 4.4.8. *A is total and deterministic.*

PROOF. This statement is equivalent to saying that for any configuration $c = \langle q_{\phi,i}, (u,\pi)\rangle \in C_{A,t}$, exactly one rule $r \in R_A$ is applicable to c. Let $\langle q_{\phi,i}, (u,\pi)\rangle \in C_{A,t}$ be any configuration. Input configuration (u,π) satisfies condition $\psi_{\nu_t(u),\text{out}_t(u),r}$ with $r(\hat{\phi}) = \epsilon_t\left(\hat{\phi}(u), u\right)$ for $\hat{\phi} \in \text{out}_t(u)$. There exists a rule for any combination of a state $q_{\phi,i}$ and condition $\psi_{\sigma,F,r}$, so A is total. In addition, (u,π) satisfies no other conditions, as all of the conditions $\psi_{\sigma,F,r}$ are mutually exclusive (which can be easily verified). □

LEMMA 4.4.9. *Let $d \in R_{\Sigma,\Phi'}(C_{A,t})$. If $snt(\langle q_{lift,0}, (root_t, \lambda)\rangle) \Rightarrow^*_{A,t} d$, then there exists an injective edge homomorphism h from d to $\tau(t)$ that satisfies the following properties:*

(1) *For all $w \in V_{d,\Sigma}$, $\nu_d(w) = \nu_{\tau(t)}(h(w))$ and $out_d(w) = out_{\tau(t)}(h(w))$.*
(2) *For all $w \in V_{d,C_{A,t}}$, if $\nu_d(w) = \langle q_{\phi,i}, (u,\lambda)\rangle$, then*
   (a) *there exists a $\pi \in V_t^i$ so that $h(w) = u_\pi$.*
   (b) *$out_d(w) = \{\phi\}$, except when $w = root_d$, in which case $\phi = lift$, $i = 0$, and $out_d(w) = \emptyset$.*

PROOF. We will prove this property by induction on the length of the computation, k.

Base ($k = 0$)*:* If $\text{snt}\left(\langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle\right) \Rightarrow^0_{A,t} d$, then

$$d \cong \text{snt}\left(\langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle\right).$$

Let $h = \left\{\left(\text{root}_d, \text{root}_{\tau(t)}\right)\right\}$. This is a rooted-tree edge homomorphism, as $h\left(\text{root}_d\right) = \text{root}_{\tau(t)}$, and as no edges need to be matched because d contains only one node. Furthermore, h is trivially injective because it contains only one element. As for the properties of h:

(1) Holds trivially because $V_{d,\Sigma} = \emptyset$.
(2) We only have $\text{root}_d \in V_{d,C_{A,t}}$, and $\nu_d\left(\text{root}_d\right) = \langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle$.
   (a) We have $h\left(\text{root}_d\right) = \text{root}_{\tau(t)} = \left(\text{root}_t\right)_\lambda$, having encoded pebble stack $\lambda \in V^0_t$, which is what the property demands.
   (b) For root_d and $\nu_d\left(\text{root}_d\right) = \langle q_{\phi,i}, (u, \lambda)\rangle$ it is required that $\phi = \text{lift}$, $i = 0$, and $\text{out}_d\left(\text{root}_d\right) = \emptyset$, which is indeed the case.

Induction: Let $k \geq 0$, and assume the property holds for any k-length derivation

$$\text{snt}\left(\langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle\right) \Rightarrow^k_{A,t} d_k$$

with $d_k \in R_{\Sigma,\Phi'}\left(C_{A,t}\right)$. We can derive the property for derivation

$$\text{snt}\left(\langle q_{\text{lift},0}, (\text{root}_t, \lambda)\rangle\right) \Rightarrow^k_{A,t} d_k \Rightarrow_{A,t} d_{k+1}$$

of length $k+1$ as follows. Let $w_k \in V_k$ be the computation step node of step $d_k \Rightarrow_{A,t} d_{k+1}$, with $\nu_k(w_k) = \langle q_{\phi,i}, (u, \lambda)\rangle$, $u \in V_t$, and $q_{\phi,i} \in Q_A$. (Note that the pebble stack is always λ because A is a twtt.) The one and only applicable rule for this configuration is $\langle q_{\phi,i}, \psi\rangle \to \iota$ with $\psi = \psi_{\nu_t(u),\text{out}_t(u),r}$ and $\iota = \iota_{\phi,i,\nu_t(u),\text{out}_t(u),r}$, with $r\left(\hat{\phi}\right) = \epsilon_t(\hat{\phi}(u), u)$. We then have $d_{k+1} \cong d_k\left[w_k \leftarrow \iota\left((u, \lambda)\right)\right]$; we can assume without loss of generality that $d_{k+1} = d_k\left[w_k \leftarrow \iota\left((u, \lambda)\right)\right]$ and that $h_{d_k \to d_{k+1}}$ is an identity isomorphism, because it can be easily verified that if the property applies to a tree d, then it applies to any tree isomorphic to d as well. As π is always equal to λ in A-configurations, we will use the shorthand definition $\iota\left(u\right) = \iota\left((u, \lambda)\right)$ for any instruction $\iota \in I_A$ and node $u \in V_t$. By the induction assumption, there exists an injective edge homomorphism h_k from d_k to $\tau\left(t\right)$ that satisfies all of the properties specified by the Lemma. We will now show that there exists an edge homomorphism h_{k+1} from d_{k+1} to $\tau\left(t\right)$ that also satisfies all of these properties.

In order to construct h_{k+1}, we must provide corresponding nodes for the extra nodes that are in d_{k+1} but not in d_k. By property 2b of the induction assumption (and as

$\iota(u)$ has exactly one level of nodes below the root), this set can be described as

$$\left\{\hat{\phi}(w_k) \mid \hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\}\right\},$$

and we want to bind each of these $\hat{\phi}(w_k)$ to $\hat{\phi}(h_k(w_k))$. However, this can only be done if $\hat{\phi}(h_k(w_k))$ *exists* for all $\hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\}$. By property 2a of the induction assumption, $\nu_k(w_k) = \langle q_{\phi,i}, (u, \lambda)\rangle$ implies that $h_k(w_k) = u_\pi$ for some $\pi \in V_t^i$, so what we want to prove is that $\text{out}_{\tau(t)}(u_\pi) \supseteq \text{out}_{k+1}(w_k) - \{\phi\}$. We will, in fact, prove the following slightly stronger statement.

CLAIM. $\text{out}_{\tau(t)}(u_\pi) = \text{out}_{k+1}(w_k)$.

PROOF. From the definition of $\tau(t)$ we can derive:

$$\text{out}_{\tau(t)}(u_\pi) = \text{out}_t(u) \cup \{\text{drop} \mid |\pi| < m\} \cup \{\text{lift} \mid \pi(|\pi|) = u\}.$$

We know that because $d_{k+1} = d_k[w_k \leftarrow \iota(u)]$, $\text{out}_{k+1}(w_k) = \text{out}_k(w_k) \cup \text{out}_{\iota(u)}\left(\text{root}_{\iota(u)}\right)$. We can derive from the definition of $\iota(u)$ that

$$\text{out}_{\iota(u)}\left(\text{root}_{\iota(u)}\right) = (\text{out}_t(u) - \{\phi\}) \cup \{\text{drop} \mid i < m\}.$$

When $w_k = \text{root}_k$, then $\text{out}_k(w_k) = \emptyset$, $i = 0$, and $\phi = \text{lift}$ (by property 2b), which implies $\text{out}_{k+1}(w_k) = \text{out}_k(w_k) \cup \text{out}_{\iota(u)}\left(\text{root}_{\iota(u)}\right) = \text{out}_{\iota(u)}\left(\text{root}_{\iota(u)}\right)$. As $\phi = \text{lift}$ and $|\pi| = i = 0$, we know that $\{\text{lift} \mid \pi(|\pi|) = u\} = \emptyset$, and we have

$$\begin{aligned}
\text{out}_{\iota(u)}\left(\text{root}_{\iota(u)}\right) &= (\text{out}_t(u) - \{\phi\}) \cup \{\text{drop} \mid i < m\} \\
&= \text{out}_t(u) \cup \{\text{drop} \mid |\pi| < m\} \\
&= \text{out}_t(u) \cup \{\text{drop} \mid |\pi| < m\} \cup \{\text{lift} \mid \pi(|\pi|) = u\} \\
&= \text{out}_{\tau(t)}(u_\pi)
\end{aligned}$$

which is what we wanted to show. Now consider $w_k \neq \text{root}_k$. Then $\text{out}_k(w_k) = \{\phi\}$ (by property 2b), and (as explained below):

$$\begin{aligned}
\text{out}_{k+1}(w_k) &= \text{out}_k(w_k) \cup \text{out}_{\iota(u)}\left(\text{root}_{\iota(u)}\right) \\
&= \{\phi\} \cup ((\text{out}_t(u) - \{\phi\}) \cup \{\text{drop} \mid i < m\}) \\
&= \{\text{lift} \mid \phi = \text{lift}\} \cup \text{out}_t(u) \cup \{\text{drop} \mid |\pi| < m\} \\
&= \{\text{lift} \mid \pi(|\pi|) = u\} \cup \text{out}_t(u) \cup \{\text{drop} \mid |\pi| < m\} \\
&= \text{out}_{\tau(t)}(u_\pi)
\end{aligned}$$

Note first that we use the fact that $\{\phi\} \cup (\text{out}_t(u) - \{\phi\}) = \text{out}_t(u)$ iff $\phi \in \text{out}_t(u)$, which is the case iff $\phi \neq \text{lift}$. The case that $\phi = \text{lift}$ is covered by $\{\text{lift} \mid \phi = \text{lift}\}$. This is the case exactly when $h_k(w_k) = u_\pi$ has $\pi(|\pi|) = u$, for the following reasons. First of all, ϕ labels the first outgoing edge on the path from w_k to root_k, as w_k has no other

outgoing edge labels. Secondly, $h_k(\text{root}_k) = \text{root}_{\tau(t)}$. As h_k is injective, this implies that ϕ also labels the first outgoing edge on the path from $h_k(w_k) = u_\pi$ to $\text{root}_{\tau(t)}$. Now observe that if node u_π of $\tau(t)$ has an outgoing edge labeled *lift*, then this edge is always the first edge on the path from that node to $\text{root}_{\tau(t)}$, as this path leads through the *lift* edge to $u_{\text{left}(\pi,|\pi|-1)}$, through the component $\tau_{\text{left}(\pi,|\pi|-1)}(t)$ to the next pebble at $\pi(|\pi|-1)_{\text{left}(\pi,|\pi|-1)}$, through the lift edge to $\pi(|\pi|-1)_{\text{left}(\pi,|\pi|-2)}$, etcetera until the pebble stack is empty, and then to $(\text{root}_t)_\lambda$. Summarizing, $h_k(w_k) = u_\pi$ has a *lift* edge iff $\pi(|\pi|) = u$; the *lift* edge will be the first edge on the path to $\text{root}_{\tau(t)}$, as is ϕ, so $\phi = \text{lift}$ iff $\pi(|\pi|) = u$. This concludes the proof of the Claim. □

Having shown that the outgoing edge labels of w_k in d_{k+1} match those of $h_k(w_k)$ in $\tau(t)$, we can now define h_{k+1} as follows:

$$h_{k+1} \quad = \quad h_k \cup \left\{ \left(\hat{\phi}_{\langle k+1 \rangle}(w_k), \hat{\phi}_{\langle \tau(t) \rangle}(h_k(w_k)) \right) \mid \hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\} \right\}.$$

It is obvious that h_{k+1} is a total function from V_{k+1} to $V_{\tau(t)}$ because, as observed before, $V_{k+1} - V_k = \left\{ \hat{\phi}_{\langle k+1 \rangle}(w_k) \mid \hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\} \right\}$. Furthermore, we can show that h_{k+1} is injective using the following reasoning. Of course, h_k is injective, so for any two nodes $w, w' \in V_k \subseteq V_{k+1}$, $w \neq w'$ implies $h_{k+1}(w) = h_k(w) \neq h_k(w') = h_{k+1}(w')$. Furthermore, for any nodes $w = \hat{\phi}_{\langle k+1 \rangle}(w_k)$ and $w' = \hat{\phi}'_{\langle k+1 \rangle}(w_k)$, with $\hat{\phi}, \hat{\phi}' \in \text{out}_{k+1}(w_k) - \{\phi\}$, $w \neq w'$ obviously implies $\hat{\phi} \neq \hat{\phi}'$ and therefore $h_{k+1}(w) = \hat{\phi}_{\langle \tau(t) \rangle}(h_k(w_k)) \neq \hat{\phi}'_{\langle \tau(t) \rangle}(h_k(w_k)) = h_{k+1}(w')$. Lastly, consider a pair of nodes $w \in V_k \subseteq V_{k+1}$ and $w' = \hat{\phi}_{\langle k+1 \rangle}(w_k)$, $\hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\}$. We have $h_{k+1}(w) = h_k(w) \in h_k(V_k)$. Assume that $h_{k+1}(w') = h_{k+1}(w)$, i.e., $\hat{\phi}_{\langle \tau(t) \rangle}(h_k(w_k)) = h_k(w)$. Clearly $w \neq w_k$. Then $\text{path}_k(w_k, w) = w_k \phi_{\langle k \rangle}(w_k) \cdots w$ in d_k (as the only outgoing edge label of w_k in d_k is ϕ), and because h_k is an injective homomorphism, $\text{path}_{\tau(t)}(h_k(w_k), h_k(w)) = h_k(w_k)\, \phi_{\langle \tau(t) \rangle}(h_k(w_k)) \cdots h_k(w)$ in $\tau(t)$. But if our assumption that $\hat{\phi}_{\langle \tau(t) \rangle}(h_k(w_k)) = h_k(w)$ is true, then if $\left|\text{path}_{\tau(t)}(h_k(w_k), h_k(w))\right| \neq 2$, we have $(h_k(w_k), h_k(w)) \in E_{\tau(t)}$ and so $\text{path}_{\tau(t)}(h_k(w_k), h_k(w))$ is a *cycle* in $\tau(t)$, which contradicts the fact that $\tau(t)$ is a tree. On the other hand, if $\left|\text{path}_{\tau(t)}(h_k(w_k), h_k(w))\right| = 2$, then $\phi_{\langle \tau(t) \rangle}(h_k(w_k)) = h_k(w)$, which contradicts our assumption that $\hat{\phi}_{\langle \tau(t) \rangle}(h_k(w_k)) = h_k(w)$, as $\hat{\phi} \neq \phi$. We have now verified all combinations from the domain of h_{k+1}, and we can conclude that h_{k+1} is in fact injective.

Next we show properties 1, 2a and 2b for h_{k+1}. As for all nodes $w \in V_k$ with $w \neq w_k$, $\text{out}_k(w) = \text{out}_{k+1}(w)$ and $\nu_k(w) = \nu_{k+1}(w)$, it is obvious that the properties hold for all nodes in $V_k - \{w_k\}$. We will therefore need to verify the properties only for w_k and for all $\hat{\phi}_{\langle k+1 \rangle}(w_k)$ with $\hat{\phi} \neq \phi$.

(1) Let us begin by checking the labels. The only considered output node with a label in Σ is w_k, which has $\nu_{k+1}(w_k) = \nu_{\iota(u)}\left(\text{root}_{\iota(u)}\right) = \nu_t(u)$, whereas $h_{k+1}(w_k) = h_k(w_k) = u_\pi$ has label $\nu_{\tau(t)}(u_\pi) = \nu_t(u)$, which is a match. Now let us look at the outgoing edge labels. Again, we only need to consider w_k. Using the Claim, we have $\text{out}_{k+1}(w_k) = \text{out}_{\tau(t)}(u_\pi) = \text{out}_{\tau(t)}(h_k(w_k)) = \text{out}_{\tau(t)}(h_{k+1}(w_k))$.

(2) (a) As discussed earlier, we only need to consider w_k and all $\hat{\phi}_{\langle k+1\rangle}(w_k)$ with $\hat{\phi} \neq \phi$. Because $\nu_{k+1}(w_k) \notin C_{A,t}$, that leaves the $\hat{\phi}_{\langle k+1\rangle}(w_k)$, $\hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\}$. We can derive the labels of these nodes from the definition of $\iota(u)$, as $d_{k+1} = d_k[w_k \leftarrow \iota(u)]$. If $\hat{\phi} \neq \text{drop}$, we have

$$\nu_{k+1}\left(\hat{\phi}_{\langle k+1\rangle}(w_k)\right) = \left\langle q_{r\left(\hat{\phi}\right),i}, \left(\hat{\phi}_{\langle t\rangle}(u), \lambda\right)\right\rangle.$$

As $h_{k+1}\left(\hat{\phi}_{\langle k+1\rangle}(w_k)\right) = \hat{\phi}_{\langle\tau(t)\rangle}(h_k(w_k)) = \hat{\phi}_{\langle\tau(t)\rangle}(u_\pi) = \left(\hat{\phi}_{\langle t\rangle}(u)\right)_\pi$, we have $\pi \in V_t^i$ (taken directly from the induction assumption). However, if $\hat{\phi} = \text{drop}$, then we have $\nu_{k+1}\left(\text{drop}_{\langle k+1\rangle}(w_k)\right) = \langle q_{\text{lift},i+1}, (u, \lambda)\rangle$. As $h_{k+1}\left(\text{drop}_{\langle k+1\rangle}(w_k)\right) = \text{drop}_{\langle\tau(t)\rangle}(h_k(w_k)) = \text{drop}_{\langle\tau(t)\rangle}(u_\pi) = u_{\pi u}$, we have $\pi u \in V_t^{i+1}$ (as the induction assumption provides $\pi \in V_t^i$, and $u \in V_t$). This completes the proof that this property holds for h_{k+1}.

(b) Like with the preceding property, we only need to consider the $\hat{\phi}_{\langle k+1\rangle}(w_k)$, $\phi \in \text{out}_{k+1}(w_k) - \{\phi\}$. Consider such a node $\hat{\phi}_{\langle k+1\rangle}(w_k)$. It cannot be root_{k+1}, as root_{k+1} is labeled in Σ in all but the initial intermediate output tree. Now let us first consider the case that $\hat{\phi} \neq \text{drop}$. In the definition of $\iota(u)$, $\hat{\phi}_{\langle\iota(u)\rangle}\left(\text{root}_{\iota(u)}\right) = u_{\hat{\phi}}$ with reverse edge label $\epsilon_{\iota(u)}\left(u_{\hat{\phi}}, \text{root}_{\iota(u)}\right) = r\left(\hat{\phi}\right)$. The derived node $\hat{\phi}_{\langle k+1\rangle}(w_k)$ has label $\left\langle q_{r\left(\hat{\phi}\right),i}, \left(\hat{\phi}_{\langle t\rangle}(u), \lambda\right)\right\rangle$. It is easily understood that node $u_{\hat{\phi}}$ has only the "reverse edge" $r\left(\hat{\phi}\right)$ as its outgoing edge label, so $\text{out}_{\iota(u)}\left(u_{\hat{\phi}}\right) = \left\{r\left(\hat{\phi}\right)\right\}$, and as $u_{\hat{\phi}}$ is not the root of $\iota(u)$, the corresponding node $\hat{\phi}_{\langle k+1\rangle}(w_k)$ must also have $\text{out}_{k+1}\left(\hat{\phi}_{\langle k+1\rangle}(w_k)\right) = \left\{r\left(\hat{\phi}\right)\right\}$, which matches its label. Now consider the case that $\hat{\phi} = \text{drop}$. In this case, $\iota(u)$ defines $\hat{\phi}_{\langle\iota(u)\rangle}\left(\text{root}_{\iota(u)}\right) = u_{\text{drop}}$, with $\text{out}_{\iota(u)}(u_{\text{drop}}) = \{\text{lift}\}$. On the other hand, we have $\nu_{k+1}\left(\text{drop}_{\langle k+1\rangle}(w_k)\right) = \langle q_{\text{lift},i+1}, \text{stay}\rangle$, and $\text{out}_{k+1}\left(\text{drop}_{\langle k+1\rangle}(w_k)\right) = \{\text{lift}\}$. This proves that this property holds for h_{k+1}.

What remains is that we need to show that h_{k+1} is an edge homomorphism from d_{k+1} to $\tau(t)$. Clearly, all edges in d_{k+1} that correspond to edges in d_k match up perfectly with edges in $\tau(t)$, because h_{k+1} fully contains h_k. This means that we will only need to look at the outgoing edges $\hat{\phi} \in \text{out}_{k+1}(w_k) - \{\phi\}$ to establish h_{k+1} as an edge homomorphism. Using the first property we derived for h_{k+1}, we already

know that all of the outgoing edges are present. Also, we know that h_{k+1} maps these edges as $\left(\hat{\phi}_{\langle k+1\rangle}(w_k), \hat{\phi}_{\langle \tau(t)\rangle}(h_k(w_k))\right)$. By this definition, the outgoing edge labels obviously match up; in order to establish the homomorphism we still need to check that $\epsilon_{k+1}\left(\hat{\phi}_{\langle k+1\rangle}(w_k), w_k\right) = \epsilon_{\tau(t)}\left(\hat{\phi}_{\langle \tau(t)\rangle}(u_\pi), u_\pi\right)$. This is obviously the case when $\hat{\phi} = \text{drop}$, as $\epsilon_{k+1}\left(\text{drop}_{\langle k+1\rangle}(w_k), w_k\right) = \text{lift} = \epsilon_{\tau(t)}\left(\text{drop}_{\langle \tau(t)\rangle}(u_\pi), u_\pi\right)$. For $\hat{\phi} \neq \text{drop}$ it is true as well: $\epsilon_{k+1}\left(\hat{\phi}_{\langle k+1\rangle}(w_k), w_k\right) = r\left(\hat{\phi}\right) = \epsilon_t(\hat{\phi}_{\langle t\rangle}(u), u)$, and $\epsilon_{\tau(t)}\left(\hat{\phi}_{\langle \tau(t)\rangle}(u_\pi), u_\pi\right) = \epsilon_{\tau(t)}\left(\left(\hat{\phi}_{\langle t\rangle}(u)\right)_\pi, u_\pi\right) = \epsilon_t\left(\hat{\phi}_{\langle t\rangle}(u), u\right)$. Therefore, h_{k+1} is a homomorphism, and by induction the Lemma is proven. □

THEOREM 4.4.10. *For all trees $t \in R_{\Sigma,\Phi}$, $\tau_A(t) = \tau(t)$.*

PROOF. A is total and deterministic (by Lemma 4.4.8), so it will always successfully compute at most one output tree (modulo isomorphism). We will now prove that this tree is $\tau(t)$. First of all, the computation will proceed for at least $\left|V_{\tau(t)}\right|$ steps. When only $k < \left|V_{\tau(t)}\right|$ computation steps have been applied, there must be a node in the intermediate output tree d that is labeled with a configuration, by the following reasoning. Let $h_{d\to\tau(t)}$ be the injective edge homomorphism from d to $\tau(t)$, which exists by Lemma 4.4.9. There are $k = |V_{d,\Sigma}| < \left|V_{\tau(t)}\right|$ nodes that have labels in Σ (by Lemma 4.4.7); for every one of these nodes, the set of output edges matches those of the corresponding node in $\tau(t)$ (by Lemma 4.4.9(1)). If each of these outgoing edges would lead to one of the nodes in $V_{d,\Sigma}$, then this would also be true for the corresponding edges in $\tau(t)$. As $\left|h_{d\to\tau(t)}(V_{d,\Sigma})\right| = |V_{d,\Sigma}| = k < \left|V_{\tau(t)}\right|$, that would imply that $\tau(t)$ is not connected. However, $\tau(t)$ is a tree and is therefore connected, so we can only conclude that at least one of the outgoing edges of nodes in $V_{d,\Sigma}$ leads to a node not in $V_{d,\Sigma}$. This node must then be labeled with a configuration in $C_{A,t}$. Because A is total, there will always be an applicable rule for that configuration, so we can draw the conclusion that the computation will not complete in less than $\left|V_{\tau(t)}\right|$ computation steps. Now consider a computation of $k = \left|V_{\tau(t)}\right|$ steps that yields an intermediate output tree d. Then $|V_d| \geq |V_{d,\Sigma}| = k = \left|V_{\tau(t)}\right|$ by Lemma 4.4.7. On the other hand, $h_{d\to\tau(t)}$ is an injective edge homomorphism, so $|V_d| \leq \left|V_{\tau(t)}\right|$. Together with $|V_d| \geq \left|V_{\tau(t)}\right|$ this implies $\left|V_{\tau(t)}\right| = |V_d|$, i.e., $h_{d\to\tau(t)}$ is an edge *isomorphism*. All of the $|V_{d,\Sigma}| = k = |V_d|$ nodes in d have labels in Σ, which match their counterparts in $\left|V_{\tau(t)}\right|$ completely, according to property 1 of Lemma 4.4.9. This implies that $h_{d\to\tau(t)}$ is a full isomorphism, and $d \cong \tau(t)$. □

4.4.4. The Simulating $(n-m)$-ptt. In order to complete the decomposition, we must now define the simulating ptt $M' = (n-m, (\Sigma, \Phi'), (\Delta, \Gamma), Q, q_0, R')$ such that M' performs a stepwise simulation of M using a tree transformed by τ. In preparation for this definition, we require the following definitions and lemmas.

DEFINITION 4.4.11. Let C be a set of relation symbols, and let $C \supseteq A_{\Sigma,\Phi'}$ for some node label alphabet Σ and edge label alphabet Φ'. Furthermore, let $\Phi \subseteq \Phi'$, and let x and y be node variables. We define the C-MSO predicate $\text{path}_\Phi(x,y)$ as follows, using C-MSO predicates $\psi_{1..5}$ as intermediate formulas for presentation purposes, because of the length of the complete formula:

$$\begin{aligned}
\psi_1 &\equiv (\neg\text{eq}(a,b) \wedge \text{uedge}_\Phi(a,p) \wedge \text{uedge}_\Phi(p,b)) \\
\psi_2 &\equiv \exists a,b \in P : \psi_1 \\
\psi_3 &\equiv \forall p \in P : (\neg\text{eq}(p,x) \wedge \neg\text{eq}(p,y)) \implies \psi_2 \\
\psi_4 &\equiv (\exists b \in P : \text{uedge}_\Phi(x,b)) \wedge \psi_3 \\
\psi_5 &\equiv x \in P \wedge y \in P \wedge (\text{eq}(x,y) \vee \psi_4) \\
\text{path}_\Phi(x,y) &\equiv \exists P : \psi_5
\end{aligned}$$

where we make use of the auxiliary predicate $\text{uedge}_\Phi(x,y) \equiv (\text{edge}_\Phi(x,y) \wedge \text{edge}_\Phi(x,y))$, which indicates the existence of an undirected edge labeled in Φ in both directions.

The interpretation of $\text{path}_\Phi(x,y)$ is that "there exists a path from x to y whose edges are all labeled in Φ". The formulation of $\text{path}_\Phi(x,y)$ is such that it is valid for any acyclic graph. We will now state this interpretation formally, and provide proof that it is correct.

LEMMA 4.4.12. *Let Σ be a node label alphabet and let Φ' be an edge label alphabet. Let $g \in G_{\Sigma,\Phi'}$ be an acyclic graph over Σ and Φ', let $\Phi \subseteq \Phi'$, and let x and y be node variables. Then $\text{path}_\Phi(x,y)$ is true for binding $(b_{F_{node}}, b_{F_{set}}, b)$ with $b \supseteq b_{A_{\Sigma,\Phi'},g}$ if and only if there is a path $v_1 v_2 \cdots v_k$ from $v_1 = b_{F_{node}}(x)$ to $v_k = b_{F_{node}}(y)$ with $\epsilon_g(v_i, v_{i+1}) \in \Phi$ and $\epsilon_g(v_{i+1}, v_i) \in \Phi$ for $i \in [1, k-1]$.*

PROOF. In this proof, we will use the same subpredicates $\psi_{1..5}$ that were used in the definition of $\text{path}_\Phi(x,y)$ to identify the components of $\text{path}_\Phi(x,y)$.

($\Rightarrow$) Assume $\text{path}_\Phi(x,y)$ is true with binding $B = (b_{F_{\text{node}}}, b_{F_{\text{set}}}, b)$, $b \supseteq b_{A_{\Sigma,\Phi'},g}$. Let $b_x = b_{F_{\text{node}}}(x)$ and $b_y = b_{F_{\text{node}}}(y)$. Then there exists a node set $b_P \in \mathcal{P}(V_g)$ so that ψ_5 is true with binding $B_5 = (b_{F_{\text{node}}}, b_{F_{\text{set}}} \oplus (P, b_P), b)$. Then by conjuncts $x \in P$ and $y \in P$ of ψ_5, $b_x \in b_P$ and $b_y \in b_P$. We will now construct a path $v_1 v_2 \cdots v_k$ from $v_1 = b_x$ to $v_k = b_y$ where $v_i \in b_P$ for $i \in [1,k]$, and where $\epsilon_g(v_i, v_{i+1}) \in \Phi$ and $\epsilon_g(v_{i+1}, v_i) \in \Phi$ for $i \in [1, k-1]$. We start with $v_1 = b_x$. By conjunct $(\text{eq}(x,y) \vee \psi_4)$ of ψ_5, at least one of $\text{eq}(x,y)$ and ψ_4 is true with binding B_5. If $\text{eq}(x,y)$ is true, then we are done: $v_1 = b_x = b_y$ is a path from b_x to b_y, and $v_1 \in b_P$. If not, then we know that ψ_4 must be true with binding $B_4 = B_5$, which implies that conjunct $\exists b \in P : \text{uedge}_\Phi(x,b)$ is true, so there exists a $v_2 \in b_P$ so that $\epsilon_g(v_1, v_2) \in \Phi$ and $\epsilon_g(v_2, v_1) \in \Phi$. Note that

$v_2 \neq v_1$ because g is acyclic (and so $v_1 v_2$ is a path). Having these first two nodes, we can complete the path as follows by adding a node at a time, making sure that our path is only connected by undirected edges in Φ, and that we do not go back along an edge that is already in the path. Let v_i be the last node that was added to the path. If v_i is equal to b_y, then we are done. If not, then v_i is not equal to $v_1 = b_x$ either (as $v_1 \cdots v_i$ is a path), and as predicate ψ_4 is true with binding B_4, conjunct ψ_3 is true with binding $B_3 = B_4$, and as $v_i \neq b_x$ and $v_i \neq b_y$, ψ_2 is true with binding $B_2 = (b_{F_{\text{node}}} \oplus (p, v_i), b_{F_{\text{set}}} \oplus (P, b_P), b)$. This, in turn, implies that ψ_1 is true with binding $B_1 = (b_{F_{\text{node}}} \oplus (p, v_i) \oplus (a, w_1) \oplus (b, w_2), b_{F_{\text{set}}} \oplus (P, b_P), b)$, for a pair of nodes $w_1, w_2 \in b_P$, which implies that $\epsilon_g(v_i, w_1), \epsilon_g(w_1, v_i), \epsilon_g(v_i, w_2), \epsilon_g(w_2, v_i) \in \Phi$. Now consider the following two situations. The first is that $v_{i-1} = w_1$ or, symmetrically, $v_{i-1} = w_2$. In this situation, we take $v_{i+1} = w_2$ (if $v_{i-1} = w_1$) or $v_{i+1} = w_1$ (if $v_{i-1} = w_2$), which is necessarily different from all previous nodes in the path, as there would otherwise be a cycle in g. If $v_{i-1} \neq w_1$ and $v_{i-1} \neq w_2$, then we know for certain that ψ_1 is also true with binding $B_1' = (b_{F_{\text{node}}} \oplus (p, v_i) \oplus (a, v_{i-1}) \oplus (b, w), b_{F_{\text{set}}} \oplus (P, b_P), b)$ for $w \in \{w_1, w_2\}$, and we can fall back to the first situation and continue for any of these paths, both of which would eventually lead to b_y: as every new node we add to the path is different from all the previous ones, and as b_P is finite, adding nodes following this procedure will eventually yield a complete path from b_x to b_y. However, this cannot happen: as both paths would lead to b_y, and both paths are distinct, they would form a cycle in g when joined together. Therefore, this case cannot occur.

($\Leftarrow$)Assume that there is a path $v_1 v_2 \cdots v_k$ with $v_1 = b_{F_{\text{node}}}(x)$, $v_k = b_{F_{\text{node}}}(y)$, and both $\epsilon_g(v_i, v_{i+1}) \in \Phi$ and $\epsilon_g(v_{i+1}, v_i) \in \Phi$ for $i \in [1, k-1]$. Then $\text{path}_\Phi(x, y)$ is true for binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b)$ because ψ_5 is true with binding $B_5 = (b_{F_{\text{node}}}, b_{F_{\text{set}}} \oplus (P, b_P), b)$ with $b_P = \{v_1, v_2, \ldots, v_k\}$. This can be seen as follows. Let $b_x = b_{F_{\text{node}}}(x)$ and $b_y = b_{F_{\text{node}}}(y)$. Obviously, $b_x = v_1 \in b_P$ and $b_y = v_k \in b_P$, so the conjuncts $x \in P$ and $y \in P$ of ψ_5 are true with binding B_5. Now, if $k = 1$, then $v_1 = v_k = b_x = b_y$, so $\text{eq}(x, y)$ will be true, which implies that ψ_5 will be true. Now consider the case that $k > 1$. Then ψ_4 is true with binding B_5, because conjunct $\exists b \in P : \text{uedge}_\Phi(x, b)$ is true as $\epsilon_g(b_x, v_2), \epsilon_g(v_2, b_x) \in \Phi$ and $v_2 \in b_P$, and because ψ_3 is true as well, which we will show next. For ψ_3 to be true, we should show that ψ_2 is true with binding $B_2 = (b_{F_{\text{node}}} \oplus (p, v_i), b_{F_{\text{set}}} \oplus (P, b_P), b)$ for any $v_i \notin \{b_x, b_y\}$, i.e., $i \in [2, k-1]$. This is the case when ψ_1 is true with binding $B_1 = (b_{F_{\text{node}}} \oplus (p, v_i) \oplus (a, w_1) \oplus (b, w_2), b_{F_{\text{set}}} \oplus (P, b_P), b)$ for a pair of nodes $w_1, w_2 \in b_P$. It is easy to see that ψ_1 is true with binding B_1 if $\{w_1, w_2\} = \{v_{i-1}, v_{i+1}\}$. This concludes the proof that $\text{path}_\Phi(x, y)$ is true with binding $(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b)$. □

DEFINITION 4.4.13. Let Σ, Φ, Φ' be alphabets, with $\Phi' \supseteq \Phi$. Let $C \supseteq A_{\Sigma,\Phi'} \cup \{\text{head}\}$ be a set of relation symbols. The C-MSO predicate $\text{reachable}_\Phi(x)$ is defined as

$$\text{reachable}_\Phi(x) \equiv \exists h : (\text{head}(h) \wedge \text{path}_\Phi(h, x)).$$

The predicate $\text{reachable}_\Phi(x)$ indicates that x is reachable from the reading head location while crossing only edges labeled in Φ. We will now prove a special case of this interpretation, which is that in a tree $\tau(t)$ (as in Definition 4.4.3), $\text{reachable}_\Phi(x)$ is true for all nodes in the same component tree $\tau_\pi(t)$ where the reading head resides.

LEMMA 4.4.14. *Let Σ, Φ, Φ' be alphabets, with $\Phi' = \Phi \cup \{drop, lift\}$. Let $C \supseteq A_{\Sigma,\Phi'} \cup \{head\}$ be a set of relation symbols. Let $m \geq 1$, and let $\tau : R_{\Sigma,\Phi} \to R_{\Sigma,\Phi'}$ be defined as in Definition 4.4.3. Furthermore, let $t \in R_{\Sigma,\Phi}$, $u \in V_t$, $\pi \in V_t^{\leq m}$, and let $B = (b_{F_{node}}, b_{F_{set}}, b)$ be a binding for C-MSO predicates with $b \supseteq b_{A_{\Sigma,\Phi'},\tau(t)} \cup \{(head, \{u_\pi\})\}$. Then $\text{reachable}_\Phi(x)$ is true with binding B if and only if $b_{F_{node}}(x) = w_\pi$ for some $w \in V_t$.*

PROOF. We have $\text{reachable}_\Phi(x) \equiv \exists h : (\text{head}(h) \wedge \text{path}_\Phi(h, x))$, which is true with binding B iff $\text{path}_\Phi(h, x)$ is true with binding $(b_{F_{\text{node}}} \oplus (h, u_\pi), b_{F_{\text{set}}}, b)$, as $\text{head}(h)$ is only true when $b_{F_{\text{node}}}(h) = u_\pi$. According to Lemma 4.4.12, $\text{path}_\Phi(h, x)$ is true with this binding iff there exists a path $v_1 v_2 \cdots v_k$ with $v_1 = b_{F_{\text{node}}}(h)$, $v_k = b_{F_{\text{node}}}(x)$ for which $\epsilon_{\tau(t)}(v_i, v_{i+1}) \in \Phi$ and $\epsilon_{\tau(t)}(v_{i+1}, v_i) \in \Phi$ for $i \in [1, k-1]$. If the path from $b_{F_{\text{node}}}(h)$ to $b_{F_{\text{node}}}(x)$ satisfies this condition, then $b_{F_{\text{node}}}(x)$ and $b_{F_{\text{node}}}(h) = u_\pi$ must be in the same component tree $\tau_\pi(t)$ of $\tau(t)$, as none of the edges leading to other component trees are labeled in Φ. All nodes in component tree $\tau_\pi(t)$ are w_π for some $w \in V_t$. On the other hand, if $b_{F_{\text{node}}}(x)$ and $b_{F_{\text{node}}}(h) = u_\pi$ are in the same component tree $\tau_\pi(t)$ of $\tau(t)$, then $\text{path}_{\tau(t)}(b_{F_{\text{node}}}(h), b_{F_{\text{node}}}(x))$ is entirely within that component tree and hence contains only edges with labels in Φ. This completes the proof of the Lemma. □

DEFINITION 4.4.15. Let Σ, Φ, Φ' be alphabets with $\Phi' \supseteq \Phi$, and let C be a set of relation symbols. If $\psi \in \text{MSO}_C$, then ψ *constrained to* Φ, written $\psi[\Phi]$, is a C'-MSO predicate, with $C' = C \cup A_{\Sigma,\Phi'} \cup \{\text{head}\}$, defined as follows:

- If $\psi \equiv x \in X$ for a node variable x and a node set variable X, or if $\psi \in \text{AF}_\alpha$ for a relation symbol $\alpha \in C$, then $\psi[\Phi] \equiv \psi$.
- If $\psi \equiv \neg\psi_1$ with $\psi_1 \in \text{MSO}_C$, then $\psi[\Phi] \equiv \neg\psi_1[\Phi]$.
- If $\psi \equiv (\psi_1 \vee \psi_2)$ with $\psi_1, \psi_2 \in \text{MSO}_C$, then $\psi[\Phi] \equiv (\psi_1[\Phi] \vee \psi_2[\Phi])$.
- If $\psi \equiv \forall x : \psi_1$ with $\psi_1 \in \text{MSO}_C$ and x a node variable, then $\psi[\Phi]$ is defined as

$$\forall x : (\text{reachable}_\Phi(x) \implies \psi_1[\Phi]).$$

- If $\psi \equiv \forall X : \psi_1$ with $\psi_1 \in \mathrm{MSO}_C$ and X a node set variable, then $\psi[\Phi]$ is defined as

$$\forall X : ((\forall x \in X : \mathrm{reachable}_\Phi(x)) \implies \psi_1[\Phi])$$

  for some node variable x.

In the definition of $\psi[\Phi]$, the predicate $\mathrm{reachable}_\Phi(x)$ is used to transform an MSO predicate $\psi \in \mathrm{MSO}_C$, that, say, works for trees $t \in T_{\Sigma,\Phi}$, to a predicate that is equivalent to ψ when it is applied to a larger tree $t' \in T_{\Sigma,\Phi'}$ that contains an isomorphic copy of t, and where the part that is isomorphic to t can only be left through "marker edges" that are labeled outside of the edge label domain of t. Note that such a larger tree t' may contain several components that are each an isomorphic copy of a (possibly different) tree; the predicate $\psi[\Phi]$ considers only the component that the *reading head* resides in. In our simulating ptt M', we will use constrained MSO predicates by taking the conditions of the original ptt M, that work on a tree $t \in R_{\Sigma,\Phi}$, and then constraining them to work on $\tau(t) \in R_{\Sigma,\Phi\cup\{\mathrm{lift},\mathrm{drop}\}}$. We will constrain the conditions of M to the edge set Φ, which will make sure that the conditions work exactly on the nodes of the component tree $\tau_\pi(t)$ of $\tau(t)$ in which the reading head resides when the condition is evaluated.

The next definition shows how to transform an MSO predicate by replacing an atomic formula by an MSO predicate. This will allow us to replace references to pebbles in M's conditions by equivalent predicates in the simulating ptt M'.

DEFINITION 4.4.16. Let C be a set of relation symbols and let $\psi \in \mathrm{MSO}_C$. Let $C' \subseteq C$, and let $f : \left(\bigcup_{\alpha\in C'} \mathrm{AF}_\alpha\right) \to \mathrm{MSO}_C$ so that for $\alpha \in C'$ and $\alpha\left(x_1, \ldots, x_{\mathrm{arity}(\alpha)}\right) \in \mathrm{AF}_\alpha$, $f\left(\alpha\left(x_1, \ldots, x_{\mathrm{arity}(\alpha)}\right)\right)$ is an MSO predicate with free variables $(F_{\mathrm{node}}, \emptyset)$, $F_{\mathrm{node}} \subseteq \left\{x_1, \ldots, x_{\mathrm{arity}(\alpha)}\right\}$. Then $\psi[f]$ is defined recursively as follows:

- If $\psi \equiv x \in X$, then $\psi[f] \equiv x \in X$.
- If $\psi \equiv \neg\psi_1$ with $\psi_1 \in \mathrm{MSO}_C$, then $\psi[f] \equiv \neg\psi_1[f]$.
- If $\psi \equiv (\psi_1 \vee \psi_2)$ with $\psi_1, \psi_2 \in \mathrm{MSO}_C$, then $\psi[f] \equiv (\psi_1[f] \vee \psi_2[f])$.
- If $\psi \equiv \forall x : \psi_1$ with $\psi_1 \in \mathrm{MSO}_C$ and a node variable x, then $\psi[f] \equiv \forall x : \psi_1[f]$.
- If $\psi \equiv \forall X : \psi_1$ with $\psi_1 \in \mathrm{MSO}_C$ and a node set variable X, then $\psi[f] \equiv \forall X : \psi_1[f]$.
- If $\psi \in \mathrm{AF}_\alpha$, $\alpha \notin C'$, then $\psi[f] \equiv \psi$.
- If $\psi \in \mathrm{AF}_\alpha$, $\alpha \in C'$, then $\psi[f] \equiv f(\psi)$.

With these definitions, we can now define M'.

DEFINITION 4.4.17. For a given ptt $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ and an integer $m \in [1, \mathrm{fg}(M)]$, we define $M' = (n-m, (\Sigma, \Phi'), (\Delta, \Gamma), Q, q_0, R')$, where

$$\begin{aligned}
\Phi' &= \Phi \cup \{\mathrm{drop}, \mathrm{lift}\} \\
R' &= \{\langle q, \psi_i\rangle \to \iota_i \mid \langle q, \psi\rangle \to \iota \text{ is in } R \text{ and } i \in [0, n]\} \\
\psi_i &= \begin{cases} \text{false} & \text{if } \psi \text{ uses } \mathrm{peb}_k \text{ for a local pebble } k < i \\ \psi[\Phi][f_i] \wedge \mathrm{simpebcount}_i \wedge \mathrm{safe}_{\iota,i} & \text{otherwise} \end{cases} \\
\iota_i &= \iota[s_i]
\end{aligned}$$

with $f_i : \left(\bigcup_{k\in[1,m]} \mathrm{AF}_{\mathrm{peb}_k}\right) \to \mathrm{MSO}_{A_{M'}}$ and $s_i : \mathrm{CI}_M \to \mathrm{CI}_{M'}$ defined as follows:

$$\begin{aligned}
f_i &= \left\{\left(\mathrm{peb}_{\min(i,m)}(x), \exists a : \left(\mathrm{edge}_{lift}(x, a)\right)\right) \mid x \text{ is a node variable}\right\} \cup \\
&\quad \{(\mathrm{peb}_k(x), \text{false}) \mid x \text{ is a node variable and } k \in [i+1, m]\} \cup \\
&\quad \left\{\left(\mathrm{peb}_k(x), \mathrm{peb}_{k-m}(x)\right) \mid x \text{ is a node variable and } k \in [m+1, n]\right\} \\
s_i &= \left\{\left(\langle q, \mathrm{drop}\rangle, \left\langle q, \mathrm{go}_{drop}\right\rangle\right) \mid q \in Q,\ i < m\right\} \cup \\
&\quad \left\{\left(\langle q, \mathrm{lift}\rangle, \left\langle q, \mathrm{go}_{lift}\right\rangle\right) \mid q \in Q,\ i \le m\right\}.
\end{aligned}$$

The predicate ψ_i uses $A_{M'}$-MSO predicate $\mathrm{simpebcount}_i$, which verifies whether the "simulated" pebble count is equal to i, and which is defined as:

$$\mathrm{simpebcount}_i \equiv \mathrm{level}_{\min(i,m)} \wedge \mathrm{pebcount}_{\max(0,i-m)}.$$

Recall that the simulated pebble count is equal to the number of "drop" edges that have been followed plus the number of real pebbles that have been dropped. The *simpebcount* predicate uses the *level* subpredicate to check the number of "lift" edges that can be followed from the reading head in the simulating tree (which, of course, corresponds to the number of "drop" edges that have been followed), and uses *pebcount*, as defined in Definition 3.2.1 (but with $n-m$ instead of n), to check the additional real pebbles that have been dropped in the simulating ptt. The *level* predicate is defined as follows for $j \in [0, m]$:

$$\begin{aligned}
\mathrm{level}_j \equiv\ & \exists x_0, y_0, x_1, y_1, x_2, y_2, \ldots, x_j : \mathrm{head}(x_0) \\
& \wedge \left(\left(\bigwedge_{k\in[0,j-1]} \left(\mathrm{path}_\Phi(x_k, y_k) \wedge \mathrm{edge}_{\mathrm{lift}}(y_k, x_{k+1})\right)\right)\right. \\
& \left. \wedge \neg\left(\exists y_j, x_{j+1} :\ \left(\mathrm{path}_\Phi(x_j, y_j) \wedge \mathrm{edge}_{\mathrm{lift}}(y_j, x_{j+1})\right)\right)\right).
\end{aligned}$$

The ψ_i predicate also uses the *safe* auxiliary predicate, which verifies that the input instructions contained in the translated instruction ι_i are, in fact, valid in the situation at hand. The inclusion of this predicate is necessitated by the fact that, by Definition 3.2.1(3), the right hand side of a ptt rule can only use instructions whose validity is

implied by the rule's condition. The *safe* predicate is defined as:

$$\mathrm{safe}_{\iota,i} \equiv \bigwedge_{\nu_{\iota_i}(w)=\langle q,\omega\rangle \notin \Delta\,(w \in V_{\iota_i})} \begin{cases} \exists x : \exists y : \big(\mathrm{head}\,(x) \wedge \mathrm{edge}_\phi\,(x,y)\big) & \text{if } \omega = \mathrm{go}_\phi\ (\phi \in \Phi') \\ \neg\exists x : \mathrm{peb}_{n-m}\,(x) & \text{if } \omega = \mathrm{drop} \\ \exists x : \big(\mathrm{head}\,(x) \wedge \mathrm{peb}_{i-m}\,(x)\big) & \text{if } \omega = \mathrm{lift}. \end{cases}$$

LEMMA 4.4.18. *M' is a ptt.*

PROOF. Since M' clearly has the same structure as a ptt, proving that it is a ptt is a matter of proving that the rules in R', which are of the form $\langle q, \psi_i\rangle \to \iota_i$ actually have $q \in Q$, ψ_i a valid condition of M', and $\iota_i \in I_{M',\psi_i}$. Furthermore, we need to specify $\mathrm{out}_{R'} : Q \to \mathcal{P}\,(\Gamma)$. First of all, for any rule $\langle q, \psi_i\rangle \to \iota_i$, we have $\langle q, \psi\rangle \to \iota \in R$, so $q \in Q$. Also, it is easy to verify that ψ_i is a well-formed, closed $A_{M'}$-MSO predicate.

Secondly, observe that ψ_i has been defined to include, in the top-level conjunct $\mathrm{safe}_{\iota,i}$, for every input instruction ω used in ι_i *except for the* lift *instruction*, the exact predicate χ that ψ_i needs to imply for instruction ω to be in II_{M',ψ_i}, by the definition of II_{M',ψ_i}. As these are top-level conjunctions, it is clear that for any input instruction $\omega \neq \mathrm{lift}$ that occurs in ι_i, $M' \models (\psi_i \implies \chi)$, and so $\omega \in \mathrm{II}_{M',\psi_i}$ and $\iota_i \in I_{M',\psi_i} = R_{\Delta,\Gamma}\,(Q \times \mathrm{II}_{M',\psi_i})$. In the case that $\omega = \mathrm{lift}$, we have $M' \models \big(\psi_i \implies \exists x : \big(\mathrm{head}\,(x) \wedge \mathrm{peb}_{i-m}\,(x)\big)\big)$, but also $M' \models (\psi_i \implies \mathrm{simpebcount}_i)$, and as $M' \models \big(\mathrm{simpebcount}_i \implies \mathrm{pebcount}_{\max(0,i-m)}\big)$, $M' \models \big(\psi_i \implies \mathrm{pebcount}_{\max(0,i-m)}\big)$. Because $\exists x : \big(\mathrm{head}\,(x) \wedge \mathrm{peb}_{i-m}\,(x)\big)$ implies that the pebble count is at least 1, we therefore have $M' \models \big(\psi_i \implies \mathrm{pebcount}_{i-m}\big)$. We derive

$$\begin{aligned} & \exists x : \big(\mathrm{head}\,(x) \wedge \mathrm{peb}_{i-m}\,(x)\big) \wedge \mathrm{pebcount}_{i-m} \\ \equiv\ & \exists x : \big(\mathrm{head}\,(x) \wedge \mathrm{peb}_{i-m}\,(x) \wedge \mathrm{pebcount}_{i-m}\big) \\ \implies\ & \exists x : \left(\mathrm{head}\,(x) \wedge \bigvee_{k\in[1,n-m]} \left(\mathrm{peb}_k\,(x) \wedge \mathrm{pebcount}_k\right)\right) \\ \equiv\ & \exists x : (\mathrm{head}\,(x) \wedge \mathrm{toppebble}\,(x)) \end{aligned}$$

which is exactly the predicate that ψ_i needs to imply for *lift* to be in II_{M',ψ_i}. Therefore, the reasoning that we used earlier for the other input instructions is also valid for the *lift* instruction.

Finally, let $\mathrm{out}_{R'}\,(q) = \mathrm{out}_R\,(q)$ and observe that if the conditions specified by Definition 3.2.1(6) hold for $\mathrm{out}_R\,(q)$ and a rule $\langle q, \psi\rangle \to \iota \in R$, then they also hold for $\mathrm{out}_{R'}\,(q) = \mathrm{out}_R\,(q)$ and all $\langle q, \psi_i\rangle \to \iota_i \in R'$ $(i \in [0, n])$, as $\iota_i = \iota\,[s_i]$ has configuration instructions with a state q at the exact same nodes where ι has them, with the same set of outgoing edges. This completes the proof that M' is a ptt. □

We will now prove that the configuration mapping cc_t, as defined in Definition 4.4.2, satisfies the conditions given by the definition of stepwise simulation (Definition 4.3.1).

LEMMA 4.4.19. *Let* $(u,\pi) \in IC_{n,t}$, $\psi \in CMSO_{A_M}$ *where* ψ *does not use any relation symbols in* $P_{local} = \{peb_k \mid k < |\pi|,\ k \text{ local in } M\}$. *Then* ψ *is true with binding* $\left(\emptyset, \emptyset, b_{A_M,t,(u,\pi)}\right)$ *if and only if* $\psi\left[\Phi\right]\left[f_{|\pi|}\right]$ *is true with binding* $\left(\emptyset, \emptyset, b_{A_{M'},\tau(t),cic_t((u,\pi))}\right)$.

PROOF. We will prove this using structural induction. Let $h_{t\to\tau_{\text{left}(\pi,m)}(t)} : V_t \to V_{\tau(t)}$ be the injective homomorphism defined as just after Definition 4.4.3, and let $h_{\text{left}(\pi,m)} = h_{t\to\tau_{\text{left}(\pi,m)}(t)}$ for brevity. For $\psi \in \text{MSO}_{A_M}$, let g_{node} be a function from node variable bindings for ψ on t to node variable bindings for $\psi\left[\Phi\right]\left[f_{|\pi|}\right]$ on $\tau(t)$ so that for $b_{F_{\text{node}}} : F'_{\text{node}} \to V_t$, $g_{\text{node}}\left(b_{F_{\text{node}}}\right) : F'_{\text{node}} \to V_{\tau(t)}$ with $g_{\text{node}}\left(b_{F_{\text{node}}}\right)(x) = h_{\text{left}(\pi,m)}\left(b_{F_{\text{node}}}(x)\right)$ for all $x \in F'_{\text{node}}$. Similarly, let g_{set} be a function from node set variable bindings for ψ on t to node set variable bindings for $\psi\left[\Phi\right]\left[f_{|\pi|}\right]$ on $\tau(t)$ so that for $b_{F_{\text{set}}} : F'_{\text{set}} \to \mathcal{P}(V_t)$, $g_{\text{set}}\left(b_{F_{\text{set}}}\right) : F'_{\text{set}} \to \mathcal{P}\left(V_{\tau(t)}\right)$ with $g_{\text{set}}\left(b_{F_{\text{set}}}\right)(X) = \left\{h_{\text{left}(\pi,m)}(w) \mid w \in b_{F_{\text{set}}}(X)\right\}$ for all $X \in F'_{\text{set}}$. As $h_{\text{left}(\pi,m)}$ is total and injective, so are g_{node} and g_{set}. For brevity, let $b = b_{A_M,t,(u,\pi)}$ and $b' = b_{A_{M'},\tau(t),\text{cic}_t((u,\pi))} = b_{A_{M'},\tau(t),\left(u_{\text{left}(\pi,m)},\text{stackenc}_t(\pi)\right)}$.

Property: If $\psi \in \text{MSO}_{A_M}$, ψ does not use relation symbols in P_{local}, then ψ is true with binding $B = \left(b_{F_{\text{node}}}, b_{F_{\text{set}}}, b\right)$ if and only if $\psi\left[\Phi\right]\left[f_{|\pi|}\right]$ is true with binding $B' = \left(g_{\text{node}}\left(b_{F_{\text{node}}}\right), g_{\text{set}}\left(b_{F_{\text{set}}}\right), b'\right)$.

Base: The base of the structural induction is formed by all predicates that do not use subformulas, i.e., by the atomic formulas. We will separately consider $x \in X$ and the atomic formulas over each of the relation symbols in $A_M = \{\text{eq}, \text{head}, \text{false}, \text{true}\} \cup \{\text{lab}_\sigma \mid \sigma \in \Sigma\} \cup \left\{\text{edge}_\phi \mid \phi \in \Phi\right\} \cup \{\text{peb}_i \mid i \in [1,n]\}$.

- Consider $\psi \equiv \text{false}$. Then $\psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \text{false}$. Obviously, ψ is always false with binding B and $\psi\left[\Phi\right]\left[f_{|\pi|}\right]$ is always false with binding B'.
- Consider $\psi \equiv \text{true}$. Then $\psi\left[\Phi\right]\left[f_{|\pi}\right] \equiv \text{true}$. Again, ψ is obviously always true with binding B, and $\psi\left[\Phi\right]\left[f_{|\pi|}\right]$ is always true with binding B'.
- Consider $\psi \equiv \psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv x \in X$ for some node variable x and node set variable X. ψ is true with binding B if and only if $b_{F_{\text{node}}}(x) \in b_{F_{\text{set}}}(X)$, which is true if and only if

$$\begin{aligned} g_{\text{node}}\left(b_{F_{\text{node}}}\right)(x) &= h_{\text{left}(\pi,m)}\left(b_{F_{\text{node}}}(x)\right) \\ &\in \left\{h_{\text{left}(\pi,m)}(w) \mid w \in b_{F_{\text{set}}}(X)\right\} \\ &= g_{\text{set}}\left(b_{F_{\text{set}}}\right)(X) \end{aligned}$$

  (where the iff is justified by the fact that $h_{\text{left}(\pi,m)}$ is total and injective), which is true if and only if $x \in X$ is true with binding B'.
- Consider $\psi \equiv \psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \text{head}(x)$ for some node variable x. Clearly, $\text{head}(x)$ is true with binding B iff $b_{F_{\text{node}}}(x) = u$ iff $g_{\text{node}}\left(b_{F_{\text{node}}}\right)(x) = u_{\text{left}(\pi,m)}$ (where the iff is again justified by the fact that $h_{\text{left}(\pi,m)}$ is total and injective) iff $\text{head}(x)$ is true with binding B'.

- Consider $\psi \equiv \psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \mathrm{eq}\left(x,y\right)$ for some node variables x and y. Clearly, $\mathrm{eq}\left(x,y\right)$ is true with binding B iff $b_{F_{\mathrm{node}}}\left(x\right) = b_{F_{\mathrm{node}}}\left(y\right)$ iff $h_{\mathrm{left}(\pi,m)}\left(b_{F_{\mathrm{node}}}\left(x\right)\right) = h_{\mathrm{left}(\pi,m)}\left(b_{F_{\mathrm{node}}}\left(y\right)\right)$ (as $h_{\mathrm{left}(\pi,m)}$ is total and injective) iff $g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right) = g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(y\right)$ iff $\mathrm{eq}\left(x,y\right)$ is true with binding B'.
- Consider $\psi \equiv \psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \mathrm{lab}_{\sigma}\left(x\right)$ for some node variable x and node label $\sigma \in \Sigma$. We find that $\mathrm{lab}_{\sigma}\left(x\right)$ is true with binding B iff $\nu_t\left(b_{F_{\mathrm{node}}}\left(x\right)\right) = \sigma$, and $\mathrm{lab}_{\sigma}\left(x\right)$ is true with binding B' iff

$$\begin{aligned}\nu_{\tau(t)}\left(g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right)\right) = \\ \nu_{\tau(t)}\left(\left(b_{F_{\mathrm{node}}}\left(x\right)\right)_{\mathrm{left}(\pi,m)}\right) = \\ \nu_t\left(b_{F_{\mathrm{node}}}\left(x\right)\right) = \sigma\end{aligned}$$

  (using the fact that $\nu_{\tau(t)}\left(w_{\pi}\right) = \nu_t\left(w\right)$, by the definition of $\tau\left(t\right)$).
- Consider $\psi \equiv \psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \mathrm{edge}_{\phi}\left(x,y\right)$ for some node variables x,y and an edge label $\phi \in \Phi$. We find that $\mathrm{edge}_{\phi}\left(x,y\right)$ is true with binding B iff $\epsilon_t\left(b_{F_{\mathrm{node}}}\left(x\right), b_{F_{\mathrm{node}}}\left(y\right)\right) = \phi$, and $\mathrm{edge}_{\phi}\left(x,y\right)$ is true with binding B' iff

$$\begin{aligned}\epsilon_{\tau(t)}\left(g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right), g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right)\right) = \\ \epsilon_{\tau(t)}\left(\left(b_{F_{\mathrm{node}}}\left(x\right)\right)_{\mathrm{left}(\pi,m)}, \left(g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right)\right)_{\mathrm{left}(\pi,m)}\right) = \\ \epsilon_t\left(b_{F_{\mathrm{node}}}\left(x\right), b_{F_{\mathrm{node}}}\left(y\right)\right) = \phi\end{aligned}$$

  (using the fact that $\epsilon_{\tau(t)}\left(w_{\pi}, w'_{\pi}\right) = \epsilon_t\left(w,w'\right)$, by the definition of $\tau\left(t\right)$).
- Consider $\psi \equiv \mathrm{peb}_k\left(x\right)$ for some node variable x and some pebble number $k \in \left[m+1,n\right]$. Then $\psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \mathrm{peb}_{k-m}\left(x\right)$. Now, $\mathrm{peb}_k\left(x\right)$ is true with binding B iff $b_{F_{\mathrm{node}}}\left(x\right) = \pi\left(k\right)$ iff $g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right) = \left(b_{F_{\mathrm{node}}}\left(x\right)\right)_{\mathrm{left}(\pi,m)} = \left(\pi\left(k\right)\right)_{\mathrm{left}(\pi,m)} = \mathrm{pebenc}_t\left(\pi\left(k\right), \mathrm{left}\left(\pi,m\right)\right) = \mathrm{stackenc}_t\left(\pi\right)\left(k-m\right)$ (where again we use the fact that $h_{\mathrm{left}(\pi,m)}$ is total and injective) iff $\mathrm{peb}_{k-m}\left(x\right)$ is true with binding B'.
- Consider $\psi \equiv \mathrm{peb}_k\left(x\right)$ for some node variable x and $k = \min(\left|\pi\right|, m)$. Then $\psi\left[\Phi\right]\left[f_{|\pi|}\right] \equiv \exists a : \left(\mathrm{edge}_{lift}\left(x,a\right)\right)$. Also, because $k = \min\left(\left|\pi\right|, m\right)$, $\mathrm{left}\left(\pi,k\right) = \mathrm{left}\left(\pi,m\right)$. Now, $\mathrm{peb}_k\left(x\right)$ is true with binding B iff $\pi\left(k\right) = b_{F_{\mathrm{node}}}\left(x\right)$. We have $g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right) = \left(\pi\left(k\right)\right)_{\mathrm{left}(\pi,m)} = \left(\pi\left(k\right)\right)_{\mathrm{left}(\pi,k)}$, which means that $\left(\pi\left(k\right)\right)_{\mathrm{left}(\pi,m)} = \left(\pi\left(k\right)\right)_{\pi'\pi(k)}$ where $\pi' = \mathrm{left}\left(\pi, k-1\right)$. By the definition of $\tau\left(t\right)$, there then exists an edge $\left(\left(\left(\pi\left(k\right)\right)_{\pi'\pi(k)}, \left(\pi\left(k\right)\right)_{\pi'}\right), \mathrm{lift}\right)$, which means that $\exists a : \left(\mathrm{edge}_{lift}\left(x,a\right)\right)$ is true with binding B'. Reasoning in the other direction, the definition of $\tau\left(t\right)$ implies that the only nodes $g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right)$ that have an outgoing edge labeled *lift* are of the form $\left(\pi\left(k\right)\right)_{\pi'\pi(k)}$, which gives us $g_{\mathrm{node}}\left(b_{F_{\mathrm{node}}}\right)\left(x\right) = \left(b_{F_{\mathrm{node}}}\left(x\right)\right)_{\mathrm{left}(\pi,m)} = \left(\pi\left(k\right)\right)_{\pi'\pi(k)}$ and therefore $b_{F_{\mathrm{node}}}\left(x\right) = \pi\left(k\right)$, which implies

that $\text{peb}_k(x)$ is true with binding B. This proves the double implication for this case.

- Consider $\psi \equiv \text{peb}_k(x)$ for some node variable x and $k \in [|\pi|+1, m]$. Then $\psi[\Phi]\left[f_{|\pi|}\right] \equiv$ false, which is false for any binding B'. Then we need to show that $\text{peb}_k(x)$ is false for any binding B. Take a look at $k \in [|\pi|+1, m]$. Then $k > |\pi|$, which means that $\text{peb}_k(x)$ is false.

Note that the three cases for $\psi = \text{peb}_k(x)$ are exhaustive, because all pebbles $k < \min(|\pi|, m)$ are local in M and hence are in P_{local}. This concludes the base proof.

Induction: Assume that the property holds for all subpredicates that may be used in a predicate. Now let $\psi \in \text{MSO}_{A_M}$ where ψ does not use relation symbols in P_{local}. We will show that ψ is true with binding $B = (b_{F_{\text{node}}}, b_{F_{\text{set}}}, b)$ if and only if $\psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding $B' = (g_{\text{node}}(b_{F_{\text{node}}}), g_{\text{set}}(b_{F_{\text{set}}}), b')$. We have already shown that this is true when ψ does not contain any subpredicates, so we will now consider all of the possibilities for a ψ that *does* contain subpredicates.

- Consider $\psi \equiv \neg\psi_1$. As $\psi[\Phi]\left[f_{|\pi|}\right] \equiv \neg\psi_1[\Phi]\left[f_{|\pi|}\right]$ (by definition of $[\Phi]$ and $\left[f_{|\pi|}\right]$), and ψ_1 is true with binding B iff $\psi_1[\Phi]\left[f_{|\pi|}\right]$ is true with binding B' (by assumption), we have that $\psi \equiv \neg\psi_1$ is *false* with binding B iff $\psi[\Phi]\left[f_{|\pi|}\right] \equiv \neg\psi_1[\Phi]\left[f_{|\pi|}\right]$ is false with binding B', and as a predicate is always either true or false, ψ is *true* with binding B iff $\psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding B'.
- Consider $\psi \equiv (\psi_1 \vee \psi_2)$. This predicate is true with binding B iff at least one of ψ_1 and ψ_2 is true with binding B, iff at least one of $\psi_1[\Phi]\left[f_{|\pi|}\right]$ and $\psi_2[\Phi]\left[f_{|\pi|}\right]$ is true with binding B' (because the property holds for ψ_1 and ψ_2, by assumption) iff $\left(\psi_1[\Phi]\left[f_{|\pi|}\right] \vee \psi_2[\Phi]\left[f_{|\pi|}\right]\right) \equiv (\psi_1 \vee \psi_2)[\Phi]\left[f_{|\pi|}\right] \equiv \psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding B'.
- Consider $\psi \equiv \forall x : \psi_1$. We have $\psi[\Phi] \equiv \forall x : (\psi_1[\Phi] \vee \neg\text{reachable}_\Phi(x))$, and as $\text{reachable}_\Phi(x)$ does not use formulas affected by $f_{|\pi|}$, $\text{reachable}_\Phi(x)\left[f_{|\pi|}\right] = \text{reachable}_\Phi(x)$ and therefore $\psi[\Phi]\left[f_{|\pi|}\right] \equiv \forall x : \left(\psi_1[\Phi]\left[f_{|\pi|}\right] \vee \neg\text{reachable}_\Phi(x)\right)$. By Lemma 4.4.14 (applied to $\text{left}(\pi, m)$), for all $w \in V_t$, $\text{reachable}_\Phi(x)$ is false with binding

$$(g_{\text{node}}(b_{F_{\text{node}}}) \oplus (x, w_{\pi'}), g_{\text{set}}(b_{F_{\text{set}}}), b')$$

if and only if $\pi' \neq \text{left}(\pi, m)$. Hence, $\psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding B' iff $\left(\psi_1[\Phi]\left[f_{|\pi|}\right] \vee \neg\text{reachable}_\Phi(x)\right)$ is true for all bindings

$$(g_{\text{node}}(b_{F_{\text{node}}}) \oplus (x, w), g_{\text{set}}(b_{F_{\text{set}}}), b'),$$

with $w \in V_{\tau(t)}$, iff $\psi_1[\Phi]\left[f_{|\pi|}\right]$ is true for all bindings

$$\left(g_{\text{node}}(b_{F_{\text{node}}}) \oplus \left(x, w_{\text{left}(\pi,m)}\right), g_{\text{set}}(b_{F_{\text{set}}}), b'\right),$$

with $w \in V_t$, iff ψ_1 is true for all bindings $(b_{F_{\text{node}}} \oplus (x, w), b_{F_{\text{set}}}, b)$ with $w \in V_t$, iff ψ is true with binding B.

- Consider $\psi \equiv \forall X : \psi_1$. By the same reasoning as in the previous case, $\psi[\Phi]\left[f_{|\pi|}\right] \equiv \forall X : \left(\psi_1[\Phi]\left[f_{|\pi|}\right] \vee \exists x \in X : \neg\text{reachable}_\Phi(x)\right)$. By Lemma 4.4.14, for all $w \in V_t$, $\text{reachable}_\Phi(x)$ is false with binding

$$(g_{\text{node}}(b_{F_{\text{node}}}) \oplus (x, w_{\pi'}), g_{\text{set}}(b_{F_{\text{set}}}), b')$$

if and only if $\pi' \neq \text{left}(\pi, m)$, and therefore $\exists x \in X : \neg\text{reachable}_\Phi(x)$ is false with binding $(g_{\text{node}}(b_{F_{\text{node}}}), g_{\text{set}}(b_{F_{\text{set}}}) \oplus (X, b_X), b')$ if and only if there is a $w_{\pi'} \in b_X$ so that $\pi' \neq \text{left}(\pi, m)$. Therefore,

$$\forall X : \left(\psi_1[\Phi]\left[f_{|\pi|}\right] \vee \exists x \in X : \neg\text{reachable}_\Phi(x)\right)$$

is true with binding B' iff $\left(\psi_1[\Phi]\left[f_{|\pi|}\right] \vee \exists x \in X : \neg\text{reachable}_\Phi(x)\right)$ is true with binding B' combined with all possible bindings $b_X \in \mathcal{P}\left(V_{\tau(t)}\right)$ for X, iff $\psi_1[\Phi]\left[f_{|\pi|}\right]$ is true with binding B' combined with all possible bindings $b_X \in \mathcal{P}\left(V_{\tau_{\text{left}(\pi,m)}(t)}\right)$ for X, iff ψ_1 is true with binding B combined with all possible bindings $b_X \in \mathcal{P}\left(h^{-1}_{t \to \tau_{\text{left}(\pi,m)}(t)}\left(V_{\tau_{\text{left}(\pi,m)}(t)}\right)\right) = \mathcal{P}(V_t)$ for X, iff ψ is true with binding B.

This completes the induction step, which implies that the property holds for all $\psi \in \text{MSO}_{A_M}$ that do not use relation symbols in P_{local}. □

In order to be able to fully line up the truth values of ψ and ψ_i, we need to consider the extra conjuncts that are included in ψ_i next to $\psi[\Phi][f_i]$, plus the case that ψ uses relation symbols from P_{local}.

LEMMA 4.4.20. *Let* $\overline{M} = \left(\overline{n}, \left(\overline{\Sigma}, \overline{\Phi}\right), \left(\overline{\Delta}, \overline{\Gamma}\right), \overline{Q}, \overline{q_0}, \overline{R}\right)$ *be a ptt, let* $t \in R_{\overline{\Sigma},\overline{\Phi}}$, *let* $(u, \pi) \in IC_{\overline{n},t}$, *and let* $j \in [0, \overline{n}]$. *Then* pebcount_j *is true with binding* $B = \left(\emptyset, \emptyset, b_{A_{\overline{M}},t,(u,\pi)}\right)$ *iff* $j = |\pi|$.

PROOF. Recall that $\text{pebcount}_j \equiv \left(\exists x : \text{peb}_j(x)\right) \wedge \left(\bigwedge_{k \in [j+1,\overline{n}]} \neg\exists y : \text{peb}_k(y)\right)$ for $j \in [1, \overline{n}]$. This predicate is true with binding B iff there is a $\pi(j)$ and there are no $\pi(j')$ for $j' > j$, which is iff $|\pi| = j$. On the other hand, for $j = 0$, we have $\text{pebcount}_j \equiv \left(\bigwedge_{k \in [j+1,\overline{n}]} \neg\exists y : \text{peb}_k(y)\right)$, which is clearly true iff there is no $\pi(j')$ for any $j' > j$, i.e., when $|\pi| \leq j$, i.e., $|\pi| = 0$. □

LEMMA 4.4.21. *Let* $(u, \pi) \in IC_{n,t}$, *and let* $j \in [0, m]$. *Then* level_j *is true with binding* $B' = \left(\emptyset, \emptyset, b_{A_{M'},\tau(t),cic_t((u,\pi))}\right)$ *iff* $|left(\pi, m)| = j$, *i.e.,* $j = min(m, |\pi|)$.

PROOF. From the definition of level_j, it is clear that level_j is true iff there exists a series of nodes $w_0, w_0', w_1, w_1', \ldots, w_j \in V_{\tau(t)}$ so that $w_0 = u_{\text{left}(\pi,m)}$ is at the reading head and so that

(1) for all k, $\epsilon_{\tau(t)}(w'_k, w_{k+1}) = \text{lift}$ and $\text{path}_{\tau(t)}(w_k, w'_k)$ only crosses edges labeled in Φ,
(2) there exist *no* w'_j, w_{j+1} where $\text{path}_{\tau(t)}(w_j, w'_j)$ only crosses edges labeled in Φ, and $\epsilon_{\tau(t)}(w'_j, w_{j+1}) = \text{lift}$.

Let $\pi' = \text{left}(\pi, m)$, so $w_0 = u_{\pi'}$. Constraint 1 on $w_0, w'_0, w_1, w'_1, \ldots, w_j \in V_{\tau(t)}$ is satisfied iff for $k \in [0, j-1]$, $w'_k = \pi'(|\pi'| - k)_{\text{left}(\pi', |\pi'|-k)}$ and $w_{k+1} = \pi'(|\pi'| - k)_{\text{left}(\pi', |\pi'|-(k+1))}$, as within any component tree $\tau_{\pi''}(t)$ of $\tau(t)$, there can be at most one node w'_k that is reachable through a Φ-path that has a *lift*-edge, which is $\pi''(|\pi''|)_{\pi''}$, and as this *lift*-edge always leads to $\pi''(|\pi''|)_{\text{left}(\pi'', |\pi''|-1)}$. Constraint 2 is clearly satisfied iff w_j is in $\tau_\lambda(t)$, i.e., iff $\text{left}(\pi', |\pi'| - j) = \lambda$ iff $|\pi'| - j = 0$ iff $|\pi'| = j$. As $|\pi'| = |\text{left}(\pi, m)| = \min(m, |\pi|)$, this proves that level_j is true iff $j = \min(m, |\pi|)$. □

LEMMA 4.4.22. *Let $(u, \pi) \in IC_{n,t}$ and let $i \in [0, n]$. Then pebcount_i (as defined in Definition 3.2.1) is true with binding $B = (\emptyset, \emptyset, b_{A_M, t, (u,\pi)})$ iff simpebcount_i is true with binding $B' = (\emptyset, \emptyset, b_{A_{M'}, \tau(t), cic_t((u,\pi))})$.*

PROOF. We will compare the truth values of simpebcount_i and pebcount_i.

CASE 1. When $i \leq m$, we have $\text{simpebcount}_i \equiv \text{level}_i \wedge \text{pebcount}_0$. Two subcases:

- $|\pi| > m$. Then $|\text{stackenc}_t(\pi)| > 0$, so pebcount_0 is false with binding B' (by Lemma 4.4.20) and therefore simpebcount_0 is false with binding B'. From $i \leq m$ and $|\pi| > m$ we can deduce that $i \neq |\pi|$, and therefore pebcount_i is false with binding B.
- $|\pi| \leq m$. Then $|\text{stackenc}_t(\pi)| = 0$ and pebcount_0 is true with binding B'. This implies that the truth value of simpebcount_i is given by level_i. By Lemma 4.4.21, level_i is true with binding B' iff $i = |\text{left}(\pi, m)|$, i.e., $i = |\pi|$ (because $|\pi| \leq m$). This corresponds exactly with pebcount_i, which is true with binding B iff $i = |\pi|$.

CASE 2. When $i > m$, $\text{simpebcount}_i \equiv \text{level}_m \wedge \text{pebcount}_{i-m}$. Two subcases:

- $|\pi| > m$. By Lemma 4.4.21, level_m is true with binding B', because $m = \min(m, |\pi|)$. By Lemma 4.4.20, pebcount_{i-m} is true with binding B' iff $i - m = |\text{stackenc}_t(\pi)| = |\pi| - m$, i.e., $i = |\pi|$. Again, this corresponds exactly with pebcount_i.
- $|\pi| \leq m$. Then $|\text{stackenc}_t(\pi)| = 0$. By Lemma 4.4.20, pebcount_{i-m} is true with binding B' iff $|\text{stackenc}_t(\pi)| = i - m$, and as $i > m$, $i - m > 0$ and therefore this condition is not satisfied, and simpebcount_i is false with binding B'. Likewise, from $i > m$ and $|\pi| \leq m$ we can conclude that $i \neq |\pi|$ and that pebcount_i is false with binding B.

□

LEMMA 4.4.23. *Let* $(u,\pi) \in IC_{n,t}$ *and let* $\langle q,\psi\rangle \to \iota \in R$. *Then* ψ *is true with binding* $B = \left(\emptyset,\emptyset,b_{A_M,t,(u,\pi)}\right)$ *if and only if* $\psi_{|\pi|}$ *is true with binding* $B' = \left(\emptyset,\emptyset,b_{A_{M'},\tau(t),cic_t((u,\pi))}\right)$.

PROOF. First, we will prove the special case that ψ uses a relation symbol in $P_{\text{local}} = \{\text{peb}_k \mid k < |\pi|, k \text{ local in } M\}$, which implies $\psi_{|\pi|} \equiv \text{false}$, which is always false. As ψ uses $\text{peb}_k \in P_{\text{local}}$, $\psi \implies \text{free}_{k+1}$ is true for (u,π) (by the definition of a local pebble), and as $\text{free}_{k+1} \implies \neg\exists x : \text{peb}_{k+1}(x)$ while $k+1 \leq |\pi|$, free_{k+1} is false with binding B, and therefore the fact that $\psi \implies \text{free}_{k+1}$ is true implies that ψ must be false. This proves the double implication for this special case.

Now for the general case in which $\psi_{|\pi|}$ consists of $\psi[\Phi]\left[f_{|\pi|}\right]$ and two other conjuncts. By Lemma 4.4.19, ψ is true with binding B iff $\psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding B'. That means that the Lemma's statement is true iff $\psi[\Phi]\left[f_{|\pi|}\right]$ implies both of the other conjuncts, $\text{simpebcount}_{|\pi|}$ and $\text{safe}_{\iota,|\pi|}$. Assume ψ is true with binding B and $\psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding B'. By Lemma 4.4.22, $\text{simpebcount}_{|\pi|}$ is true with binding B'.

The truth of $\text{safe}_{\iota,|\pi|}$ is a bit more involved: we will have to prove that all of the conjuncts in $\text{safe}_{\iota,|\pi|}$ are true with binding B'. These cases cover all possible conjuncts in $\text{safe}_{\iota,|\pi|}$:

- Conjunct $\exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right)$ is in $\text{safe}_{\iota,|\pi|}$ with $\phi \in \Phi$ iff there exists a $w \in V_{\iota_{|\pi|}}$ so that $\nu_{\iota_{|\pi|}}(w) = \langle q, \text{go}_\phi\rangle$ iff there exists a $w \in V_\iota$ so that $\nu_\iota = \langle q, \text{go}_\phi\rangle$. This implies that $\psi \implies \exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right)$ is true with binding B (because $\text{go}_\phi \in \text{II}_{M,\psi}$), which implies that $\psi[\Phi]\left[f_{|\pi|}\right] \implies \left(\exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right)\right)[\Phi]\left[f_{|\pi|}\right]$ is true with binding B' (by Lemma 4.4.19). We know that $\psi[\Phi]\left[f_{|\pi|}\right]$ is true with binding B', so

  $$\left(\exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right)\right)[\Phi]\left[f_{|\pi|}\right]$$

  must be true with binding B' as well. Transforming this predicate into predicates with the same truth value with binding B' yields

  $$\begin{aligned} & \left(\exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right)\right)[\Phi]\left[f_{|\pi|}\right] \\ \equiv\ & \exists x : \left(\exists y : \left(\left(\text{head}(x) \wedge \text{edge}_\phi(x,y)\right) \wedge \text{reachable}_\Phi(y)\right) \wedge \text{reachable}_\Phi(x)\right) \end{aligned}$$

using the (standard) fact that for a predicate $\chi \in \mathrm{MSO}_{A_M}$ and a node variable x,

$$\begin{aligned}(\exists x : \chi)\,[\Phi] &\equiv (\neg\forall x : \neg\chi)\,[\Phi] \\ &\equiv \neg\forall x : ((\neg\chi) \vee \neg\mathrm{reachable}_{\Phi}(x)) \\ &\equiv \neg\forall x : \neg(\chi \wedge \mathrm{reachable}_{\Phi}(x)) \\ &\equiv \exists x : (\chi \wedge \mathrm{reachable}_{\Phi}(x))\,.\end{aligned}$$

This obviously implies that $\exists x : \exists y : \big(\mathrm{head}(x) \wedge \mathrm{edge}_{\phi}(x,y)\big)$ is true with binding B' as well, which is what we were trying to prove.

- Conjunct $\neg\exists x : \mathrm{peb}_{n-m}(x)$ is in $\mathrm{safe}_{\iota,|\pi|}$ iff there exists a $w \in V_{\iota_{|\pi|}}$ so that $\nu_{\iota_{|\pi|}}(w) = \langle q, \mathrm{drop}\rangle$ iff there exists a $w \in V_{\iota}$ so that $\nu_{\iota} = \langle q, \mathrm{drop}\rangle$ and $|\pi| \geq m$. This implies that $\psi \implies \neg\exists x : \mathrm{peb}_{n}(x)$ is true with binding B (because $\mathrm{drop} \in \mathrm{II}_{M,\psi}$), which implies that $\psi\,[\Phi]\,[f_{|\pi|}] \implies (\neg\exists x : \mathrm{peb}_{n}(x))\,[\Phi]\,[f_{|\pi|}]$ is true with binding B' (by Lemma 4.4.19). First, consider $|\pi| = n$. Then ψ must be false, because $\psi \implies \neg\exists x : \mathrm{peb}_{n}(x)$ is true, which contradicts our assumption that ψ is true. Now consider $|\pi| < n$. We have already shown that $\mathrm{simpebcount}_{|\pi|}$ is true, and if $|\pi| \geq m$, $\mathrm{simpebcount}_{|\pi|}$ contains conjunct $\mathrm{pebcount}_{|\pi|-m}$, which includes conjunct $\neg\exists x : \mathrm{peb}_{n-m}(x)$ (because $|\pi| < n$). Therefore, $\neg\exists x : \mathrm{peb}_{n-m}(x)$ is true.
- Conjunct $\exists x : \big(\mathrm{head}(x) \wedge \mathrm{peb}_{|\pi|-m}(x)\big)$ is in $\mathrm{safe}_{\iota,|\pi|}$ iff there exists a $w \in V_{\iota_{|\pi|}}$ so that $\nu_{\iota_{|\pi|}}(w) = \langle q, \mathrm{lift}\rangle$ iff there exists a $w \in V_{\iota}$ so that $\nu_{\iota} = \langle q, \mathrm{lift}\rangle$ and $|\pi| > m$. This implies that $\psi \implies \exists x : (\mathrm{head}(x) \wedge \mathrm{toppebble}(x))$ is true with binding B (because $\mathrm{lift} \in \mathrm{II}_{M,\psi}$), so because we know that ψ is true with binding B, we know that $\exists x : (\mathrm{head}(x) \wedge \mathrm{toppebble}(x))$ is true with binding B as well. Recall that $\mathrm{toppebble}(x) \equiv \bigvee_{k\in[1,n]} (\mathrm{peb}_{k}(x) \wedge \mathrm{pebcount}_{k})$ in M. As $\mathrm{pebcount}_{k}$ is true with binding B if and only if $k = |\pi|$ (by Lemma 4.4.20), $\exists x : (\mathrm{head}(x) \wedge \mathrm{toppebble}(x))$ is true with binding B if and only if $\exists x : \big(\mathrm{head}(x) \wedge \mathrm{peb}_{|\pi|}(x)\big)$ is true with binding B. This is true if and only if $\big(\exists x : \big(\mathrm{head}(x) \wedge \mathrm{peb}_{|\pi|}(x)\big)\big)\,[\Phi]\,[f_{|\pi|}]$ is true with binding B'. Because $|\pi| > m$,

$$\begin{aligned}&\big(\exists x : \big(\mathrm{head}(x) \wedge \mathrm{peb}_{|\pi|}(x)\big)\big)\,[\Phi]\,[f_{|\pi|}] \equiv \\ &\big(\exists x : \big(\mathrm{head}(x) \wedge \mathrm{peb}_{|\pi|-m}(x) \wedge \mathrm{reachable}_{\Phi}(x)\big)\big)\end{aligned}$$

and therefore we have that $\exists x : \big(\mathrm{head}(x) \wedge \mathrm{peb}_{|\pi|-m}(x)\big)$ is true with binding B'.
- Conjunct $\exists x : \exists y : \big(\mathrm{head}(x) \wedge \mathrm{edge}_{\mathrm{drop}}(x,y)\big)$ is in $\mathrm{safe}_{\iota,|\pi|}$ iff there exists a $w \in V_{\iota_{|\pi|}}$ so that $\nu_{\iota_{|\pi|}}(w) = \langle q, \mathrm{go}_{drop}\rangle$ iff there exists a $w \in V_{\iota}$ so that

$\nu_\iota = \langle q, \text{drop}\rangle$ and $|\pi| < m$. As $|\text{left}(\pi, m)| < m$, $u_{\text{left}(\pi,m)}$ has an outgoing *drop* edge according to the definition of $\tau(t)$, so $\exists x : \exists y : \left(\text{head}(x) \wedge \text{edge}_{\text{drop}}(x, y)\right)$ is true with binding B'.

- Conjunct $\exists x : \exists y : (\text{head}(x) \wedge \text{edge}_{\text{lift}}(x, y))$ is in $\text{safe}_{\iota,|\pi|}$ iff there exists a $w \in V_{\iota_{|\pi|}}$ so that $\nu_{\iota_{|\pi|}}(w) = \langle q, \text{go}_{lift}\rangle$ iff there exists a $w \in V_\iota$ so that $\nu_\iota = \langle q, \text{lift}\rangle$ and $|\pi| \leq m$. This implies that $\psi \implies \exists x : (\text{head}(x) \wedge \text{toppebble}(x))$ is true with binding B, and as ψ is true with binding B, we know that $\exists x : (\text{head}(x) \wedge \text{toppebble}(x))$ is true with binding B. We can deduce from the definition of $\text{toppebble}(x)$ that if $\text{toppebble}(x)$ is true, then there is at least one k for which $(\text{peb}_k(x) \wedge \text{pebcount}_k)$ is true. As pebcount_k is only true with binding B when $k = |\pi|$, we have that $\exists x : \left(\text{head}(x) \wedge \text{peb}_{|\pi|}(x) \wedge \text{pebcount}_{|\pi|}\right)$ is true with binding B. Then

$$\left(\exists x : \left(\text{head}(x) \wedge \text{peb}_{|\pi|}(x)\right)\right)[\Phi]\left[f_{|\pi|}\right] \equiv$$
$$\exists x : ((\text{head}(x) \wedge \exists a : \text{edge}_{\text{lift}}(x, a)) \wedge \text{reachable}_\Phi(x))$$

is true with binding B', and so $\exists x : (\text{head}(x) \wedge \exists a : \text{edge}_{\text{lift}}(x, a))$ is true with binding B'. This predicate is clearly equivalent to $\exists x : \exists y : (\text{head}(x) \wedge \text{edge}_{\text{lift}}(x, y))$, which is therefore true with binding B' as well.

□

LEMMA 4.4.24. *Let $t \in R_{\Sigma,\Phi}$, $c = \langle q, (u, \pi)\rangle \in C_{M,t}$, $r : \langle q, \psi\rangle \to \iota \in R$, and $i \in [0, n]$. Then $r' : \langle q, \psi_i\rangle \to \iota_i$ is applicable to $cc_t(c)$ if and only if $i = |\pi|$ and $r : \langle q, \psi\rangle \to \iota$ is applicable to c.*

PROOF. Let $c' = \text{cc}_t(c) = \text{cc}_t(\langle q, (u, \pi)\rangle) = \langle q, \text{cic}_t((u, \pi))\rangle$.

($\Rightarrow$) Let r' be applicable to c'. Clearly, this implies that $\psi_i \not\equiv$ false. Then $\psi_i \equiv \psi[\Phi][f_i] \wedge \text{simpebcount}_i \wedge \text{safe}_{\iota,i}$, which implies that simpebcount_i must be true for c', which is if and only if $i = |\pi|$ (by Lemmas 4.4.20 and 4.4.22). By Lemma 4.4.23, ψ is true with binding $\left(\emptyset, \emptyset, b_{A_M,t,(u,\pi)}\right)$ if and only if $\psi_{|\pi|}$ is true with binding $\left(\emptyset, \emptyset, b_{A_{M'},\tau(t),\text{cic}_t((u,\pi))}\right)$, so because the latter statement is true (because c' satisfies $\psi_i = \psi_{|\pi|}$), the former is also true, which implies that c satisfies ψ. As the state specified in r matches the state in c as well, r is applicable to c.

($\Leftarrow$) Let r be applicable to c. We will show that $r' : \left\langle q, \psi_{|\pi|}\right\rangle \to \iota_{|\pi|} \in R'$ is applicable to c'. First of all, the state components of c' and of r' match. That leaves the condition. Because ψ is applicable to c, we know that ψ is true with binding $\left(\emptyset, \emptyset, b_{A_M,t,(u,\pi)}\right)$, which by Lemma 4.4.23 implies that $\psi_{|\pi|}$ is true with binding $\left(\emptyset, \emptyset, b_{A_{M'},\tau(t),\text{cic}_t((u,\pi))}\right)$, which implies that rule r' is applicable to c'. □

THEOREM 4.4.25. *M' performs a stepwise simulation of M using a tree transformed by τ.*

PROOF. This requires that $\mathrm{CC}_{M,t,M',\tau(t)}$ is nonempty for any tree $t \in R_{\Sigma,\Phi}$. We will prove this by showing that $\mathrm{cc}_t \in \mathrm{CC}_{M,t,M',\tau(t)}$. Consider the three conditions for membership of $\mathrm{CC}_{M,t,M',\tau(t)}$ given by Definition 4.3.1:

(1) $\mathrm{cc}_t(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) = \langle q_0, ((\mathrm{root}_t)_\lambda, \lambda)\rangle = \langle q_0, (\mathrm{root}_{\tau(t)}, \lambda)\rangle$.

(2) Let $c = \langle q_c, (u, \pi)\rangle \in C_{M,t}$, and let $r : \langle q_c, \psi\rangle \to \iota$ in R be applicable to c. By Lemma 4.4.24, rule $r' : \langle q_c, \psi_{|\pi|}\rangle \to \iota_{|\pi|} \in R'$ is applicable to $c' = \mathrm{cc}_t(c) = \mathrm{cc}_t(\langle q_c, (u, \pi)\rangle) = \langle q_c, \mathrm{cic}_t((u, \pi))\rangle = \langle q_c, (u_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi))\rangle$. We now have to establish that $\iota_{|\pi|}(c') \cong \iota(c)[\mathrm{cc}_t]$, i.e., that the fragments of intermediate output tree produced by the rules are isomorphic except for the fact that the configurations in M''s intermediate output trees are transformed by cc_t. We will in fact prove that they are equal, i.e., $\iota_{|\pi|}(c') = \iota(c)[\mathrm{cc}_t]$. The left hand side is $\iota[s_{|\pi|}][g_{c'}]$ where $g_{c'}(\langle q, \omega\rangle) = \langle q, \omega((u_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi)))\rangle$, while the right hand side is $\iota[g_c][\mathrm{cc}_t]$, where $g_c(\langle q, \omega\rangle) = \langle q, \omega((u, \pi))\rangle$. None of g_c, $g_{c'}$, $s_{|\pi|}$ and cc_t are defined for $\delta \in \Delta$, which means that nodes u with $\nu_\iota(u) = \delta \in \Delta$ will have their equivalents labeled with δ in $\iota[s_{|\pi|}][g_{c'}]$ and $\iota[g_c][\mathrm{cc}_t]$ as well. As $s_{|\pi|}$ is not total, let s be $s_{|\pi|}$ so that $s(\langle q, \omega\rangle) = s_{|\pi|}(\langle q, \omega\rangle)$ when $s_{|\pi|}(\langle q, \omega\rangle)$ is defined, and $s(\langle q, \omega\rangle) = (\langle q, \omega\rangle)$ otherwise. What remains to be proven is that $g_{c'}(s(\langle q, \omega\rangle)) = \mathrm{cc}_t(g_c(\langle q, \omega\rangle))$ for all $\langle q, \omega\rangle \in \mathrm{CI}_{M,\psi}$. We will cover all cases.

- $\langle q, \omega\rangle = \langle q, \mathrm{stay}\rangle$. We have $s(\langle q, \mathrm{stay}\rangle) = \langle q, \mathrm{stay}\rangle$, and

$$\begin{aligned} g_{c'}(\langle q, \mathrm{stay}\rangle) &= \langle q, \mathrm{stay}((u_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi)))\rangle \\ &= \langle q, (u_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi))\rangle \end{aligned}$$

  On the other hand, $g_c(\langle q, \mathrm{stay}\rangle) = \langle q, \mathrm{stay}((u, \pi))\rangle = \langle q, (u, \pi)\rangle$ and $\mathrm{cc}_t(\langle q, (u, \pi)\rangle) = \langle q, (u_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi))\rangle$.

- $\langle q, \omega\rangle = \langle q, \mathrm{go}_\phi\rangle$ for some $\phi \in \Phi$. We have $s(\langle q, \mathrm{go}_\phi\rangle) = \langle q, \mathrm{go}_\phi\rangle$ and

$$\begin{aligned} g_{c'}(\langle q, \mathrm{go}_\phi\rangle) &= \langle q, \mathrm{go}_\phi((u_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi)))\rangle \\ &= \langle q, (\phi(u_{\mathrm{left}(\pi,m)}), \mathrm{stackenc}_t(\pi))\rangle \\ &= \langle q, ((\phi(u))_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi))\rangle \end{aligned}$$

  On the other hand, we have $g_c(\langle q, \mathrm{go}_\phi\rangle) = \langle q, \mathrm{go}_\phi((u, \pi))\rangle = \langle q, (\phi(u), \pi)\rangle$, and $\mathrm{cc}_t(\langle q, (\phi(u), \pi)\rangle) = \langle q, ((\phi(u))_{\mathrm{left}(\pi,m)}, \mathrm{stackenc}_t(\pi))\rangle$.

- $\langle q, \omega \rangle = \langle q, \text{drop} \rangle$ and $|\pi| < m$. In this case, we have $s(\langle q, \text{drop} \rangle) = \langle q, \text{go}_{drop} \rangle$ and

$$\begin{aligned} g_{c'}\left(\langle q, \text{go}_{drop} \rangle\right) &= \langle q, \text{go}_{drop}\left(\left(u_{\text{left}(\pi,m)}, \text{stackenc}_t(\pi)\right)\right)\rangle \\ &= \langle q, \left(\text{drop}\left(u_{\text{left}(\pi,m)}\right), \text{stackenc}_t(\pi)\right)\rangle \\ &= \langle q, \left(u_{\text{left}(\pi,m)\cdot u}, \text{stackenc}_t(\pi)\right)\rangle \\ & \quad \langle q, \left(u_{\text{left}(\pi u,m)}, \text{stackenc}_t(\pi)\right)\rangle \end{aligned}$$

  On the other hand, we have $g_c(\langle q, \text{drop} \rangle) = \langle q, \text{drop}((u, \pi)) \rangle = \langle q, (u, \pi u) \rangle$ and $\text{cc}_t(\langle q, (u, \pi u) \rangle) = \langle q, \left(u_{\text{left}(\pi u,m)}, \text{stackenc}_t(\pi u)\right)\rangle$, and as $|\pi| < m$, $\text{stackenc}_t(\pi u) = \text{stackenc}_t(\pi) = \lambda$.
- $\langle q, \omega \rangle = \langle q, \text{drop} \rangle$ and $|\pi| \geq m$. In this case, we have $s(\langle q, \text{drop} \rangle) = \langle q, \text{drop} \rangle$ and

$$\begin{aligned} g_{c'}\left(\langle q, \text{drop} \rangle\right) &= \langle q, \text{drop}\left(\left(u_{\text{left}(\pi,m)}, \text{stackenc}_t(\pi)\right)\right)\rangle \\ &= \langle q, \left(u_{\text{left}(\pi,m)}, \text{stackenc}_t(\pi) \cdot u_{\text{left}(\pi,m)}\right)\rangle \\ &= \langle q, \left(u_{\text{left}(\pi u,m)}, \text{stackenc}_t(\pi u)\right)\rangle \end{aligned}$$

  On the other hand, we have $\text{cc}_t(g_c(\langle q, \text{drop} \rangle)) = \langle q, \left(u_{\text{left}(\pi u,m)}, \text{stackenc}_t(\pi u)\right)\rangle$, as in the previous case.
- $\langle q, \omega \rangle = \langle q, \text{lift} \rangle$ and $|\pi| \leq m$. Then we must have $\pi = \pi' u$ where $\pi' = \text{left}(\pi, |\pi| - 1)$. We have $s_{|\pi|}(\langle q, \text{lift} \rangle) = \langle q, \text{go}_{lift} \rangle$ and

$$\begin{aligned} g_{c'}\left(\langle q, \text{go}_{lift} \rangle\right) &= \langle q, \text{go}_{lift}\left(\left(u_{\text{left}(\pi,m)}, \text{stackenc}_t(\pi)\right)\right)\rangle \\ &= \langle q, \left(\text{lift}\left(u_{\text{left}(\pi',m)\cdot u}\right), \text{stackenc}_t(\pi)\right)\rangle \\ &= \langle q, \left(u_{\text{left}(\pi',m)}, \text{stackenc}_t(\pi)\right)\rangle \end{aligned}$$

  On the other hand, we have $g_c(\langle q, \text{lift} \rangle) = \langle q, \text{lift}((u, \pi)) \rangle = \langle q, (u, \pi') \rangle$ and $\text{cc}_t(\langle q, (u, \pi') \rangle) = \langle q, \left(u_{\text{left}(\pi',m)}, \text{stackenc}_t(\pi')\right)\rangle$, and as $|\pi'| < |\pi| \leq m$, we have $\text{stackenc}_t(\pi') = \text{stackenc}_t(\pi) = \lambda$.
- $\langle q, \omega \rangle = \langle q, \text{lift} \rangle$ and $|\pi| > m$. In this case we also must have $\pi = \pi' u$ where $\pi' = \text{left}(\pi, |\pi| - 1)$. Furthermore, note that $\text{stackenc}_t(\pi) = \text{stackenc}_t(\pi') \cdot u_{\text{left}(\pi,m)}$, and that $\text{left}(\pi, m) = \text{left}(\pi', m)$. We have $s(\langle q, \text{lift} \rangle) = \langle q, \text{lift} \rangle$ and

$$\begin{aligned} g_{c'}\left(\langle q, \text{lift} \rangle\right) &= \langle q, \text{lift}\left(\left(u_{\text{left}(\pi,m)}, \text{stackenc}_t(\pi)\right)\right)\rangle \\ &= \langle q, \left(u_{\text{left}(\pi',m)}, \text{stackenc}_t(\pi')\right)\rangle . \end{aligned}$$

On the other hand,

$$\begin{aligned}\mathrm{cc}_t\left(g_c\left(\langle q, \mathrm{lift}\rangle\right)\right) &= \langle q, \mathrm{lift}\left((u,\pi)\right)\rangle \\ &= \langle q, (u,\pi')\rangle \\ &= \left\langle q, \left(u_{\mathrm{left}(\pi',m)}, \mathrm{stackenc}_t\left(\pi'\right)\right)\right\rangle,\end{aligned}$$

as in the previous case.

(3) Let $c = \langle q, (u,\pi)\rangle \in C_{M,t}$ and let $c' = \mathrm{cc}_t(c) = \langle q, \mathrm{cic}_t((u,\pi))\rangle$. When a rule $r' : \langle q, \psi_i\rangle \to \iota_i \in R'$ is applicable to c', Lemma 4.4.24 implies that $i = |\pi|$ and rule $r : \langle q, \psi\rangle \to \iota$ in R is applicable to c. The equality $\iota_{|\pi|}\left(\mathrm{cc}_t(c)\right) \cong \iota(c)\,[\mathrm{cc}_t]$ was already proven in the second item of this proof. This completes the proof of the Theorem.

□

LEMMA 4.4.26. *A pebble $k \in [m+1, n]$ is local in M if and only if pebble $k-m$ is local in M'.*

PROOF. Pebbles n in M and $n-m$ in M' are both global, so we need to consider only $k < n$. Let $k \in [m+1, n-1]$ be a local pebble in M, and let $k-m$ be the corresponding pebble in M'. Let $r : \langle q, \psi\rangle \to \iota \in R$ and $r' : \langle q, \psi_i\rangle \to \iota_i \in R'$ for some $i \in [0, n]$, and let ψ_i reference relation symbol peb_{k-m}. Recall that $\psi_i \equiv \psi\,[\Phi]\,[f_i] \wedge \mathrm{simpebcount}_i \wedge \mathrm{safe}_{\iota,i}$. Two cases:

- $k < i$. If peb_{k-m} occurs in $\psi\,[\Phi]\,[f_i]$, then peb_k occurs in ψ and then $\psi_i \equiv \mathrm{false}$ by definition, which contradicts the fact that it references relation symbol peb_{k-m}. If peb_{k-m} occurs in $\mathrm{simpebcount}_i \equiv \mathrm{level}_m \wedge \mathrm{pebcount}_{i-m}$, then it must occur in $\mathrm{pebcount}_{i-m}$, and as $\mathrm{pebcount}_j$ only references pebbles numbered j and higher, this implies that $k-m \geq i-m$, i.e., $k \geq i$, which contradicts our assumption that $k < i$. Finally, if peb_{k-m} occurs in $\mathrm{safe}_{\iota,i}$, then there are two possibilities. The first is that it is in the conjunct $\neg\exists x : \mathrm{peb}_{n-m}(x)$, which only occurs when $k = n$ while we only consider $k < n$ here. The second is that it is in a conjunct $\exists x : \left(\mathrm{head}(x) \wedge \mathrm{peb}_{i-m}(x)\right)$, which implies $k = i$, which contradicts our assumption that $k < i$.
- $k \geq i$. By definition, ψ_i contains conjunct $\mathrm{simpebcount}_i$, which is defined as $\mathrm{level}_{\min(i,m)} \wedge \mathrm{pebcount}_{\max(0,i-m)}$. Furthermore, we have $\mathrm{pebcount}_{\max(0,i-m)} \implies \mathrm{free}_{\max(1,i-m+1)}$. Now observe that $\mathrm{free}_x \implies \mathrm{free}_y$ for $y \geq x$. When $i \leq m$, we then have $\mathrm{free}_1 \implies \mathrm{free}_{k-m+1}$, and when $i > m$, we have $\mathrm{free}_{i-m+1} \implies \mathrm{free}_{k-m+1}$. This implies that $M' \models \psi_i \implies \mathrm{free}_{k-m+1}$.

Now let $k \in [m+1, n-1]$ be a global pebble in M, and let $k-m$ be the corresponding pebble in M'. Then there is a rule $r : \langle q, \psi\rangle \to \iota$ where ψ references peb_k and where

$\psi \implies \text{free}_{k+1}$ is false for some tree $t \in R_{\Sigma,\Phi}$ and input configuration $(u,\pi) \in \text{IC}_{n,t}$. Then ψ is true and free_{k+1} is false with binding $B = \left(\emptyset, \emptyset, b_{A_M,t,(u,\pi)}\right)$, which implies that $k+1 \leq |\pi|$, i.e., $|\pi| > k$. Then by Lemma 4.4.23, $\psi_{\iota,|\pi|}$ is true for binding $B' = \left(\emptyset, \emptyset, b_{A_{M'},\tau(t),\text{cic}_t((u,\pi))}\right)$. As ψ references peb_k, $k > m$, and $\psi_{|\pi|} \not\equiv$ false (which means that ψ does not reference any relation symbol peb_j with $j < |\pi|$ where j is local in M), $\psi_{|\pi|}$ references peb_{k-m} in its conjunct $\psi\,[\Phi]\,[f_i]$. On the other hand, free_{k-m+1} is false with binding B' because $|\pi| > m$ and $k-m+1 \leq |\pi| - m = |\text{stackenc}_t(\pi)|$. Therefore, for the rule $\left\langle q, \psi_{|\pi|}\right\rangle \to \iota_{|\pi|}$, $\psi_{|\pi|} \implies \text{free}_{k-m+1}$ is false while $\psi_{|\pi|}$ references peb_{k-m}, so pebble $k-m$ is global in M'. □

Technically, we do not really need to know that the number of global pebbles stays the same, in order to prove that a decomposition in $k+1$ pebbles is possible, where k is the number of global pebbles in the original ptt M. Nevertheless, it is an interesting fact, as it places a lower bound on the size of the decompositions that may be achieved using this exact decomposition method.

4.4.5. Deterministic PTTs. We have gathered all of the ingredients for defining a decomposition method δ on ptts. However, it is desirable that dptt be closed under decomposition by δ, and there is no guarantee that M' will be deterministic if M is deterministic. In fact, if M is deterministic, M' will behave deterministically as long as its input tree is in $\tau(R_{\Sigma,\Phi})$ and as long as its configurations are in $\text{cc}_t(C_{M,t})$. Using this fact (which we will prove in the next Lemma), we will construct a ptt M'' from M' that *is* deterministic if M is deterministic, and that behaves exactly like M' on input trees in $\tau(R_{\Sigma,\Phi})$ and on configurations in $\text{cc}_t(C_{M,t})$. We will then prove that M'' also performs a stepwise simulation of M using a tree transformed by τ.

LEMMA 4.4.27. *Let M be deterministic, and let $t \in R_{\Sigma,\Phi}$. Then for every configuration $c \in C_{M,t}$, there is at most one rule in R' that is applicable to $\text{cc}_t(c)$.*

PROOF. Let $c = \langle q, (u,\pi)\rangle \in C_{M,t}$. As M is deterministic, there is at most one rule $r : \langle q, \psi\rangle \to \iota \in R$ that is applicable to c. Lemma 4.4.24 implies that the only rules in R' that can be applicable to $\text{cc}_t(c)$ are $r' : \langle q, \psi_i\rangle \to \iota_i \in R'$ with $i = |\pi|$. All of the components of this rule are fixed, so there is at most one such rule. □

Given that M' is deterministic in its behaviour on all configurations that occur in the simulations, we shall now define a ptt M'' that is equivalent to M' wherever M' is deterministic, and that has no true conditions in any other cases.

DEFINITION 4.4.28. Given M and M', where M is deterministic, we define $M'' = (n-m, (\Sigma,\Phi'), (\Delta,\Gamma), Q, q_0, R'')$ where $R'' = \{\langle q, \psi_i \wedge \text{excl}_{q,\psi_i}\rangle \to \iota_i \mid \langle q, \psi_i\rangle \to \iota_i \in R'\}$

with $\text{excl}_{q,\psi_i} \equiv \bigwedge_{\psi' \in X} \neg\psi'$ where

$$X = \{\psi'_j \mid \langle q, \psi'_j\rangle \to \iota'_j \in R' - \{\langle q, \psi_i\rangle \to \iota_i\} \text{ and } \psi'_j \text{ does not reference } \text{peb}_{k-m} \text{ if } k-m \text{ local in } M' \text{ and } k < i\}.$$

LEMMA 4.4.29. *M'' is a ptt.*

PROOF. This can be derived from the fact that M' is a ptt. First of all, $\text{out}_{R''} = \text{out}_{R'}$ clearly satisfies the conditions placed upon it by Definition 3.2.1(6). Furthermore, whenever a condition $\psi_i \wedge \text{excl}_{q,\psi_i}$ is true, then ψ_i is true as well, which implies that if $\iota \in I_{M,\psi_i}$, then $\iota \in I_{M,\psi_i \wedge \text{excl}_{q,\psi_i}}$ as well. □

LEMMA 4.4.30. *Let $t' \in T_{\Sigma,\Phi'}$, let $c' = \langle q, (u, \pi)\rangle \in C_{M',t'} = C_{M'',t'}$, and let $R'_{c'}$ be the set of all rules in R' that are applicable to c'. Then rule $r'' : \langle q, \psi_i \wedge \text{excl}_{q,\psi_i}\rangle \to \iota_i \in R''$ is applicable to c' if and only if $R'_{c'} = \{\langle q, \psi_i\rangle \to \iota_i\}$.*

PROOF. Let r be the rule $\langle q, \psi_i\rangle \to \iota_i$, and let r'' be the rule $\langle q, \psi_i \wedge \text{excl}_{q,\psi_i}\rangle \to \iota_i$.

($\Leftarrow$) Let $R'_{c'} = \{r\}$. Then ψ_i is true for c', and for all $r' : \langle q, \psi'_j\rangle \to \iota'_j \in R'$, $r' \neq r$, ψ'_j is false for c' (as $r' \notin R'_{c'}$). Condition $\psi_i \wedge \text{excl}_{q,\psi_i}$ asserts that ψ_i is true, and asserts the falseness for a subset of the set of conditions that we just determined were false. Therefore, r'' is applicable to c'.

($\Rightarrow$) Let r'' be applicable to c'. Then conjunct ψ_i is true for c', which means that $r \in R'_{c'}$. Now assume that $R'_{c'} \neq \{r\}$. That implies that there is a rule $r' : \langle q, \psi'_j\rangle \to \iota'_j \in R'_{c'}$, $r' \neq r$, and that ψ'_j is true for c'. But then also $r' \in R'$, which implies that either excl_{q,ψ_i} contains a conjunct $\neg\psi'_j$, or ψ'_j references peb_{k-m}, $k-m$ local in M', $k < i$. The former case would contradict our assumption that ψ'_j is true. In the latter case, $\psi'_j \implies \text{free}_{k-m+1}$. Also, $\psi_i \implies \text{simpebcount}_i$, and as peb_{k-m} is used, $k > m$, which implies $i > m$, which in turn implies that $\text{simpebcount}_i \implies \text{pebcount}_{i-m} \implies \exists x : \text{peb}_{i-m}(x)$, while free_{k-m+1} implies the opposite because $k-m+1 \leq i-m$. This contradicts our assumption that ψ'_j is true. Therefore, $R'_{c'} = \{r\}$. □

LEMMA 4.4.31. *M'' is deterministic.*

PROOF. Let $r''_1 : \left\langle q, \psi_i^{(1)} \wedge \text{excl}_{q,\psi_i^{(1)}}\right\rangle \to \iota_i^{(1)}$ and $r''_2 : \left\langle q, \psi_j^{(2)} \wedge \text{excl}_{q,\psi_j^{(2)}}\right\rangle \to \iota_j^{(2)}$ be distinct rules in R''. Then there are distinct rules $r'_1 : \left\langle q, \psi_i^{(1)}\right\rangle \to \iota_i^{(1)}$ and $r'_2 : \left\langle q, \psi_j^{(2)}\right\rangle \to \iota_j^{(2)}$ in R'. Let $t' \in T_{\Sigma,\Phi'}$ and let $c' \in C_{M',t'} = C_{M'',t'}$, and assume that both r''_1 and r''_2 are applicable to c'. By Lemma 4.4.30, this implies that the set of rules in R' that are applicable to c' is equal to $\{r'_1\}$ (because r''_1 is applicable to c') and to $\{r'_2\}$ (because r''_2 is applicable to c'), and $\{r'_1\} \neq \{r'_2\}$, which is a contradiction. Therefore, r''_1 and r''_2 cannot both be applicable to c', and therefore there is at most one rule that can be applicable to any configuration $c' \in C_{M'',t'}$, which implies that M'' is deterministic. □

LEMMA 4.4.32. *If M is deterministic, then M'' performs a stepwise simulation of M using a tree transformed by τ.*

PROOF. We verify that $\mathrm{cc}_t \in \mathrm{CC}_{M,t,M'',\tau(t)}$, using the conditions specified in Definition 4.3.1:

(1) $\mathrm{cc}_t(\langle q_0, (\mathrm{root}_t, \lambda)\rangle) = \langle q_0, (\mathrm{root}_{\tau(t)}, \lambda)\rangle$.
(2) Let $c = \langle q, (u, \pi)\rangle \in C_{M,t}$, and let $r : \langle q, \psi\rangle \to \iota$ in R be applicable to c. Then by Theorem 4.4.25, there is a rule $r' : \langle q', \psi'\rangle \to \iota' \in R'$ that is applicable to $\mathrm{cc}_t(c)$, and $\iota'(\mathrm{cc}_t(c)) \cong \iota(c)\,[\mathrm{cc}_t]$. As $c \in C_{M,t}$ and M is deterministic, Lemma 4.4.27 specifies that r' is the only rule in R' that is applicable to $\mathrm{cc}_t(c)$. Then by Lemma 4.4.30 there is a rule $r'' : \langle q', \psi''\rangle \to \iota' \in R''$ that is applicable to $\mathrm{cc}_t(c)$.
(3) Let $c = \langle q, (u, \pi)\rangle \in C_{M,t}$, and let $r'' : \langle q, \psi_i \wedge \mathrm{excl}_{q,\psi_i}\rangle \to \iota_i$ in R'' be applicable to $\mathrm{cc}_t(c)$. Then by Lemma 4.4.30, there is a rule $r' : \langle q, \psi_i\rangle \to \iota_i$ in R' that is applicable to $\mathrm{cc}_t(c)$. The remainder follows from Theorem 4.4.25.

□

LEMMA 4.4.33. *Let $k \in [m+1, n]$. Pebble $k-m$ is local in M' iff pebble $k-m$ is local in M''.*

PROOF. Let $k-m$ be local in M', and let $r'' : \langle q, \psi_i \wedge \mathrm{excl}_{q,\psi_i}\rangle \to \iota_i \in R''$ be a rule that references peb_{k-m}. The first possibility is that ψ_i references peb_{k-m}, in which case $M' \models \psi_i \implies \mathrm{free}_{k-m+1}$ because $k-m$ is local in M', and therefore $M'' \models \psi_i \wedge \mathrm{excl}_{q,\psi_i} \implies \mathrm{free}_{k-m+1}$. The second possibility is that excl_{q,ψ_i} references peb_{k-m}. In that case, the definition of excl_{q,ψ_i} dictates that $k \geq i$, and then either $M' \models \psi_i \implies \mathrm{simpebcount}_i \implies \mathrm{pebcount}_{i-m} \implies \mathrm{free}_{i-m+1} \implies \mathrm{free}_{k-m+1}$ (for $i \geq m$) or $M' \models \psi_i \implies \mathrm{simpebcount}_i \implies \mathrm{pebcount}_0 \implies \mathrm{free}_{k-m+1}$ (for $i < m$). This implies $M'' \models \psi_i \wedge \mathrm{excl}_{q,\psi_i} \implies \mathrm{free}_{k-m+1}$.

Now let $k-m$ be global in M'. Then by Lemma 4.4.26, pebble k is global in M. In the second part of the proof of Lemma 4.4.26 we showed that there then exists a tree $t \in R_{\Sigma,\Phi}$, a tree $\tau(t) \in R_{\Sigma,\Phi'}$, configurations $c = \langle q, (u, \pi)\rangle \in C_{M,t}$ and $c' = \mathrm{cc}_t(c) \in C_{M',\tau(t)}$ and a rule $r' : \langle q, \psi_{|\pi|}\rangle \to \iota_{|\pi|} \in R'$ so that $\psi_{|\pi|}$ references peb_{k-m}, $\psi_{|\pi|}$ is true for c', and free_{k-m+1} is false for c'. Then by the fact that M is deterministic, Lemma 4.4.27 tells us that r' is unique, and Lemma 4.4.30 then implies that $r'' : \langle q, \psi_{|\pi|} \wedge \mathrm{excl}_{q,\psi_{|\pi|}}\rangle \to \iota_{|\pi|} \in R''$ is applicable to c', which means that $\psi_{|\pi|} \wedge \mathrm{excl}_{q,\psi_{|\pi|}}$ is true for c'. However, free_{k-m+1} is still false, so $\psi_{|\pi|} \wedge \mathrm{excl}_{q,\psi_{|\pi|}} \implies \mathrm{free}_{k-m+1}$ is false and pebble $k-m$ is global in M''. □

4.4.6. Finalizing the Decomposition Method.

DEFINITION 4.4.34. $\delta_{\text{generic}} : (\text{ptt} - \text{twtt}) \to (\text{twtt} \times \text{ptt})$ is defined as $\delta_{\text{generic}}(M) = (A, M')$, where A and M' are constructed from M using the method described in this chapter using $m = \text{fg}(M)$. $\delta_{\text{det}} : (\text{dptt} - \text{twtt}) \to \text{twtt} \times \text{dptt}$ is defined for deterministic ptts as $\delta_{\text{det}}(M) = (A, M'')$, where A and M'' are constructed from M using the method described in this chapter, again using $m = \text{fg}(M)$. $\delta : (\text{ptt} - \text{twtt}) \to \text{twtt} \times \text{ptt}$ is defined as $\delta(M) = \delta_{\text{det}}(M)$ when M is deterministic, and $\delta(M) = \delta_{\text{generic}}(M)$ otherwise.

COROLLARY 4.4.35. *$(M, (A, M'))$ is a decomposition step, and if M is deterministic, then $(M, (A, M''))$ is a decomposition step as well. $\delta_{generic}$ is a total decomposition method, δ_{det} is a decomposition method that is total on dptt, and δ is a total decomposition method. Furthermore, dptt is closed under decomposition by δ_{det} and δ.*

PROOF. That $(M, (A, M'))$ is a decomposition step follows directly from Theorems 4.4.25 and 4.4.10, Lemma 4.3.8, and Definition 4.2.1, which defines the concept of a decomposition step. Lemma 4.4.32 provides the remaining proof for the fact that $(M, (A, M''))$ is a decomposition step. By Definition 4.2.2, a function δ is a decomposition method if all $(M, (A, M')) \in \delta$ are decomposition steps, which is the case for δ_{generic}, δ_{det} and δ. Furthermore, δ_{generic} is total and δ_{det} is total on dptt because the methods described in this chapter do not place any constraints on their input M. The fact that δ is total can be easily derived from the fact that δ_{generic} and δ_{det} are total. That dptt is closed under decomposition by δ_{det} follows from Lemma 4.4.31 and Lemma 4.4.8, and that dptt is closed under decomposition by δ follows from the fact that $\delta(M) = \delta_{\text{det}}(M)$ if $M \in \text{dptt}$. □

THEOREM 4.4.36. *A ptt M can be decomposed into $k+1$ twtts, where k is equal to the number of global pebbles of M. If M is deterministic, then M can be decomposed into $k+1$ deterministic twtts.*

PROOF. By repeatedly applying the decomposition method δ, we know we can decompose M into some number of twtts. At each decomposition step $\delta(M_i) = (A_i, M_{i+1})$ $(M_1 = M)$, the number of twtts in the total decomposition is increased by one (in the form of A_i), while the number of global pebbles remaining in the ptt is reduced by *exactly one*, because the decomposition method δ encodes all pebbles *up to and including the first global pebble* into the tree, while the remaining pebbles in M_{i+1} have the same number of global pebbles (by Lemma 4.4.26 and Lemma 4.4.33). After applying $k-1$ decomposition steps, we have ptt M_k which contains only one global pebble, and as the definition of local pebble does not allow the highest-numbered pebble to be local, the global pebble *must* be the last one. The step $\delta(M_k) = \delta(A_k, M_{k+1})$ then leaves a ptt M_{k+1} with zero pebbles, because $\delta(M_k)$ will encode all pebbles up to and including

the first global pebble, which is the last pebble. The ptt M_{k+1} with zero pebbles is also a twtt, and combined with the twtts $A_1, \ldots, A_k$ this gives a total of $k+1$ twtts in the decomposition. If M is deterministic, then all of the twtts will be deterministic because dptt is closed under decomposition by δ. $\square$

CHAPTER 5

Application to XML Transformation Languages

5.1. Introduction

In this chapter, we will explore the capabilities of our extended Pebble Tree Transducer to perform XML transformations. We will begin by providing some background into XML, XML transformations, XML document types, and the role of the pebble tree transducer in the type checking of XML transformations.

5.1.1. Historical Background. XML [**23**] was designed to be a leaner replacement for SGML, a document markup language that was standardized in 1986 [**2**]. Standardized by the WWW Consortium (W3C) in 1998, its primary focus was on the storage and interchange of documents. In particular, it was to be the basis for XHTML, a replacement for HTML, the SGML-based language for documents on the World Wide Web. Shortly after the XML standard was published, the WWW Consortium published a document-oriented transformation language called XSL Transformations (XSLT) [**7**], whose structure was highly suitable for presentational transformations, such as would be needed to transform XML data sets into XHTML documents. However, XML was also being used as a generic format for storage of data, i.e., for databases, and XSLT was not too well suited to the types of transformations or "queries" normally used on databases. This left a vacuum, which was soon filled by a myriad of database-oriented XML query languages. Eventually, the WWW Consortium embarked on the standardization of a query language called XML Query or XQuery [**22**], whose semantics are not unlike those of SQL, the standard query language for relational databases. At the time of this writing, the XQuery standardization process is nearing completion, while the second version of XSLT [**16**] is nearly finished as well.

5.1.2. XML Document Structure. An XML document has a tree structure, where every node has a textual label (which is called a *tag* in XML terminology), a set of name-value pairs called *attributes*, and a sequence of child nodes. In an XML document, a node having tag `foo` is represented by a *start-tag* `<foo>` and an *end-tag* `</foo>`. The start-tag and end-tag enclose the XML that represents the child nodes of the `foo` node, e.g.:

```
<foo>
  <childnode1> ... </childnode1>
```

```
  <childnode2> ... </childnode2>
</foo>
```

If a node has no child nodes, then the start-tag and end-tag may be written as a combined start-end tag `<foo />`, which is short for `<foo></foo>`. The attributes of a node are listed within the start-tag (or combined start-end tag) of the node, like this:

```
<foo attributename1=''value1'' attributename2=''value2''>
  ...
</foo>
```

At the highest level, an XML document consists of a string of such node descriptions. Technically, an XML document has an implicit root element (called the *document entity*) that is not included in the textual representation but that does exist in the conceptual structure of the document.

When we translate this description of XML to tree-theoretical terminology, an XML document is essentially an unranked tree. Although the attributes and the child nodes are disjoint sequences of "subnodes" associated with a node, these sequences can be unified by viewing the attributes of a node as specially marked child nodes. In their original paper on pebble tree transducers, Milo et al. [**17**] argued that XML's unranked trees can be encoded as ranked trees. While the graph-based tree model that we use for our pebble tree transducers provides us with more freedom than ranked trees, it is not capable of encoding XML natively either: the number of child nodes of each node is limited by the number of available edge labels. We will consider an encoding of XML using our tree model in Section 5.3.1. In contrast with the encoding used by Milo et al., our encoding will include support for data values and an unbounded tag space.

5.1.3. Document Types and Transformation Languages. The XML standard allows an XML document to declare that it conforms to the restrictions of a *document type*, specified by a Document Type Definition (DTD). For example, an XHTML document declares that it is an XHTML document by referring to the DTD for XHTML. A DTD specifies the following things:

- a list of allowed node labels;
- for each node label, the allowed attribute names; and
- for each node label, the allowed child nodes, in terms of a regular expression over node labels.

Essentially, a DTD specifies a restricted kind of regular tree language consisting of XML document trees, using the concept of a regular tree language over unranked trees as described by Brüggeman-Klein and Wood [**6**]. Milo et al. [**17**] describe DTDs as extended context-free grammars, i.e., context-free grammars where the right-hand sides of productions are regular expressions over the node label alphabet. Note that

DTDs are not able to fully specify a regular tree language, as they only specify the relationships between a node's label and the order of the labels of its child nodes. However, the concept of a "document type" is usually extended to mean a regular tree language, and more recent, next-generation XML type or "schema" languages such as RELAX NG [**1**] and XML Schema [**14**] are able to describe more of the regular tree languages. In a survey by Murata et al., [**18**], it is shown that XML Schema is able to describe a specific subset of the regular tree languages that they call *single-type* regular tree languages (which are comparable to the tree languages recognizable by a deterministic top-down tree automaton), while RELAX NG is able to describe all regular tree languages. Type definitions that can fully describe the regular tree languages are referred to as *generalized* DTDs.

XSLT and XQuery transformations are allowed to specify the output type that their result will have, i.e., that the result will conform to a specific DTD. Unfortunately, the transformation languages' complicated semantics make it hard to verify such a claim statically; and performing the verification dynamically adds yet another step to the already labor-intensive transformation process. Milo et al. [**17**] showed that, when they modelled XML transformations using pebble tree transducers, a static type check of the form "if the input tree conforms to type X, then the output tree conforms to type Y" was decidable. Unfortunately, they did not provide too much information on exactly which parts of the commonly used XML transformation languages could be modelled by a pebble tree transducer. In the sections that follow, we will attempt to provide more detailed insights regarding the extent to which pebble tree transducers can model the capabilities of XSLT and XQuery.

5.1.4. Chapter Overview. In Section 5.2, we will look at the two XML transformation languages that have been standardized by the WWW Consortium, XSLT 2.0 and XQuery 1.0, and at their common subset, XPath 2.0. In Section 5.3 we then discuss how the features of these languages match up with the capabilities of a ptt, and how the features may be implemented (if they can be implemented at all). For all features, we will consider MSO-based implementations as well as implementations that do not use MSO, in order to assess the extra capabilities provided by the addition of MSO. Where the capabilities of a ptt are not sufficient for emulating XSLT or XQuery, we will consider these deficiencies in the light of using the pebble tree transducers for type checking purposes, i.e., we will consider an *approximation* of the XSLT or XQuery construct, and we will then consider whether a type check would yield the same result using that approximation. The way in which a ptt can simulate XML transformations is illustrated in Section 5.4 using a sample XQuery transformation. The chapter concludes with a section that deals with the number of global pebbles that XQuery

transformations require, and a method for minimizing the number of global pebbles required for the evaluation of certain queries.

5.2. XML Transformation Languages

5.2.1. XPath. If we want to explain the XSLT and XQuery transformation languages, we must begin by explaining *XPath expressions* **[20]**, which are the common ground between the two models. There are two kinds of XPath expressions: *path expressions*, which navigate the tree structure of an XML document to calculate sets (or, more accurately, *sequences*) of nodes, and other expressions, which calculate values. XPath terminology does not have a separate name for these other expressions, but we will refer to them as *value expressions*.

5.2.1.1. *Path Expressions.* In short, one can describe a path expression as a set of pairwise and per-node constraints on strings of nodes of a fixed length. A pairwise constraint (called *axis* in XPath terminology) is always between two consecutive nodes, and states a tree relationship, i.e., "child", "descendant", "ancestor", "following" (in the so-called "document order", which is defined as a left-to-right pre-order tree walk), "following-sibling" and the like. The per-node constraints consist of two parts: a *node test* and a *predicate*. A predicate is a boolean XPath value expression. The node test is a redundant construct that provides a subset of the functionality of predicates using a more convenient notation. Using a node test, one can express a constraint on the label of the node, and on the kind of node (node or attribute). The result of an XPath path expression is calculated by performing *pattern matching* (a term that stems from the fact that path expressions are referred to as *patterns* in XSLT terminology). The pattern matching operation calculates a set of *match* nodes from a given starting node, or *context node* in XPath terminology. Conceptually, it works as follows: one takes *all* strings of nodes that start with the context node and that satisfy the constraints of the path expression. Given a string of nodes, one can easily verify whether or not the node string satisfies the constraints of the path expression. Of each of these matching node strings, the last node is a *match*. The result of the pattern matching is a sequence consisting of all matches, in document order, with no duplicates. The "no duplicates" clause is significant, since a node may be a match for a path expression through more than one string of nodes!

The following is an example of a path expression:

```
descendant::foo[attribute::bar='xyz']/child::baz
```

This path expression specifies constraints on a string of three nodes, starting with the context node. The first constraint is `descendant::foo[attribute::bar='xyz']`. In this constraint, the *axis* is given by `descendant::`, which specifies that the second node must be a descendant of the context node. The axis is followed by a node test and

a predicate: node test `foo` specifies that the node label of the second node should be `foo`, and predicate `attribute::bar='xyz'` specifies that attribute `bar` of the second node should have value `'xyz'`. The second constraint, `/child::baz`, indicates that the path expression should locate another node, which must be a child node of the second node, and whose node label should read `baz`. This third node is a *match* of the path expression.

Path expressions also support the following syntactical conventions:

- `child::` is the default axis (e.g. `foo/bar` stands for `child::foo/child::baz`) unless the separator is `//`, in which case the default is `descendant::`, e.g. `foo//baz` stands for `child::foo/descendant::baz`.
- `@` is short for `attribute::`.
- A path expression that starts with `/` indicates that the second node of the node string must be the root of the tree, e.g. `/foo` matches all nodes labeled "foo" that are children of the root. One can also start the path expression with `//`, which is short for `/descendant::`.
- The path expression `.` indicates the context node itself.

5.2.1.2. *Value Expressions.* "Value expressions" are like regular mathematical expressions, i.e., they calculate a result by applying operators to values. In calculating their results, they have access to the tags and attribute values of nodes, as well as externally defined variables (which we will discuss later). Value expressions can access the current node, and they can access other nodes in the tree by navigating to them using a path expression. As mentioned in the preceding paragraph, XPath value expressions can be used as per-node constraints in path expressions; in this situation, the expression's current node is the node to which the constraint is being applied. For instance, in the path expression example, the value expression `attribute::bar='xyz'` retrieves an attribute from the current node, which is a potential match for the second node in the full path expression, i.e., a matching node for `descendant::foo` that is being tested with the predicate `[attribute::bar='xyz']`. Value expressions are also used for other purposes, e.g. conditionally generating output. The expressions can become arbitrarily complex: they have at their disposal a large library of functions, they can perform calculations, and so on. Furthermore, not only can they perform calculations on single values, they can also deal with *sequences* of values, which they can filter, sort, merge and aggregate. There is one limitation that should be mentioned, because it has important implications for the simulation of XPath using pebble tree transducers: per-node constraints have no way of refering to other intermediate nodes within the pattern expression.

5.2.2. XSLT. The basic entity of the XSLT transformation language is the *template rule*. A template rule consists of a *pattern*, which is a path expression that determines the nodes to which the template applies, and a so-called *sequence constructor* that determines the output tree structure that is to be produced for the input nodes that match the pattern. A template looks like this:

```
<xsl:template match="foo">
  <bar />
  <baz />
</xsl:template>
```

Note that XSLT transformations are themselves XML documents: an XSLT template is described by an XML node. This particular template matches any node having label `foo`, and replaces it with the output of a sequence constructor, which in this case consists of two nodes, one labeled `bar` and one labeled `baz`. The way that the pattern is "matched" is somewhat different from normal path expressions: a node matches a template's pattern p if the node is a match for path expression p *starting from at least one of the nodes in the tree*. For instance, if a template's pattern is `foo/bar/baz`, then any node matching `//foo/bar/baz` matches the pattern.

Using its list of templates, an XSLT transformation is able to transform a sequence of nodes by iterating over the nodes and by selecting, for each node, the highest-priority template for which the node is a match. For each node for which an appropriate template exists, the transformation then generates output by instantiating the sequence constructor of the template. An XSLT sequence constructor is basically a sequence of output nodes, that may contain fixed output but that may also contain XSLT instructions to specify how specific parts of the output tree are to be generated. When a sequence constructor is evaluated, the instructions can refer to the node that was matched as the *context node*, i.e., the instructions can evaluate nested path expressions starting from the matched node, evaluate XPath value expressions on the matched node, or they can copy the matched node. When an instruction is applied to nodes selected using a pattern expression, these nodes are called the "selection". The instructions cover the following things (leaving out some redundant constructs) :

- `xsl:apply-templates`: generate output by applying appropriate templates to a selection.
- `xsl:for-each`: for each node in a selection, include the output of a specified sequence constructor. Optionally process the selection in a specific order.
- `xsl:if`: include the output of a specified sequence constructor depending on a boolean value expression.

- `xsl:copy` and `xsl:copy-of`: Output a copy of the context node resp. a selection. The `xsl:copy` instruction generates a *shallow* copy, i.e., it copies only the node itself, while the `xsl:copy-of` instruction generates a *deep* copy, i.e., a copy of the node itself and also the complete subtree below it.
- `xsl:message`: show the user a message, and optionally abort the translation.
- `xsl:variable`: bind the result of an expression to a named variable. A variable definition has no child nodes to indicate its scope: instead, it is visible throughout the remainder of the sequence constructor in which it is contained, and in the nested sequence constructors invoked by the processing instructions contained therein. Variable definitions are local to the template within which they are defined, i.e., if a variable is defined in one template, and another template is applied while the variable is in scope, the variable's value is not visible in the invoked template.
- `xsl:with-param`: templates may specify that they take parameters using the `xsl:param` instruction; in the `xsl:apply-templates` instruction, the caller may specify values for the parameters using `xsl:with-param`.

The following is an example of an XSLT transformation:

```
<xsl:template match=''/''>
  <bar>
    <xsl:apply-templates select=''*'' />
  </bar>
</xsl:template>
<xsl:template match=''foo''>
  <baz/>
</xsl:template>
<xsl:template match=''*''>
  <xsl:if test=''@pebbles != 3;''>
    <xsl:copy />
  </xsl:if>
</xsl:template>
```

This XSLT transformation consists of three templates. The first template has match expresson `/` , which only matches the root of the XML document. This is, in effect, the starting point of the transformation: an XSLT transformation's processing starts by applying a template to the root of the input document, and this template will be the highest-priority template that matches this root node. The first template's sequence constructor defines a single node with label `bar`, with child nodes which are found by applying templates to all children of the document's root. For all of these nodes, the

only candidate templates are the second and third templates, as none of the child nodes of the root node are equal to the root node, and therefore the first template will never apply. The second template matches any node labeled `foo`, while the third template matches any node at all – except for those labeled `foo`, because the second template has a higher priority. The second template simply outputs a node labeled `baz` for every `foo` node encountered, while the third template copies the input node, but only if the value of its `pebbles` attribute is not equal to 3. Now look what happens when this transformation is applied to the following XML document:

```
<foo />
<betty pebbles=''3'' />
<foo />
<barney pebbles=''5''>
  <bambam pebbles=''5''/>
</barney>
```

The output of the transformation will be:

```
<bar>
  <baz />
  <baz />
  <barney pebbles=''5'' />
</bar>
```

Note that the `betty` node has not been copied, since its `pebbles` attribute was equal to 3. It has actually been processed by the third template, but as the condition of the `xsl:if` was not met, the final output was empty. Note also that, although the `barney` node has been copied, the copy is *shallow*, and the `bambam` child node of `barney` has not been processed. This is because the sequence constructor that processed the `barney` node (the third template) only copied the node itself using an `xsl:copy` instruction, but did not specify that anything should have happened to its children – if it wanted to have included its children in the output, processed by templates, it should have specified `<apply-templates />`.

One additional feature of XSLT that we should mention is the concept of a *mode*, a concept that is very similar to a pebble tree transducer's state. Template rules may specify that they are only valid in a certain set of modes, like this:

```
<xsl:template match=''*'' mode=''mymode''>
</xsl:template>
```

The `xsl:apply-templates` instruction may specify, using the `mode` attribute, that only templates should be applied that apply to a given mode:

```
<xsl:apply-templates mode=''othermode''>
```

In this example, the template rule defined above does not apply, because it only applies to mode `mymode` and not to mode `othermode`. One special characteristic of modes is that they are *persistent*: if an apply-templates instruction specifies a specific mode, then *all processing* as a result of that apply-templates instruction takes place in that mode – unless, of course, a nested apply-templates instruction overrides it by specifying a different mode.

5.2.2.1. *A Note on XSLT Versions.* Although we limit ourselves to XSLT 2.0 in this chapter, the differences between the XSLT 1.0 and the XSLT 2.0 processing models are not such that our conclusions do not apply to XSLT 1.0. The main difference between XSLT 1.0 and XSLT 2.0 is that XSLT 2.0 supports a richer and better-defined data model, using schema information to be able to work with typed data values. Although the processing model has been changed or clarified in many small details, the general structure remains the same.

5.2.3. XQuery. While XQuery also uses XPath expressions to locate nodes and to perform calculations, the XQuery transformation model is quite different from XSLT. Where the XSLT paradigm is based on recursive application of template rules that provide a binding between a node and its transformed output, the XQuery paradigm is based on a more direct node search and iteration concept called the *FLWOR expression*. FLWOR (pronounced "flower") stands for the clauses of an FLWOR expression, which are *For*, *Let*, *Where*, *Order By* and *Return*. The function and structure of FLWOR expressions is similar to the SQL *SELECT* statement. The *For* clause is similar to an entry in an SQL *FROM* clause, the *Where* and *Order By* clause are exactly like their SQL counterparts of the same name, and the *Return* clause is similar to SQL's *SELECT* clause, the main difference being that it does not return a tuple but a tree. The main difference from SQL is in the *Let* clause. The *Let* clause specifies a node set, like the *For* clause, but there is a difference: while all the sets of nodes specified in the *For* clause are "joined" (the query evaluation iterates over every combination of input tuples), the *Let* clause binds a set of nodes to a single variable of the *sequence* type. In the other clauses, such a set can be processed in all sorts of ways: aggregations can be calculated, the sequence's nodes can be copied in the output, the sequence can be used in a nested *For* clause, and so on.

Here is an example of a FLWOR expression (similar to an example from the XQuery standard):

```
for $d in /departments
let $e := /employees/employee[@deptno = $d/@deptno]
where fn:count($e) >= 5
order by fn:count($e) descending
```

```
return
  <big-department>
    <deptno>
      {$d/@deptno}
    </deptno>
    <headcount>
      {fn:count($e)}
    </headcount>
  </big-department>
```

Note that, as opposed to XSLT transformations, XQuery queries are not XML documents. This FLWOR expression finds all departments, for which it finds all employees in that department, it narrows down the department selection to those having at least five employees, then sorts the results by number of employees (in descending order) and then returns a `<big-department>` node for each result, including child nodes representing the department number and the headcount of the department.

5.3. Pebbles and XML Transformation Languages

We will now begin our investigation into how XPath path expressions and value expressions, XSLT transformations, and XQuery queries may be implemented using pebble tree transducers. We attempt to provide full coverage of all features of the languages (modulo redundancy), so that we achieve a complete image of the things which a pebble tree transducer can and cannot do. For every feature, we will first consider an implementation that works for an "MSO-restricted" pebble tree transducer, that we define as follows.

DEFINITION 5.3.1. An *MSO-restricted* ptt is a ptt whose conditions' truth values can be expressed as $\exists h : \text{head}(h) \wedge \psi$, where ψ is a boolean combination of the following atomic formulas and predicates:

- $\text{lab}_\sigma(h)$, where σ is a node label.
- $\exists x : \big(\text{edge}_\phi(h, x) \wedge \text{edge}_{\phi'}(x, h)\big)$, where ϕ and ϕ' are edge labels.
- $\text{peb}_k(h)$, where k is a pebble number.
- $\text{istoward}_{\phi,k}(h)$, which is a predicate that is true iff edge ϕ is the first edge on the shortest path toward pebble k.

An MSO-restricted ptt has facilities similar to those available in earlier ptt models that did not allow the use of MSO predicates. The "istoward" facility is new, but it is necessary to provide a "sense of direction" in our tree model, that does not have a concept of "up" or "down". In a sense, it replaces the fact that earlier ptt models had an implicit "is up" test for edges. We will discuss this in more detail in Section 5.3.2.

For each transformation language feature, after we consider an MSO-restricted implementation, we will discuss how an implementation using the full power of MSO logic could improve upon such an implementation. At the end of the section, we will discuss some of the more difficult issues, namely recursion, data-value joins, complex data-value calculations, and sorting.

5.3.1. XML Representation. In order to be able to discuss implementations of XML-based constructs in a ptt, we will need to assume some sort of representation of XML documents as rooted trees. We will assume the following straightforward encoding:

- There are two types of nodes that represent XML nodes: "tag" nodes and "attribute" nodes. These node types are identified by their node labels only as being "tag" or "attribute" nodes; the actual tag and attribute names are encoded in child nodes (as we will discuss below).
- A "tag" node is linked to its children by an edge labeled "firstchild", that points to the first node in its sequence of child nodes (with a reverse edge labeled "parent"). The remainder of the child nodes are then reachable by following the "next" edges (with reverse edges labeled "prev").
- Similarly, a "tag" node is linked to its attributes by an edge labeled "firstattribute", that points to the first node in its set of attributes (with a reverse edge labeled "parent"). The remainder of the attribute nodes are then reachable by following the "next" edges (with reverse edges labeled "prev").
- Unbounded values in an XML document are represented by a linked list of nodes, where the node label represents a finite chunk of the value, and an edge labeled "next" leads to the next chunk (and the reverse edge is labeled "prev"). A node that has an unbounded value associated with it identifies this value by an edge that leads to the node representing the first chunk of the value; this edge is labeled appropriately to identify the meaning of the value (e.g., "tag" or "attribute", see below), while the reverse edge is always labeled "owner". (In graphic representations, we will always depict unbounded values as single nodes, which are labeled with the unbounded value, in quotes. This does not reflect what a *real* unbounded value representation would look like; most likely, such a representation would use a node for each character in the unbounded value, or even for each bit.)

- A "tag" node's tag name is an unbounded value, identified by an edge "tag".[1] An "attribute" node has two unbounded values, identified by edges "name" and "value".
- The "document entity", i.e., the implicit root node of every XML document, is encoded as a "tag" node at the root node of the tree, with no tag, only children. This conforms to the way the document entity is treated in XML.
- We consider text nodes a redundant construct, so we will not consider them at all. We view them as equivalent to, say, XML nodes with a special tag `text`, having an attribute `value` that specifies the textual value; everything that can be implemented for such nodes can be implemented for text nodes as well.

Figure 5.3.1 shows an example of this representation for the following XML fragment:

```
<foo att=''35''>
  <bar />
  <baz />
</foo>
<bambam />
```

To prevent clutter, the Figure only shows edges in one direction; the reverse edge labels are indicated in brackets. Even though the graph display was already simplified, it remains large and difficult to interpret. In what follows, we will generally abstract away some of the details of the graph representation when we display graphs. In particular, we will generally display a "tag" node and the nodes representing its tag value as a single node, labeled with the tag value. An "attribute" node is displayed as a single node labeled *attributename="value"*. In addition, we will display only downward edges (i.e., the edges having labels "tag", "firstattribute", "name", "value", "firstchild"); the reverse edge labels are uniquely determined by the downward edge labels, so we can safely omit them from the display. Figure 5.3.2 shows the same graph as in Figure 5.3.1, but with these abstractions in place. One should remain aware that the abstracted version of an XML graph is only an abbreviation for the full version; the abstraction is simply for display purposes, nothing else.

5.3.2. Path Expressions. Path expressions can generally be evaluated using pebble tree transducers. We will discuss three approaches: a naive algorithm that does not satisfy all of the constraints imposed by XPath, a fully XPath-compliant algorithm, and an extension of that algorithm using MSO logic. Futhermore, we will discuss an approach that work only for certain types of path expressions.

[1]If the input document conforms to a DTD, then the number of possible tag values is finite and the tag can be represented in the node label. However, a well-formed XML document does not need to conform to a DTD, so we assume that tags are unbounded values.

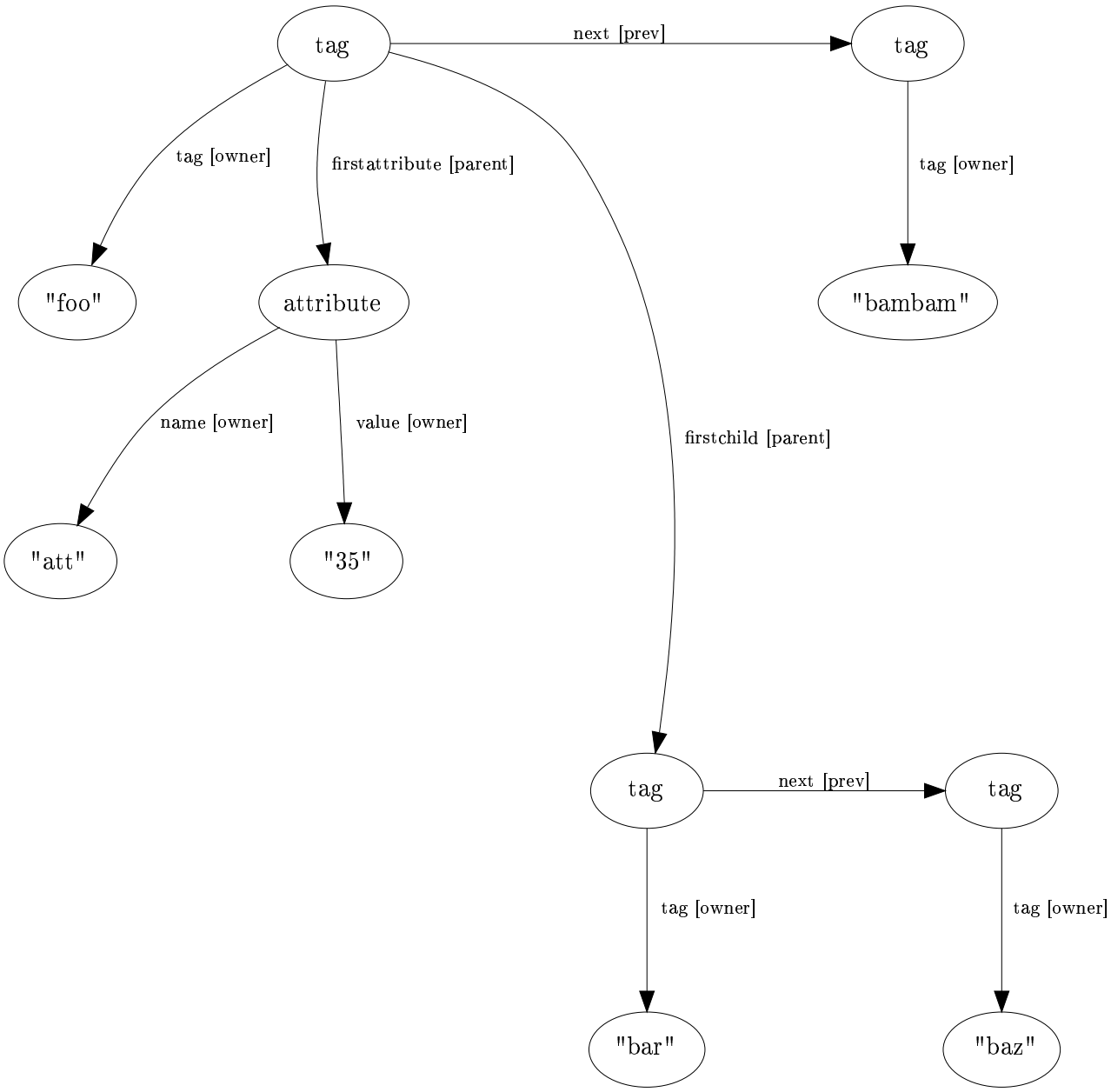


FIGURE 5.3.1. An example of the graph representation of an XML fragment.

ALGORITHM 5.3.2. *A naive algorithm for evaluating a path expression.*

(1) *Drop a pebble on the current node.*
(2) *Walk through all nodes in the tree that satisfy the next pairwise constraint. (As the pairwise constraints are simple tree-structural constraints, they can be implemented using a tree walk.) For each visited node:*
   (a) *Evaluate the per-node constraints. If they are not satisfied, continue with the next node.*

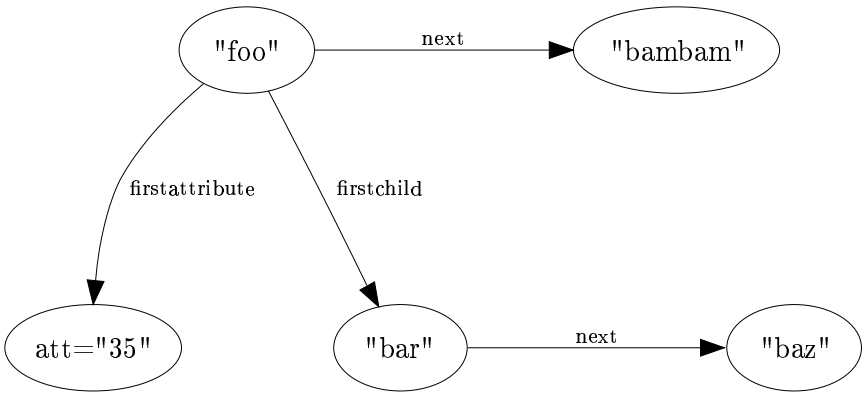


FIGURE 5.3.2. An example of the abstracted graph representation of an XML fragment.

(b) *If we are at the last pairwise constraint in the path expression, we have a match. Execute everything that needs to be done for a match.*
(c) *If we are* not *at the last pairwise constraint in the path expression, recursively execute steps 1, 2 and 3 to find the nodes satisfying the next pairwise constraint relative to the node that we are currently at.*

(3) *Lift the pebble dropped in step 1.*

For a path expression over strings of n nodes, this algorithm drops n local pebbles and no global pebbles (excluding pebbles dropped for evaluating per-node constraints). The algorithm uses the stack of local pebbles to find the matches of a path expression using the obvious implementation of n nested loops, one loop for each "step" or pairwise constraint in the path expression. Unfortunately, this algorithm has several drawbacks. First of all, the algorithm does not filter out duplicates: the algorithm will find a node multiple times if there are multiple ways in which it satisfies the constraints of the path expression, i.e., if there are multiple strings of intermediate nodes for which the pattern's constraints are satisfied. Also, the output of XPath is defined to be given in document order, and this algorithm does not generate its output in that order. The advantage of the algorithm is that it requires no global pebbles. Now consider the following algorithm.

ALGORITHM 5.3.3. *Alternative tree-based algorithm for evaluating a path expression.*

(1) *Drop a pebble on the starting node.*
(2) *Walk through all nodes in the tree, in pre-order. These nodes are the candidate matches. For each visited node:*
   (a) *Evaluate a* reversed *version of the path expression, using Algorithm 5.3.2. Each time a "reverse match" of the path expression is found, compare*

the reverse match with the starting node. If the reverse match is the starting node, then the candidate match is an actual match. When this has happened, remember this in the state.

(b) *If the algorithm in (a) determined that the candidate match was an actual match, execute everything that needs to be done for the match.*

(3) *Lift the pebble dropped in step 1.*

For a path expression over n nodes, this algorithm uses one global pebble (in Step 1) and n local pebbles (in Step 2), again excluding any pebbles used in evaluating per-node constraints. This algorithm does not have the drawbacks of the previous algorithm: it generates the matches in the correct order, and it generates each match only once. It has a different drawback, however: it requires the use of a single global pebble, as the matches of the reverse pattern matching must be compared to the starting node.

In the preceding algorithms, we have not considered the possibility of using MSO logic to check the potential matches. This is possible in cases when the per-node conditions are MSO-implementable, using the following algorithm.

ALGORITHM 5.3.4. *Evaluate a path expression using MSO logic. Let the path expression to be evaluated be a path expression over n nodes, i.e., $n-1$ steps using $n-1$ per-node constraints.*

(1) *Drop a pebble m on the starting node.*

(2) *Walk through all nodes in the tree, in pre-order. At each visited node:*

(a) *Evaluate the MSO predicate*

$$\exists x, y : peb_m(x) \wedge pathexp(x, y) \wedge head(y)$$

where $pathexp(x, y) \equiv$

$$\exists x_1, \ldots, x_n : \left((x_1 = x) \wedge (x_n = y) \wedge \bigwedge_{i \in [1, n-1]} (step_i(x_i, x_{i+1}) \wedge constraint_{i+1}(x_{i+1})) \right)$$

where $step_i(x, y)$ is an MSO predicate that expresses the axis *or tree relationship required by step $i \in [1, n-1]$, and $constraint_i(x)$ is an MSO predicate that expresses the per-node constraint specified for node $i \in [2, n]$ of the path expression.*

(b) *If the predicate is true, then the visited node is a match. Execute everything that needs to be done for the match.*

(3) *Lift pebble m.*

Note that the predicate $\text{step}_i(x, y)$ is implementable for all tree relationships. For instance, the "child" relationship can be easily represented by $\bigvee_{\phi \in \Phi} \text{edge}_\phi(x, y)$. The MSO algorithm uses one local pebble, regardless of the number n of nodes in the path

expression, and it generates the matches in document order. Thus, if it can be applied, i.e., if the path expression allows it, this algorithm is preferable over Algorithms 5.3.2 and 5.3.3. This is only the case when all of the per-node constraints can be expressed using MSO logic. We will analyse whether this is possible in the next paragraph, which deals with value expressions.

We should not fail to mention an assumption that has been made in Algorithms 5.3.3 and 5.3.4, which is that it is possible to walk through the nodes of a document in pre-order, a.k.a. "document order". This poses some difficulty, since in our tree model, there is no easy way to detect which way is "up", as it cannot be deduced from the node label which edge is in the "upward" direction. It can be detected by an MSO predicate, provided that it has a method for detecting which node is the root. Our ptt model does not have a "built in" root identification method, so it is a prerequisite for a "root check" that the root is marked in some way, either by a node label or by a pebble that has been placed upon it at the beginning of the ptt's computation. Fortunately, XML provides us with an easy way to identify the root node without dropping a pebble on it: the node representing the "document entity" is the only "tag" node in the tree that has an empty tag – there is no other way to create an empty tag in XML. This method cannot, however, be used by MSO-restricted ptts. These ptts have no choice but to place a global pebble on the root node at the beginning of the ptt's computation, and to use the $\text{istoward}_{\phi,1}$ predicate to determine the direction of the root. The requirement for this global pebble is not too much of a problem, since dropping a pebble at the root generally comes free of charge for MSO-restricted ptts: they will normally start a simulation of an XSLT transformation or XQuery query with an instance of Algorithm 5.3.3, which will drop a global pebble on the starting node – the root node.

Optimizations for Specific Path Expression Types. Some path expressions can be evaluated without using any pebbles at all. For instance, the following algorithm describes how the `child::` axis may be evaluated without pebbles.

ALGORITHM 5.3.5. *Evaluate a path expression* `child::`t`[`p`]`*, where* t *is a node test and* p *is a predicate, using no pebbles (except for those used in evaluating* t *and* p*).*

(1) *Follow the* firstchild *edge.*
(2) *Test the current node using node test* t *and predicate* p*. If it satisfies the conditions, then it is a match for the path expression. Execute everything that needs to be done for a match.*
(3) *If the current node has a* next *edge, follow it and continue with Step 2.*
(4) *While the current node has a* prev *edge, follow it.*
(5) *Follow the* parent *edge.*

Obviously, a path expression consisting of a sequence of steps using the `child::` axis is equivalent to a sequence of single-step path expressions, so the above algorithm can be used to implement any path expression using only the `child::` axis. Note, however, that path expressions over multiple nodes cannot always be split up into nested evaluations of smaller path expressions without changing the outcome, as this may affect the output ordering or it may create duplicates where there were none before. For instance, the path expression `descendant::foo/descendant::bar` may return a node .../`foo`/.../`foo`/.../`bar` only once, while path expression `descendant::foo` with a nested path expression `descendant::bar` will find the same node twice. If a path expression *only* uses the `child::` axis, then neither the output ordering nor the duplicate filtering of the output is affected if the path expression is split up, so this can be safely done.

The following algorithm can, under certain conditions, be used to evaluate the `descendant::` axis using no pebbles.

ALGORITHM 5.3.6. *Evaluate a path expression* `descendant::`t`[`p`]`, *where t is a node test and p is a predicate, using no pebbles (except for those used in evaluating t and p). Only works under the following conditions:*

(1) *The input document satisfies a DTD.*
(2) *The current node's tag is an element of a known set T. (This can be known if the current node was selected using a path expression that contained a tag test for its last node.)*
(3) *According to the input DTD, for each tag in T, no descendant of a node having a tag in T can ever have a tag in T.*

The algorithm works as follows.

(1) *Execute a pre-order tree walk, starting from the context node. For each visited node:*
   (a) *Check if the node's tag is in T. If it is, we have visited all descendants, and we are back at the context node. Stop the algorithm.*
   (b) *Test the node using node test t and predicate p. If it satisfies the conditions, then it is a match for the path expression. Execute everything that needs to be done for a match.*

The two algorithms presented here are just examples of how certain types of path expressions may be optimized to be evaluated without using pebbles. Depending on the precise combination of input DTD and axis, there are many possibilities to apply pebble-free algorithms. For instance, even the `parent::` axis, which is normally not reversible (i.e., you cannot recover a node when you know only its parent), may be evaluated without using a pebble if the current node's tag is known, and if the DTD

specifies that a node bearing such a tag has a child number that is uniquely determined by the parent's tag.

5.3.3. Value Expressions. Pebble tree transducers can only evaluate a limited subset of the possible value expressions. Recall that such expressions can be complex, and they can refer to nodes and values that were saved earlier. We will now try to give an impression of the capabilities of the ptt to simulate various constructs.

Analysis of an Unbounded Value. A ptt has a built-in finite state machine, using which it can analyze the unbounded values in an XML document, by walking through the nodes that represent the values. The ptt can therefore perform any kind of analysis that can be performed by a (statically known) finite state machine, including but not limited to equality comparisons with a fixed value (e.g. `text = ''foo''`), inequalities (e.g. `x >= 327`), and matching any tag or attribute value in the input tree with a regular expression. Such analyses can be performed by an MSO predicate without requiring any tree walking: MSO predicates can recognize the regular tree languages, which implies that they can calculate the results of finite state machines over any root-to-leaf path in a tree. Unbounded values are on a root-to-leaf path, which means that they can be analyzed.

Comparison of Multiple Unbounded Values. In general, this is not possible. Comparison of an attribute value to another attribute value requires the comparison of two segments of tree, potentially unlimited in size. Pebble tree transducers cannot perform such a comparison, a fact which can be seen as follows. As shown by Milo et al. [**17**], the domain of a ptt can always be described by an MSO predicate. MSO predicates can recognize exactly the regular tree languages, which are the tree languages recognizable by finite tree automata; and it is well known that finite tree automata are not able to compare segments of tree of arbitrarily large sizes. Intuitively, the inability of ptts to compare unbounded values can be understood as follows. To compare two unbounded values, one cannot remember one of the values in the state, so it seems logical that the comparison algorithm must iterate over finite chunks of data from the first value and the second value in parallel, comparing the chunks one-by-one. Imagine that we are at the Nth step in this process, i.e., we are comparing the Nth chunk of data, and we have our reading head at the Nth chunk in the first value. We memorize the contents of the chunk in the state, and then we move to the corresponding group of nodes in the other value. We have to store our position in the first value though; we cannot simply store N in the state because N is unbounded, so we *have* to drop a pebble to mark our location. Now we move to the Nth chunk in the second value, which we presume we can find because we have dropped a pebble there (not being able to store N in the state). We compare the chunk with the remembered state, lift our previous pebble, and drop it again at the $(N+1)$th chunk. But this is impossible, because

the pebble indicating the position in the second value *is not the top pebble*. Therefore, this algorithm cannot possibly work. As we already stated above, MSO logic cannot compare unbounded segments of tree of arbitrarily large sizes, so the addition of MSO logic capabilities to ptts does not provide additional capabilities in this respect.

Multiple Unbounded Values in General. Although comparisons between multiple unbounded values are not possible, there are some cases in which expressions using multiple unbounded values can be evaluated. If a calculation can be split into parts that:

(1) have at most one unbounded input,
(2) can be executed by a finite state machine, and
(3) produce a result from a finite set

then they can be executed by a ptt. For instance, the three-part expression `x > 3 and y < 5` is no problem: the two comparisons can be evaluated individually and yield results from the finite set $\{\text{true}, \text{false}\}$, and the boolean `and` operation that calculates the expression's end result has only finite inputs. However, an expression like $(x^3 \bmod 7)$ (which is expressible in XPath as `((x*x*x)/7-fn:floor((x*x*x)/7))*7`, for positive x) has a result in the finite set $[0, 6]$ (provided that x is integral) but can probably not be executed by a finite state machine; the expression $\frac{x}{3} > y+5$ has multiple unbounded inputs and no intermediate calculations that reduce the unbounded inputs to a finite value, so this expression cannot be executed for the same reasons that comparisons between attributes are not possible.

Now let us consider this in the light of MSO logic. In general, expressions that satisfy the same constraints as specified above, can be evaluated using a single MSO predicate, as follows. Number the component expressions $1, \ldots, m$, where component expression 1 is the topmost expression in the dependency tree. Describe each component expression k using n_k inputs having values in finite sets $I_1, \ldots, I_{n_k}$, respectively, and an output in O_k. Obviously, all of the finite inputs of the component expressions are either predefined in the state or defined by the outputs of some other component expression. For each component expression k, define predicates $\text{hasvalue}_{k,o}$ for $o \in O_k$ as $\bigvee_{i_j \in I_j\ (j \in [1,n_k])} \left(\bigwedge_{j \in [1,n_k]} \left(\text{inputhasvalue}_{j,i_j} \right) \wedge \text{analyze}_{k,o,i_1,\ldots,i_{n_k}} \right)$, where $\text{analyze}_{k,o,i_1,\ldots,i_{n_k}}$ represents the state machine-computable unbounded value analysis for component expression k in the case that the other inputs have values $i_1, \ldots, i_{n_k}$. Furthermore, in this predicate, we use $\text{inputhasvalue}_{j,i_j} \equiv \text{hasvalue}_{c,i_j}$ if input j is defined by component expression c, and otherwise $\text{inputhasvalue}_{j,i_j}$ is true if input j has value i_j, and false otherwise. The predicate $\text{hasvalue}_{1,o}$ now expresses whether the entire expression has value $o \in O_1$. Note that if the expression uses subexpressions that cannot be evaluated using MSO logic, then obviously the expression as a whole cannot be evaluated using a single MSO predicate.

Expressions using Variables and Parameters. In XSLT and XQuery, XPath expressions are evaluated in a context which may define named *variables* (and in XSLT also *parameters*), using `xsl:variable` (and `xsl:with-param`) in XSLT and using the *Let* clause in an XQuery FLWOR expression. These variables contain values from the enclosing calculation. Pebble tree transducers can only model variable values in a "simple" way if they contain values from a statically known finite set (because then their values can be stored in the state), or if they refer to *nodes* (because then they can be identified by a pebble). If a variable refers to a node, the node can be compared to another node (which can be accomplished by comparing pebbles), or the node can be used as the starting point of further operations (looking at attribute values, evaluation of path expressions, etcetera). Note that if a variable is represented by a pebble, then this pebble is often global, but in some cases it may happen to be local, or an optimization can be applied to make it local. We will give an example of this later, in our discussion of XSLT.[2]

With some effort, some variables containing unbounded values can be modeled as well. In general, when a variable's value is defined, it is computed using an XPath expression. Of these, there are only finitely many in an XSL or XQuery transformation. Therefore, the variable's value may be stored by storing the identity of the formula that computed the variable's value (in the state), and the inputs of the computation. When a "formula variable" like this is used, its definition formula can then be expanded so that it is treated as a part of the enclosing expression. It is then, of course, subject to the general limitations on the use of unbounded values that we described in the preceding paragraph. Now, how can we perform storage of the inputs for these "formula variables"? First of all, the context node can be stored using a pebble (whose globality, as with node-valued variables, depends on the circumstances). The values of other variables that serve as input are available because variable scopes are properly nested, i.e., the variables used in another variable's definition have a scope enclosing that of the defined variable.

Expressions using Path Expressions. These kinds of expressions are possible, provided that the path expression in question can be executed by a ptt. The evaluation of the pattern subexpression can simply be nested in the evaluation of the outer pattern expression. Similarly, when we allow the use of MSO logic, the MSO representation of the pattern expression can be used as a subexpression to identify matches; no separate computations are needed to use the pattern expression.

[2]Also, note that whether a variable's pebble is global or local has nothing whatsoever to do with whether the variable itself is global or local. The scope of a "local variable" in XSLT is potentially much larger than what a ptt considers "local": in XSLT "local" means "in nested processing within the same template", a scope that may encompass the dropping of a large number of pebbles, while a ptt views local as "until the next higher-numbered pebble is dropped".

Expressions using Sequences. Expressions using sequences cannot always be modeled. To begin with, it is difficult to represent the sequences themselves: sequences can be unbounded, so it is not possible to, for instance, drop a pebble on each node in the sequence, or remember each value in the sequence in the state. However, this is not the whole story: sequence variables *can* be implemented using the same technique by which we can implement unbounded-value variables: instead of the actual nodes, we store the *expression* that defines the sequence, plus its inputs. Expressions using sequences can then be evaluated by iterating over the elements of the sequence (using the expression that defines the sequence) and evaluating the expression. A set intersection of two sequences can be evaluated by iterating over the first sequence and then checking whether the resulting nodes are also in the second sequence. A set union of two sequences can be evaluated by iterating over all nodes in the tree and by checking membership of both sequences for each nodes. Aggregations over unbounded values contained in nodes are not possible for obvious reasons, but aggregations over bounded values derived from the elements of a sequence can be computed, i.e., one can perform aggregations like "every node in this aggregation satisfies boolean expression *expr*". When we allow MSO logic, these calculations can even be done within a single MSO expression. For instance, the set intersection of two sequences is easily computed if predicates for membership of both sequences are available, simply by using the conjunction of the two predicates.

Discussion: MSO, Path Expressions and Local Pebbles. In Section 5.3.2, we mentioned that path expressions may not be evaluated using MSO logic when one of the per-node constraints could not be translated into an MSO predicate. As we mentioned earlier, Milo et al. **[17]** showed that the domain of a pebble tree transducer can always be described by an MSO predicate. Therefore, if a value expression can be evaluated by a ptt without using MSO, it can *always* be expressed as an MSO predicate as well. This reduces the number of global pebbles required for the evaluation of a path expression to *zero*, for *any* path expression that can be evaluated by a ptt. Furthermore, in contrast with non-MSO implementations, MSO logic can evaluate value expressions starting from other nodes in the tree (be they pebble-marked or found through the use of a path expression), without moving the reading head toward them. The net effect of this is that, when evaluating a subexpression starting from another point in the tree, the MSO implementation does not need to drop a pebble on its previous location to remember it, which not only saves a pebble compared to the non-MSO implementation, it also makes that the next pebble on the stack gains a larger potential to stay local compared to the non-MSO implementation, i.e., if that pebble is referenced in the subcalculation that takes place in the other location. Combining this with the fact that MSO path expressions allow for a global-pebbles-less pattern matching algorithm,

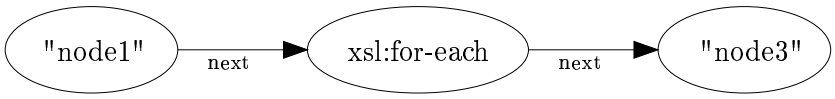
we can only conclude that an implementation using MSO yields much higher numbers of local pebbles.

5.3.4. XSLT. The model by which XSLT generates output is quite similar to that of pebble tree transducers. Where a ptt's instruction trees are segments of output tree that leave certain parts "to be calculated" by placing configuration instructions on the nodes, XSLT's *sequence constructors* are sequences of output XML nodes that leave certain sequences "to be calculated" by including XSLT instructions. In the paragraphs that follow, we will consider implementations for the instructions that we described earlier in Section 5.2.2.

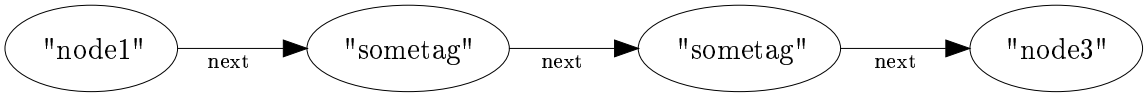
Sequence Constructor Processing. A sequence constructor is a tree segment containing instructions, so the obvious solution is to implement them one-on-one as ptt instructions. However, this is not possible using the XML node sequence representation that we described. For instance, if we have the following sequence constructor:

```
<node1 />
<xsl:for-each select=''*''>
  <sometag />
</xsl:for-each>
<node3 />
```

Then a corresponding instruction would look like this (using the abstracted XML graph format that we discussed earlier):

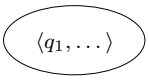


The for-each instruction may produce several nodes, which are inserted in-line into the node sequence at the position of the xsl:for-each node. Our ptt model, however, does not allow us to split a node after it has been created, i.e., we cannot use this instruction to subsequently generate a tree fragment that looks like this:

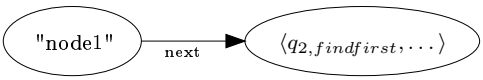


Therefore, using the chosen XML representation, we cannot execute the for-each instruction correctly using a single ptt instruction per sequence constructor. Instead, we need to build the output sequence step by step, starting from the leftmost node and first generating the output of the instructions before continuing with the remainder. While we are processing the instructions in this way, we must of course remember our position in the sequence constructor in the state. Such stepwise generation of output would look somewhat like this:

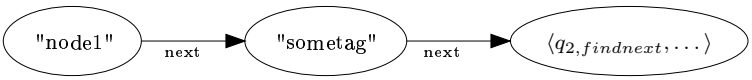
(1) Initial node with label q_1:



(2) Generate the first node, and a configuration which generates the output for the `xsl:for-each` instruction:

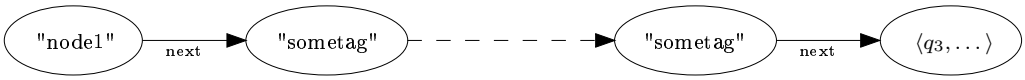


(3) Generate the first output node of the `xsl:for-each` instruction, and a configuration node which calculates the next output node:

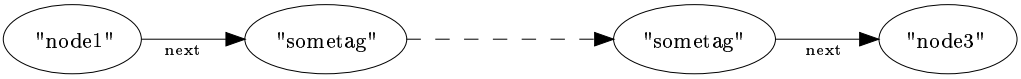


(4) Generate more output for the `xsl:for-each` instruction until the last has been generated:

(5) Switch to state q_3, which generates the third node in the sequence constructor:



(6) After q_3, the final result is:



In this example, the state records the location in this particular sequence constructor, the output nodes of the for-each are generated completely before `node3` is generated, and only then is `node3` generated. Note that, in general, the sequence constructors instantiated by an `xsl:for-each` instruction may itself contain instructions; for instance, in the example above, the fragment `<sometag />` may be replaced by a sequence constructor containing a nested `xsl:for-each` instruction, and possibly more, thus obtaining for instance the following sequence constructor:

```
<node1 />
<xsl:for-each select=''*''>
  <seq1 />
  <xsl:for-each select=''*''>
    <xsl:if test=''@att1 = 3''>
      <seq2 />
```

```
    </xsl:if>
  </xsl:for-each>
  <seq3 />
</xsl:for-each>
<node3 />
```

If we generate the output node sequence of the above fragment, and we are processing the `xsl:if` instruction, we must at the same time be aware of our position in all of the enclosing sequences that are still being constructed. This is no problem: all sequence constructors are finite, so we can store this information in the state. Things become more complicated when the `xsl:apply-templates` instruction is being used; in order to keep things simple for now, we will discuss this instruction last.

Iteration. Iteration in XSLT is done using the `xsl:for-each` instruction. An example:

```
<node1 />
<xsl:for-each select="child::*">
  <seq1 />
  <seq2 />
  <seq3 />
</xsl:for-each>
<node3 />
```

This instruction takes the selection (specified in the example by path expression `child::*`), and for each selected node, it includes the output of the given sequence constructor (in the example these are the nodes with tag `seq[123]`), in-line at the location of the for-each node in the sequence constructor's node sequence. In a pebble tree transducer, this can be implemented by evaluating the selection using Algorithm 5.3.4, inserting the results of the output constructor for every match found. By default the selection is processed in document order, but XSLT provides the possibility to specify a sort order. The pebble tree transducer is not able to simulate sorting, a fact which we will discuss further in Section 5.3.6.

Conditional Output. Conditionals, provided by `xsl:if`, are easy to implement in ptts. The conditions are boolean XPath value expressions, so one simply evaluates them, and depending on the result one either processes the nodes in the conditional's sequence constructor or not.

Aborting the Translation. While its ability to provide the user with a message is of no interest, the `xsl:message` instruction allows for the translation to be aborted. This is also no problem for a ptt: switching to a state for which no rules are defined is guaranteed to lead to an unsuccessful completion of the transformation.

Copying. The `xsl:copy-of` instruction creates deep copies of all of the nodes in a selection, while the `xsl:copy` instruction simply creates a shallow copy of the context node (excluding child nodes and even attributes). Copying nodes to the output is not a problem for ptts, be it deep or shallow copies, and the implementations of these instructions are obvious.

Dealing with Variables and Parameters. The instruction `xsl:variable` binds a value to a named variable, while the instruction `xsl:param` binds a value to a named parameter for nested template invocations. In Section 5.3.3, we already discussed a mechanism for implementing variables so that XPath expressions could use them: node-valued variables can be implemented by dropping a pebble on the node in question; variables whose values are from a finite set can be stored in the state; and for unbounded values, including sequence-typed values, the method of computation can be stored together with the inputs. To these regular types of variables and parameters that are common to both XSLT and XQuery, XSLT adds the possibility to bind a variable to the output of a *sequence constructor*. The value of a sequence constructor is like a node sequence such as would be returned by a path expression, except that it is a sequence of *output* nodes, accompanied by their attributes and descendant nodes. In XSLT 1.0, calculations on such output node sequences (called *result tree fragments* in XSLT 1.0) are limited: one can use them as components of other output node sequences, and one can convert them into strings, and that is it. In XSLT 2.0, node sequences can be manipulated in much more complex ways; effectively, the output of a sequence constructor is a temporary tree, and it can be processed in the same way as an input tree: path expressions can be evaluated on it, and one can even apply templates to it. Clearly, a ptt cannot possibly evaluate such complex expressions. However, the functionality of XSLT 1.0 is implementable by a ptt, insofar as the sequence constructors themselves are implementable. The storage of the output of a sequence constructor in a variable or parameter may be done using the same method that we used with the "formula variables" in Section 5.3.3: one simply stores a sequence constructor identifier (and there are a fixed number of those in an XSLT transformation) to identify the *method* of generating the output node sequence, plus the *inputs* of the method, so that the method may be executed at a later time while yielding the same result. The functionality of including an output node sequence as part of another output node sequence is trivially implemented, as the included output node sequence is just another input of the including output node sequence.

As we announced earlier in our discussion on XPath expressions using variables in general, whether pebbles used for variables are local or global depends completely on the situation, and on the optimizations applied when converting a transformation into a ptt. For instance, consider an XSLT fragment such as the following:

```
<xsl:variable name=''foo'' select='.'' />
<xsl:copy-of select=''$foo/description''/>
<xsl:foreach select=''bar//baz''>
  <foobarbaz>
    <xsl:copy-of select=''$foo/name'' />
    <xsl:copy />
  </foobarbaz>
</xsl:foreach>
```

In this example, the variable `foo` will be implemented using a pebble. The variable's use in the second line does not require this pebble to be global. If the `xsl:foreach` in the third line is implemented using Algorithm 5.3.4, then the use of `foo` in the fifth line takes place when the pebble used by Algorithm 5.3.4 is the top pebble on the stack, which would make the pebble representing `foo` global. However, it is easy to detect that the two pebbles are always dropped on the same node, so it is easy to optimize away the pebble used by Algorithm 5.3.4. If this optimization is applied, the second use of variable `foo` also takes place when `foo`'s pebble is at the top of the stack, which means that the pebble can remain completely local.

Templates. A fact that we have thus far carefully avoided to deal with is the fact that an XSLT transformation actually makes use of template rules. Template rules are the XSLT transformation's equivalent to the ptt's rules. Where a ptt rule has a state and a condition, a template rule's applicability is determined by the current mode and a path expression: the template's "match" attribute specifies which nodes are eligible for processing by the template. The transformation is started by applying the highest-priority eligible template to the root node. Also, from within a sequence constructor, one can use an instruction to apply templates to a node or a selection of nodes: the `xsl:apply-templates` instruction. This instruction works like this: it walks through the selected nodes in order, selects a template for each visited node, and generates output by applying the templates. The difference between the transformation's initial template application to the top-level nodes (the children of the "document entity") and the `xsl:apply-templates` instruction is that with `xsl:apply-templates`, a selection can be given, i.e., the templates are applied to a sequence of nodes. This implies that every matched node must be a match of *two* path expressions: the selection expression from the `xsl:apply-templates` instruction as well as the pattern expression of the template. The selection of the instruction defines the order in which the nodes are subjected to their appropriate templates, so these must be evaluated in an outer loop. We can simply iterate through the selection of the instruction, and we can then use Algorithm 5.3.2 (when not using MSO logic) with a reversed version of each template match expression, or an MSO predicate representing the template match expression, to

check whether the selected node satisfies the templates' match expressions; the first one that actually matches is applied. Note that the previous context node is not involved in the evaluation of the match expressions: the match expressions effectively specify a relationship between a node and *any node* in the tree, i.e., they match if there is *any* string of nodes that matches the pattern and that has the tested node as the match node.

Recursion. The existence of the `xsl:apply-templates` instruction brings with it a most unfortunate downside, namely the possibility of *unbounded recursion*[3]. Depending on the input XML document's Document Type Definition, an XML document may have an unbounded nesting depth. Now consider an XSLT transformation such as the following:

```
<xsl:template match=''foo'' priority=''2''>
  <bar />
</xsl:template>
<xsl:template match=''*'' priority=''1''>
  <xsl:copy>
    <xsl:apply-templates />
  </xsl:copy>
</xsl:template>
```

In this transformation the second template matches every node and simply copies it, applying templates to all of its children. The first template, which has a higher priority, matches only nodes with tag "foo", replacing the node and all its descendants by a single node with tag "bar" in the output. Now, in a ptt simulation of this XSLT transformation using the algorithms that we have discussed, every level of invocation of the second template will cost one pebble, to walk through the matches of the `xsl:apply-templates` instruction.[4] Furthermore, as the location in each nested sequence constructor must be remembered, it will also cost space in the state. For instance, consider this sequence constructor:

```
<node1 />
<xsl:apply-templates />
<node3 />
```

and consider a ptt evaluation of this sequence constructor using the step-by-step process described in the beginning of this (sub)section, at the point where it is evaluating the

[3] A subject which is covered in-depth in [**21**].

[4] In reality, this transformation is so simple that it can be implemented much better than that, as the matches of a pattern (child::*) can all be reversed easily to find the starting node of the pattern. For the sake of the argument, assume that the path expressions are so complex that this is not possible.

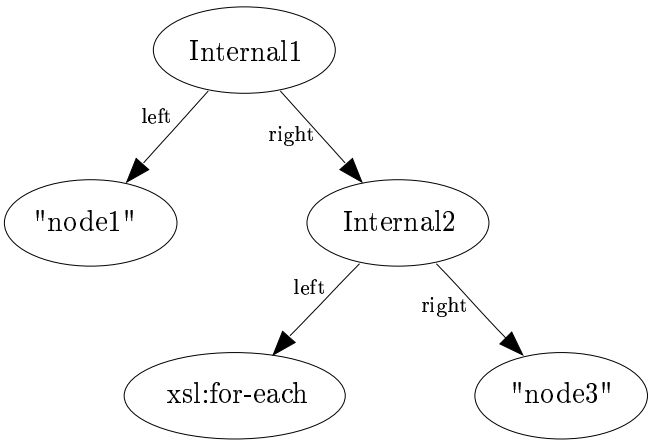
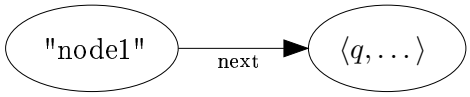


FIGURE 5.3.3. An instruction that encodes XML as a binary tree.

`xsl:apply-templates` instruction. The intermediate output tree will look somewhat like this:



where state q remembers the state of:

(1) The above sequence constructor (i.e., "we are at node 2").
(2) The template that is being evaluated.

This is not a problem, unless the template that is evaluated is *the same template* that contained the above sequence constructor. In that case, the nested template evaluation needs to reuse the part of the state that stores "the location in the above sequence constructor" (item 1), but it is still in use by the enclosing evaluation of the template. Now, there *is* a partial way out of this, but it does have a distinct lack of beauty: it involves *encoding XML differently*. Specifically, it means encoding the XML so that a sequence constructor *can* be instantiated all-at-once. A method for doing this would be to represent node sequences not as a linked list (using *next* edges), but as a *binary tree* (using *left* and *right* edges) whose leaf nodes, traversed in left-to-right order, represent the node sequence. A sequence constructor as in the example would then be evaluated using a single instruction having the structure depicted in Figure 5.3.3. In this structure, the `node3` node is generated immediately instead of after the `xsl:for-each`, and the `xsl:for-each` has its own leaf, where it can add as many nodes in the sequence as it needs to. This tree structure gets rid of the state space nesting, as the location in all nested sequence constructors need not be remembered in the state anymore. Walking through node sequences using a tree structure like this is not a problem, so the possibility to implement the other constructs of XSLT would not be negatively impacted. However, we are still left with a problem, which is the pebble

stack. Generally, the `xsl:apply-templates` invocation in the example will iterate over a set of nodes that are calculated *starting from the enclosing context node*. Therefore, in order to make this iteration possible, this enclosing context node requires a pebble on the stack. By recursively applying this argument for the entire recursion stack, one finds that there is still a pebble stack overflow which will not go away. As our MSO-based XSLT simulation only requires local pebbles (except for pebbles that represent variables – they often need to be global), one possible solution to this problem would be to allow an *unlimited* number of local pebbles, while retaining a limit on the number of global pebbles.

Additional recursion problems crop up when variables (and, for the same reasons, parameters) are involved. Because variables are local to a template, a variable can have an unbounded number of nested definitions. For variables whose values can be stored solely in the state, this is no problem, as we have shown that state space nesting can be avoided. However, for variables that require the storage of nodes (for whatever reason – they may be node-valued variables or they may store a complex non-node value using the techniques we described earlier) pebbles are placed on the stack, and those may not be reclaimed. Furthermore, because of the nature of variables, these pebbles are often required to be global. An unlimited number of local pebbles will then not be sufficient to allow for unbounded recursion in templates using such pebble-requiring variables.

5.3.5. XQuery.

FLWOR Expressions. In contrast with the inherently recursive XSLT "apply-templates" structure, the basic structure of XQuery is provided by FLWOR expressions, which provide no mechanism for recursion. Essentially, a basic XQuery transformation has the same capabilities as a *single template* would have in XSLT: it can iterate over all combinations of multiple sets of nodes, on which it can impose conditions, it has conditionals, etcetera. In essence, an XQuery FLWOR expression like the following:

```
for $f1 in [@f1], ..., $fN in [@fN]
let $l1 := [@l1], ..., $lM := [@lN]
where [expression]
return [...sequence...]
```

(where `@f1..fN` and `@l1..lN` represent path expressions) is equivalent to the following XSLT sequence constructor:

```
<xsl:variable name="start" select="." />
<xsl:for-each select="$start/[@f1]">
  ...
    <xsl:for-each select="$start/[@fN]">
      <xsl:variable name="l1" select="$start/[@l1]" />
```

```
        ...
        <xsl:variable name="lM" select="$start/[@lM]" />
        <xsl:if test="[expression]">
          [...sequence...]
        </xsl:if>
      </xsl:for-each>
    ...
</xsl:for-each>
```

We have already seen that most of the constructs used in this sequence constructor can be implemented using pebble tree transducers; hence the equivalent XQuery FLWOR expression can be implemented by a pebble tree transducer as well.

Recursion. Even though its basic structure does not allow for recursion, XQuery does have a recursive mechanism: it allows for the definition of *functions*, which are sequence constructors that can be called by their name. Functions can be recursive, and if the functions recurse over pattern expressions or sequences, we run into the same infinite-number-of-pebbles problems that occur with XSLT.

5.3.6. Discussion.

Recursion. Both XSLT and XQuery allow for recursion, and therefore for unboundedly nested evaluation of pattern expressions. This is not implementable using our pebble tree transducers, as it requires the availability of an unbounded number of pebbles (albeit local ones). Interestingly, there is a large difference between XSLT and XQuery as to the usability of the language when recursion is omitted. In XSLT, the recursion lies at the core of the processing model. The `xsl:apply-templates` instruction occurs in almost all XSLT transformations, and as documents processed by XSLT are often of types that allow unbounded nesting levels, this implies that most XSLT transformations *cannot be modeled* using the current pebble tree transducer model. On the other hand, the recursion features of XQuery lie at a considerable distance from the core features of the language and are generally not required to write a transformation in XQuery. Therefore, a much larger proportion of XQuery transformations can potentially be modeled using pebble tree transducers.

Joins. The fact that ptts cannot generally handle data values, and especially data value comparisons or *joins*, has been previously discussed by Milo et al. [**17**]. They argue that for type checking purposes, one can make the assumption that the outcomes of the data value comparisons are independent. One can then model an XML transformation that includes data value comparisons using a ptt, but such that the ptt nondeterministically chooses any outcome for every data value comparison. Notwithstanding the fact that such a ptt does not perform the *exact* transformation that it is

intended to model, if the comparison independence assumption is satisfied, its output "type" is the same, and therefore one can still use it to perform a full type check. If the assumption is not satisfied, i.e., if the comparisons are dependent, then the actual output type of the transformation is a subtype of the output type of the ptt-simulated transformation. If the desired output type of the transformation is a supertype of the output type of the ptt-simulated transformation, the ptt-based type check will still succeed, in which case the transformation type-checks as well. However, if the desired output type of the transformation is very strict, the ptt-based type check may fail, a "false negative" type-check.

Complex Calculations. A similar argument can be made for the outcomes of complex calculations based on unbounded values. When these outcomes end up in the output, this does not influence the type checking, as data values are not constrained by DTD document types.[5] When they influence a conditional decision, they can be simulated by a nondeterministic choice for either outcome. Unfortunately, in this case, the outcomes of the complex expressions cannot generally be considered independent. For instance, consider the following fragment of XSLT:

```
<xsl:if test=''x*x < 25''>
  <foo />
</xsl:if>
<xsl:if test=''x*x*x >= 125''>
  <bar />
</xsl:if>
```

If the complex calculations in this example are considered independent, then the possible outcomes of this XSLT fragment are the empty node sequence, `<foo />`, `<bar />` and `<foo /><bar />`. However, in reality, the only possible outcomes are `<foo />` and `<bar />`. Again, the output type of the ptt-simulated transformation is a supertype of the actual output type of the transformation, and this may give rise to "false negatives".

Sorting. In addition to the shortcomings listed above, the pebble tree transducer lacks the capability to iterate over a sequence in a sorted order. Even if it were possible to compare data values, sorting would be impossible because of the stack order of pebbles. To understand this restriction, consider how a pebble tree transducer would perform such a sorted iteration. As a first step, it would have to locate the *first* element in the sort order. It can only do this by iterating over every node in the input sequence, and keeping track of which of the visited nodes was the earliest in the sort order. However, the only way it can do that is by dropping a pebble on the "currently earliest". Even supposing that the ptt were able to compare the next node in the iteration with the node marked "currently earliest", the ptt cannot *replace* the

[5]However, generalized DTDs may specify data value types.

"currently earliest" pebble: it cannot lift the pebble, since it is not at that node; simply walking toward the pebble and lifting it would imply that the ptt would forget the node that it would like to replace the pebble with; and it cannot drop a pebble on the "new currently earliest" node to remember this, since that would be on the stack *on top of* the previous "currently earliest" pebble, preventing the removal of the previous pebble.

Fortunately, in many cases it can be shown that the removal of sorting has no effect on the result of type checking a transformation. Because XML's DTDs only constrain the combination of a node having a certain tag and the order of the tags of its child nodes, it is always safe to swap two nodes bearing the same tag. If every element processed in a sorted operation always yields the same sequence of tags in its output, for instance in this XSLT fragment:

```
<xsl:for-each select=''bambam''>
  <xsl:sort order=''@foo'' />
  <bar />
  <xsl:copy />
  <baz />
</xsl:for-each>
```

then the actual output of the XSLT fragment and the output of this fragment without the sorting always contain the same tag sequences, which implies that the output of the transformation satisfies a certain output DTD if and only if the output of the unsorted version of the same transformation satisfies that same output DTD. This argument unfortunately does not hold for generalized DTDs, as regular tree languages are not closed under the exchange of two subtrees whose roots have the same node label. Still, *some* generalized generalized DTDs may be closed under this operation, for *some* tags, and for those DTDs, for those tags, the argument remains valid.

5.4. XQuery and PTTs: An Example

5.4.1. Introduction. In this section, we will attempt to clarify some of the ideas presented in Section 5.3, by means of an example of how an actual XQuery transformation may be modeled using a ptt. In our example, we will use the following XQuery query X (using unspecified path expressions p_1 through p_5):

```
for $w in /p1
for $x in $w/p2
for $y in fn:intersect($w/p3, $x/p4)
for $z in $y/p5
return
  <foo>
```

```
    {$z/@foobar}
    <baz />
    {$x/@foo}
    <bar>
      {$y/@bar}
    </bar>
  </foo>
```

In the subsections that follow, we will show how one can go about implementing query X using a pebble tree transducer, utilizing exactly two global pebbles, and one local pebble. In the example, we will assume nothing about the path expressions p_1 to p_5 except that they can be expressed using an MSO predicate. Specifically, we will not assume that any of these path expressions can be evaluated using pebble-free algorithms. We will denote the MSO predicate for a path expression p_k ($k \in [1,5]$) by $\text{pathexp}_k(x,y)$.

5.4.2. The PTT. We will define a pebble tree transducer $M = (n, (\Sigma, \Phi), (\Delta, \Gamma), Q, q_0, R)$ with $n = 3$, with $(\Sigma, \Phi) = (\Delta, \Gamma)$ including all node labels and edge labels required to represent XML in the way we defined it in Section 5.3.1, plus those node labels required to represent actual data. If we specify the XML definition further by adding the specification that data consists of ones and zeroes (i.e., that it is binary data), this leaves us with the following definitions:

$$\Sigma = \Delta = \{\text{tag}, \text{attribute}, \text{zero}, \text{one}\}$$
$$\Phi = \Gamma = \{\text{firstchild}, \text{firstattribute}, \text{parent}, \text{next}, \text{prev}, \text{tag}, \text{name}, \text{value}, \text{owner}\}$$

We will not specify Q, q_0 and R in detail; however, we will specify a number of states and instructions that are used.

5.4.3. The Search Algorithm. Query X consists of four nested iterations over the results of path expressions, where each nested iteration is dependent on the current node of some of the enclosing iterations. The ptt algorithm to evaluate the query can be written to follow the query structure quite closely. We will describe it as a single line of computation, which hands off control to an output-generating component when it has found a match.

ALGORITHM 5.4.1. *Evaluate query X with ptt M. This describes the "main line of computation" only.*

(1) *Let all states in Q be marked with a single subscript from* $\{\text{findfirst}, \text{findnext}\}$*, and let this subscript be preserved in subsequent states unless it is explicitly specified that it should be changed. Let the initial state be marked* findfirst.

(2) *Iterate over matches for* `$w` *by evaluating* $/p_1$ *using Algorithm 5.3.4. For each match, execute the following step:*

(3) *Iterate over matches for* `$x` *by evaluating* `$w`$/p_2$*, using Algorithm 5.3.4. Note that during this evaluation, pebble* 1 *represents* `$w`*. For each match, execute the following step:*

(4) *Iterate over matches for* `$y` *by evaluating* `fn:intersect(`w/p_3`, `x/p_4`)` *using an obvious variant of Algorithm 5.3.4. Note that during this evaluation, pebble* 2 *represents* `$x`*, which means that both* `$w` *and* `$x` *are available to the MSO expression used to evaluate* `fn:intersect(`w/p_3`, `x/p_4`)`*. The MSO expression to test whether the head of* M *is at a node in this intersection is*

$$\exists w, x, y : (peb_1(w) \wedge peb_2(x) \wedge pathexp_3(w, y) \wedge pathexp_4(x, y) \wedge head(y)) .$$

For each match, execute the following step:

(5) *Iterate over matches for* `$z` *by evaluating* `$y`$/p_5$ *using Algorithm 5.3.4. Note that during this evaluation, pebble* 3 *represents* `$y`. For each match, execute the following step:

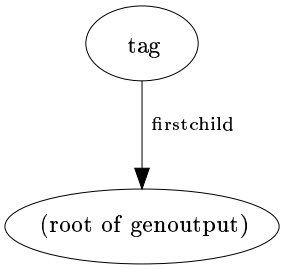
(6) *We now have pebbles 1, 2 and 3 representing* `$w`*,* `$x`*, and* `$y`*, respectively, and the reading head representing* `$z`*. Interrupt the algorithm for generating output by switching to state* $q_{genoutput,f}$*, where* $f \in \{findfirst, findnext\}$ *is the current findfirst/findnext state, and continue the iteration at this point when the state is set to* $q_{outputgenerated,f'}$.

(7) *After all of the algorithm's nested iterations have completed, execute an instruction* $snt(tag)$*, i.e., a single-node tree with label* `tag`*. This removes the algorithm's final configuration from the intermediate output tree and replaces it by the node label* tag*. (The reason for this will become clear in the next section, when we explain how output is generated.)*

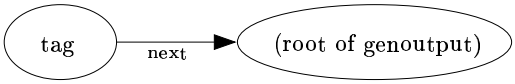
In this algorithm, pebble 1 is used in Step 4 while pebble 2 is also on the stack. Therefore, pebble 1 is global. On the other hand, pebble 2 is only used in that same step, when it is at the top of the stack, so it is a local pebble. While pebble 3 is only used while it is at the top of the stack, it is nevertheless global, as the nth pebble is always global. However, it should be noted that if the query implementation would have required more pebbles, nothing would have kept pebble 3 from being local as well.

5.4.4. Generating Output: Main Structure. Algorithm 5.4.1 signals when everything is set up for generating an output item by switching the state to $q_{\text{genoutput},f}$, where $f \in \{\text{findfirst}, \text{findnext}\}$ represents whether a segment of output has already been generated. When this state is reached, output should be generated using instruction ι_{first} when $f = \text{findfirst}$, and using instruction ι_{next} when $f = \text{findnext}$. Both of these

instructions are based on a single instruction tree $\iota_{\text{genoutput}}$, shown in Figure 5.4.1. (Note: this graph, and the following ones does *not* use the abstracted XML graph representation that we described in Section 5.3.1, except that the "upward" edges are not displayed.) Instruction ι_{first} then looks like this, generating a *tag* node that includes $\iota_{\text{genoutput}}$ as its first child:



Instruction ι_{next} has the same structure, but it includes $\iota_{\text{genoutput}}$ as the next sibling of the *tag* node:



These instructions spawn several lines of computation. The main line of computation described in Algorithm 5.4.1 continues in the root of the output generated by $\iota_{\text{genoutput}}$, while the configurations in the other nodes independently compute the output values specified in the subscripts of their states. When the state is marked "findfirst",

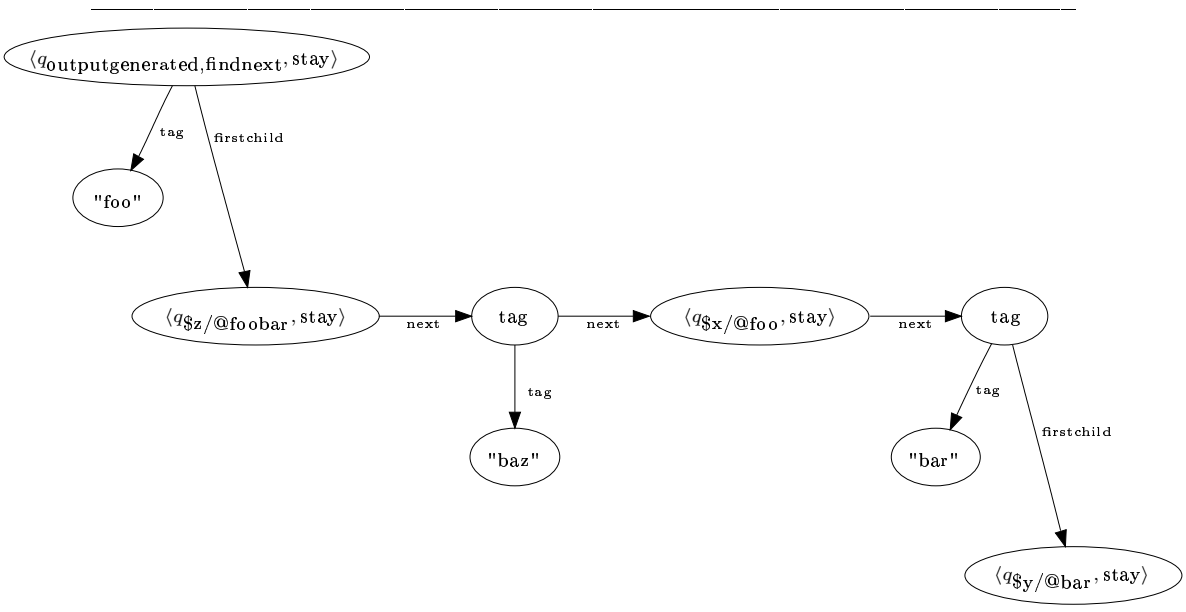
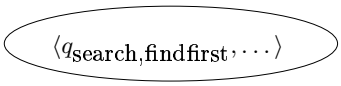


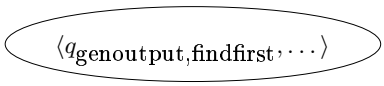
FIGURE 5.4.1. The graph for instruction $\iota_{\text{genoutput}}$.

this indicates that the main line of computation takes place at the "document entity" node at the top of the XML output, and that the first output segment should be attached to the node in which the computation takes place using an edge labeled *firstchild*. After ι_{first} has been executed, the main line of computation is continued with the state marked as *findnext*, to indicate that a fragment of output has already been generated, that the main line of computation is taking place in the node at the very top of the last generated segment of output, and that a subsequent output segment should be added to this node using an edge labeled *nextchild*. The following graphs provide a step-by-step illustration of the way the output is generated.

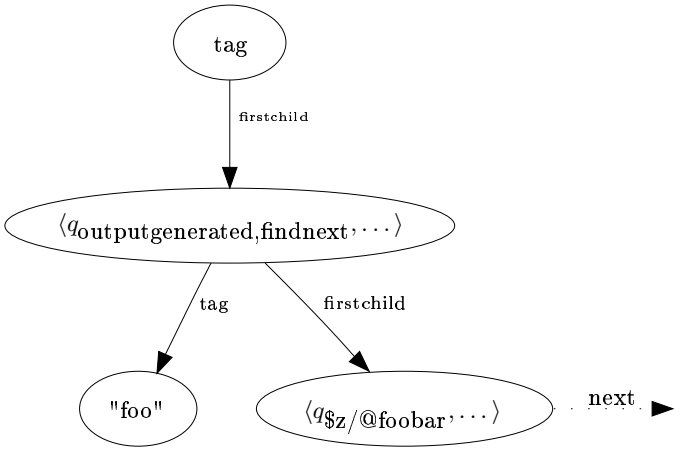
Step 1: The initial situation. In this example, any state with subscript *search* is an abstraction for any state that concerns searching for a result, i.e., the implementation of Algorithm 5.4.1.



Step 2: Algorithm 5.4.1 has found a result, so it switches the state to *genoutput*.

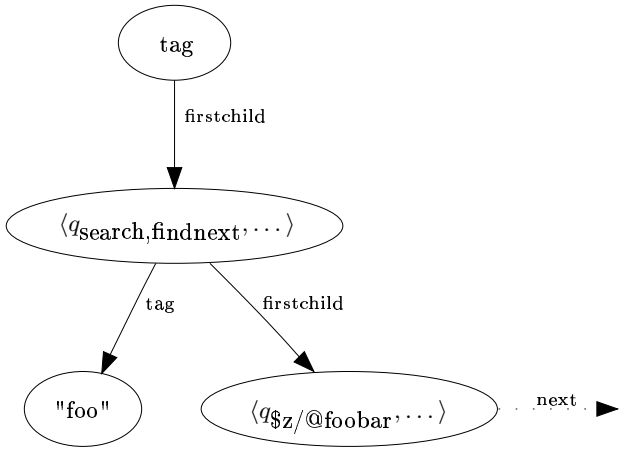


Step 3: The state is labeled *findfirst*, so the instruction ι_{first} is executed. Note that the root node is a bare "tag" node, with no actual tag value. This is the document entity, as described in Section 5.3.1. Also note that the entire structure of the first result is already generated (although much of it has been left out in this graph). The "tag" label of the root of the generated result has not been generated yet, as this node harbours the configuration that continues the "main" algorithm of the query. The state of the continuing configuration is $q_{\text{outputgenerated,findnext}}$. This triggers a continuation of Algorithm 5.4.1. By marking the state as *findnext*, the ptt remembers that the document entity has already been generated.

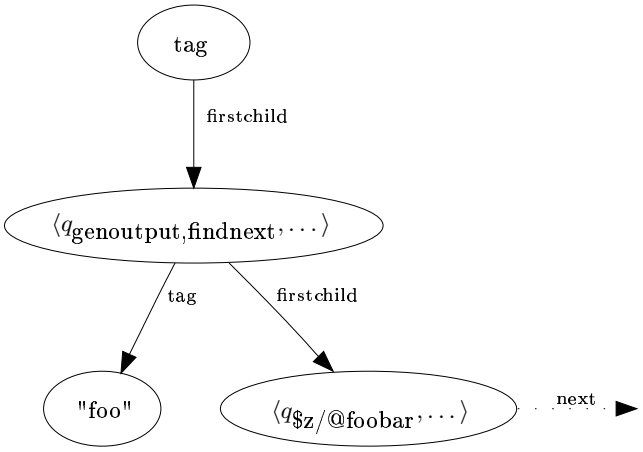


Note that the fact that we can immediately generate the *entire* result structure, including the tag of the result's topmost node, and then forget about it, is a convenient side effect of our choice to encode even a node's tag as a data value. Had we chosen to encode tags as node labels, then we would have had to remember the tag in the state until the next result were found, or until the algorithm would complete! If the node label is static, such as in this example ("foo"), this would not pose any real problem, as the tag value that is to be generated can simply be stored in the state. However, if the node label is *dynamic*, e.g., if it were copied from the input, then the search algorithm must remember an unbounded value. As the search algorithm must be able to lift all pebbles off the stack, it cannot properly do this. In contrast, our current XML graph representation has no problems with dynamic node labels.

Step 4: Algorithm 5.4.1 picks up, searching for a next result.

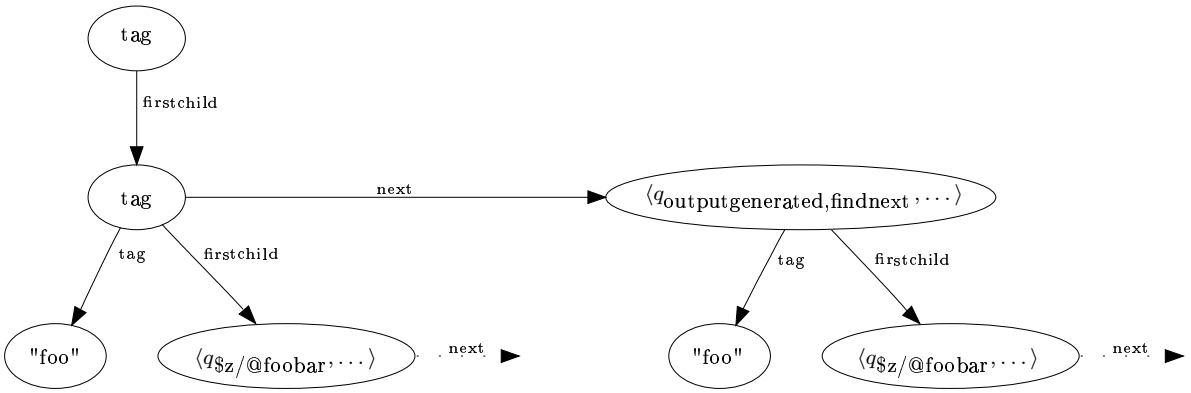


Step 5: The algorithm finds another result, so it switches to state *genoutput*.



Step 6: As the state is now labeled *findnext*, the instruction ι_{next} is executed. This generates the label of the root node of the first result, and the complete structure of

the second result. Like in Step 3, the new result's root node contains the configuration that continues the search for more results.



5.4.5. Generating Output for Individual Output Nodes. Having covered the main structure of how output is generated, we will now consider how the parallel computations that generate the actual output can achieve their goals. As the parallel computations are all very similar, we will look at only one: the computation for the node labeled $\left\langle q_{\$x/@foo}, \text{stay} \right\rangle$, that was shown in Figure 5.4.1. Remember that, at the beginning of these lines of computation, the pebble stack contains three pebbles marking `$w`, `$x` and `$y`, and the reading head marks `$z`. Therefore, the simplest implementation for generating output for `$x/@foo` is to walk to the node marked by pebble 2, to traverse its outgoing edge labeled *firstattribute*, and to iterate over all attributes until an attribute named `foo` is found, and then to copy its value to the output. However, this naive implementation would make pebble 2 a global pebble! The solution is very simple: before walking to the node marked by pebble 2, pebble 3 must be lifted. This poses no problem, as the value of `$y` is not needed in this line of computation, and the lines of computation for each calculated output node and also the main line of computation have independent pebble stacks.

5.5. XQuery Pebble Requirements

5.5.1. Introduction. The example presented in Section 5.4 required three pebbles, of which two were global. In this section, we present a more general method for determining the number of global pebbles required to evaluate a FLWOR clause in an XQuery transformation, for a selected subset of FLWOR clauses. In addition, we will provide an optimization strategy for evaluating unordered-mode FLWOR clauses.

5.5.2. Required Global Pebbles. Consider an XQuery transformation of the following form:

```
for x1 in //*, x2 in //*, ..., xn in //*
where φ1 and φ2 and ... and φm
return ...
```

where each ϕ_k ($k \in [1, m]$) is a boolean XPath expression that references variables with numbers $V_k \subseteq \{1, 2, \ldots, n\}$, that cannot be rewritten as one or more smaller expressions that each use only a subset of these variables. A ptt normally processes the above transformation by using $n-1$ pebbles and the reading head to represent variables x_1 to x_{n-1} and x_n, respectively, and by using the pebbles to perform n nested iterations which each iterate over all nodes in document order. Now consider expression ϕ_k, and let $v = \max(V_k)$. Then ϕ_k can only be evaluated by the ptt while the current iteration nesting level is in $[v, n]$, as these are the only times when all of the inputs to ϕ_k have been assigned values. Considering that when the iteration nesting level is $l \in [1, n]$, pebbles $1, \ldots, l-1$ represent variables $x_1, \ldots, x_{l-1}$ and the reading head represents variable x_l, we can describe the effects of evaluating ϕ_k at a given iteration nesting level l on the globalness of pebbles as follows:

- If $l \geq v + 2$, then the pebble stack contains at least $v + 1$ pebbles, which means that none of the variables referenced by ϕ_k are at the top of the stack. Therefore, all pebbles with numbers in V_k are forced to be global.
- If $l = v+1$, then the pebble stack contains v pebbles, which means that of the variables referenced by ϕ_k, only x_v is represented by a pebble that is at the top of the stack. The remaining pebble numbers in V_k are forced to be global.
- If $l = v$, then the pebble stack contains $v-1$ pebbles; variable x_v is represented by the reading head and variable x_{v-1} is represented by pebble $v-1$ at the top of the stack. The remaining pebble numbers in V_k are forced to be global.

Clearly, using this evaluation algorithm it is optimal to evaluate ϕ_k during iteration nesting level $l = v$. The set of pebbles that are required to be global for the evaluation of the entire XQuery transformation can then be described as

$$\{n\} \cup \bigcup_{k \in [1,m]} \left(V_k - \{\max(V_k), \max(V_k) - 1\}\right)$$

where $\{n\}$ is included because the highest-numbered pebble is always global. Note that if, for an expression ϕ_k, $V_k = \{v, v-1\}$ or $V_k = \{v\}$, i.e., the expression only references a variable and its direct neighbour in the nesting order, the expression does not cause any global pebbles to be created. If all expressions are of this kind, no global pebbles are needed (except of course pebble n).

It should be noted that in general, any query of the form:

```
for x1 in pathexp1,
    x2 in pathexp2,
```

```
        ...
        x_n in pathexp_n
    where ...
```

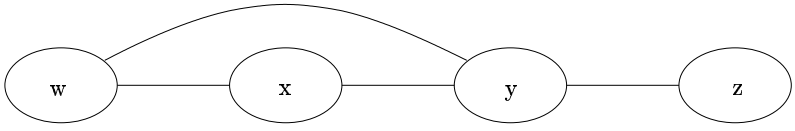
can be rewritten to the above form, by replacing all of the path expressions by `//*`, and by placing an equivalent constraint in the `where` clause. In general, a variable x_k originally defined by path expression `pathexp`k can be constrained in the where clause using the expression `fn:exists(pathexp`k`[.=`x_k`])`. This extends the above result to a considerably larger set of queries.

5.5.3. Unordered Mode Optimization. XQuery transformations can be evaluated in either *ordered* or *unordered* mode. In ordered mode, the variables in a `for` expression such as the one described above are required to iterate over their values in nested document order, i.e., the outer loop iterates over all nodes for the first variable in document order, a nested loop iterates over all nodes for the second variable, etc. In unordered mode, however, the variables may be processed in any order. So, for unordered-mode FLWOR clauses, we can attempt to reorder the variables to reduce the number of global pebbles. We will now present an ordering strategy that often succeeds at significantly reducing the number of global pebbles. Note that this alters the output of a ptt, so it may not always be a suitable optimization for typechecking purposes. In many cases, however, this does not matter; the same arguments apply that we used in Section 5.3.6, in the discussion on sorting.

For an XQuery transformation of the form presented in Section 5.5.2, let the graph g be defined as:

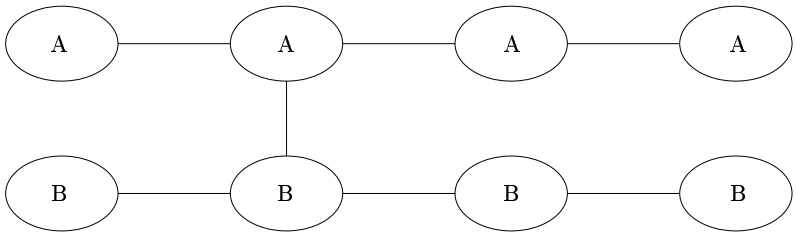
$$\begin{aligned} V_g &= \{x_i \mid i \in [1,n]\} \\ E_g &= \{(x_i, x_j) \mid \text{there is a } k \in [1,m] \text{ with } \{i,j\} \subseteq V_k\} \end{aligned}$$

with arbitrary node and edge labels. This graph contains the variables of the transformation as nodes, where two variables are connected if there exists a `where` clause constraint to which they are both input. For instance, the graph for the XQuery transformation that we discussed in Section 5.4 (restructured so that it is in the form presented in Section 5.5.2, and ignoring the connection between `w` and the root node) looks like this:



Now consider an arbitrary XQuery transformation of the form presented in Section 5.5.2, and its graph g. Observe that, for any path $x_i \ldots x_j$ through such a graph g, if we evaluate the variables in consecutive nested iterations, in the same order in which the

variables occur in the path, then the variables along that path can be represented by local pebbles. That is, of course, assuming that all other input variables are available by other means, i.e., global pebbles, and that the pebbles are not forced to be global for other reasons. We can generalize this observation into a method by which we can optimize the number of local pebbles: we simply carve up the graph into such "paths" as much as possible, and then all of the nodes in the paths can be represented by local pebbles. For this to work, the paths need to be selected so that they do not introduce the need for global pebbles *in each other's* evaluations. For instance, if g has a structure like the following, it is very tempting to select the nodes labeled "A" as a path, and the nodes labeled "B" as a second path.



However, this fails to work: the individual paths can be evaluated using only local pebbles under the assumption that the *other* inputs to the constraints on the nodes on the path are available as global pebbles. That means that either the second node from the top row, or the second node from the bottom row must be represented by a global pebble. In general, these are the conditions that a set of paths must fulfill so that the variables on *every* path in the set can be evaluated using only local pebbles:

- The paths do not overlap.
- There are no direct connections between the paths (because otherwise, at least one of the sides of the connection become global).
- The nodes in the paths are not connected to each other in ways other than by the edges in the path.

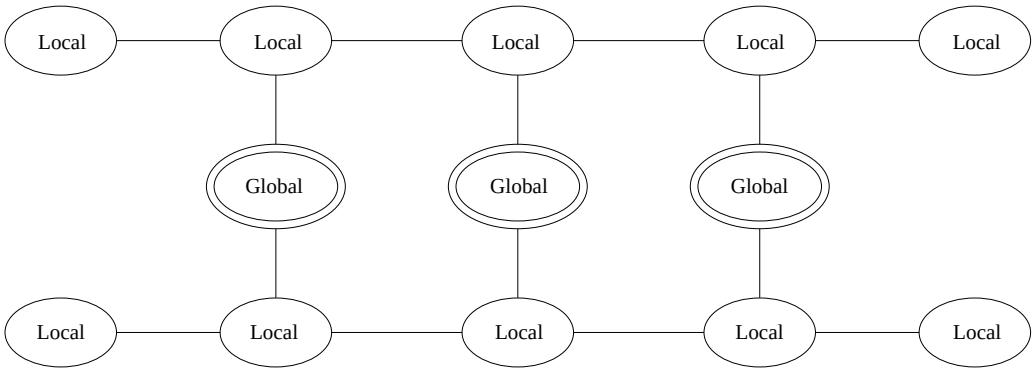
If we have selected a set of paths that fulfill these conditions, containing $k \geq 2$ nodes, then we can evaluate the entire XQuery transformation using $k-2$ local pebbles and $n-k+1$ global pebbles, by nesting the iterations as follows: the outer $n-k$ iterations iterate over all possible values for the variables not on the paths, and then the inner k iterations iterate, for each path, over the possible values for the variables on the paths, nested in the order in which they occur in the paths. The fact that the paths do not refer to each other guarantees that they can be evaluated in any nesting order, and the fact that the variables that are not on the paths are placed in the outer iterations ensures that they are available to the nested iterations that evaluate the paths. Of the inner k iterations, the deepest-nested iteration requires no pebble as it uses only the reading head, while the next deepest nested iteration uses the highest-numbered

pebble which is global; the remaining pebbles are local, yielding a total of $k-2$ local pebbles. The remaining $(n-1)-(k-2) = n-k+1$ pebbles are global.

Applying this idea to the query from Section 5.4 (for which the graph g was displayed earlier in this section), we can see that we can choose either path `wyz` or path `xyz` for "local" evaluation, making either the pebble for `x` or the pebble for `w` global. Note that in this example, reordering does not reduce the number of global pebbles: the evaluation still requires two global pebbles, as the highest-numbered pebble will be global as well.

Formally, we can express the selection of paths as follows. We select a largest set $V_{\text{local}} \subseteq V_g$ so that the V_{local}-induced subgraph of g contains no cycles, and so that the V_{local}-induced subgraph of g contains no nodes of a degree higher than two. The V_{local}-induced subgraph of g consists of a set of disconnected components that are monadic trees. These components represent the selected "paths" through g. The nodes in $V_g - V_{\text{local}}$ represent the variables that must be represented by global pebbles.

While the optimum that we achieved with this optimization strategy may not always be the absolute optimum, i.e., depending on the path expressions better algorithms may be possible, it places a reasonable upper bound on the number of global pebbles required. In some cases, using this strategy yields exceptionally good results, for instance when g looks like this:



In this example, only the central three variables need to be represented by global pebbles: after representing these variables by global pebbles, we can evaluate the variables in the path along the top and the path along the bottom using local pebbles. Note that one of the variables marked "local" in this graph will be represented by a global pebble after all, despite its inclusion in the selected paths: the highest-numbered pebble is always global. This example therefore requires a total of four global pebbles.

CHAPTER 6

Conclusions

6.1. Introduction

In this chapter, we will summarize the results shown in this thesis, and we will reiterate some of the thornier issues that we have run into. We will start by discussing our extensions to the pebble tree transducer model in relation to type checking. In the section that follows, we discuss our findings regarding the relationship between XML transformation languages and pebble tree transducers. We will wrap up our conclusions with a short discussion of some remaining issues.

6.2. Type Checking

As we discussed in the introduction, type checking of an n-pebble tree transducer has until now been shown to be feasible using an algorithm whose time complexity is $(n+2)$-fold exponential (i.e., a tower of $n+2$ exponentials). In the present thesis, we have introduced the concept of local pebbles, which leads to an algorithm that improves upon this time complexity. We have shown in Chapter 4 that a pebble tree transducer with k global pebbles and *any* number of local pebbles can be decomposed into $k+1$ twtts, yielding a type checking algorithm with a $(k+2)$-fold exponential time complexity. We have also shown that in real situations, like in the example of Section 5.4, it is possible to execute transformations that require n pebbles using algorithms that make at least some of the pebbles local, like the algorithms described in Section 5.3.2. This means that this improvement is not purely theoretical: it reduces complexity in real situations.

In addition to the concept of local pebbles, we have introduced a ptt extension which allows ptts to perform MSO tests. We have seen in Sections 5.3.2 and 5.3.3 that the use of MSO tests can lead to a decrease in the required number of global pebbles; and each global pebble that can be avoided leads to a reduction of the time complexity of the type checking algorithm by one level of exponentiation. The MSO extensions also have a cost: we observed in Section 1.3.3 that the type checking algorithm has a time complexity that includes one additional level of exponentiation. Therefore, if we compare an algorithm implementation using a regular ptt and an MSO-extended ptt, the MSO-extended ptt is at an advantage if it is able to execute the algorithm using two less global pebbles than the regular ptt.

In conclusion, we have introduced concepts which yield both theoretical and real-world improvements on the time complexity of type checking for a significant subset of the transformations that can be expressed using pebble tree transducers. The use of local pebbles is already able to provide an improvement by itself in many situations, but the potential for improvement becomes even larger when pebble tree transducers with MSO tests are used.

6.3. PTTs and XML Transformation Languages

In Chapter 5, we have looked at the semantics of XPath expressions, XSLT and XQuery, and we have described a number of techniques for implementing these semantics using pebble tree transducers, both using MSO extensions and without using MSO extensions. We have shown that a surprisingly large subset of the constructs of XSLT and XQuery are, in fact, implementable using pebble tree transducers. In particular, the possibilities for storing various variable values have turned out to be larger than we expected: we have found techniques to store node sequences, result tree fragments, and even some unbounded values in variables. On the other hand, we have also found that there are some fundamental limitations. One of these limitations, the inability to perform data value joins on unbounded values, was already discussed by Milo et al. in [**17**]. In their paper, they argue that if join conditions are *independent*, then pebble tree transducers can still be used for type checking, as the output type then remains the same. We have found that sorting, which is closely related to joins, is also not implementable using pebble tree transducers. However, we have also provided an argumentation that under specific conditions, a ptt can be used for type checking transformations that include a sorting operation.

In [**17**], Milo et al. claimed that XSLT cannot be fully implemented using pebble tree transducers. As a cause of this, they only explicitly mentioned the inability of pebble tree transducers to perform data value joins. However, next to this rather obvious limitation, we have found an unexpected additional limitation, which is that *unbounded recursion* is not possible. This limitation is a direct result of the fact that a pebble tree transducer has a bounded number of pebbles. Because of the highly recursive nature of XSLT, this implies that of the XSLT transformations, only those transformations can be simulated for which it can be statically shown that they have only bounded recursion. Unfortunately, this is a very small subset of typical XSLT transformations. So, notwithstanding the fact that most individual XSLT constructs can be implemented using pebble tree transducers, the implementation of the whole of an XSLT transformation is often problematic.

6.4. Discussion

It is important to recognize the fact that, in order to achieve good type checking time complexities using the techniques that we describe, the construction of the pebble tree transducers needs to be done with care. Various techniques for implementing XSLT- and XQuery-expressible transformations were presented in Chapter 5, each with different effects on the number of global pebbles. A careful choice of implementation techniques makes all the difference. Optimization techniques, like the one we presented in Section 5.5.3, can yield dramatic improvements compared to naive implementations. Unfortunately, we have no algorithm to generate *the* optimal implementation (i.e., the implementation with the least number of global pebbles) for a given XSLT or XQuery transformation. We explicitly do not claim that the techniques presented in Chapter 5 are optimal, and as the implementations are bound only by the semantics of the modeled XSLT and XQuery transformations, in many situations more optimal implementations may be possible.

The problems in modeling XSLT transformations are most unfortunate. As we have explained, the cause of these problems is found in the fact that a pebble tree transducer has only a bounded number of pebbles, while XSLT transformations often require unbounded recursion. There is a relatively simple solution to this problem, which is to allow an unbounded number of *local* pebbles. This solution, and its implications, are beyond the scope of this thesis; we have investigated it in detail, and a paper presenting the results is currently under review for publication [**12**].

Bibliography

[1] "RELAX NG." http://www.oasis-open.org/committees/relax-ng/.

[2] "Information processing – Text and office systems – Standard Generalized Markup Language (SGML)," tech. rep., International Organisation for Standardization (ISO), 1986.

[3] A. Aho and J. Ullman, "Translations on a context free grammar," *Inform. Control*, vol. 19, pp. 439–475, 1971.

[4] M. Blum and C. Hewitt, "Automata on a 2-dimensional tape," in *Proceedings of the 8th IEEE Symposium on Switching and Automata Theory*, pp. 155–160, 1967.

[5] M. Bojanczyk, M. Samuelides, T. Schwentick, and L. Segoufin, "Expressive power of pebble automata," in *International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 157–168, 2006.

[6] A. Brüggemann-Klein and D. Wood, "Regular tree languages over non-ranked alphabets." Unpublished manuscript, 1998.

[7] J. Clark, "XSL transformations (XSLT) version 1.0," W3C recommendation, World Wide Web Consortium (W3C), Nov. 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[8] J. Engelfriet, "Tree automata and tree grammars." Lecture Notes, DAIMI FN-10, Aarhus University, 1975.

[9] J. Engelfriet, 2006. Personal communication.

[10] J. Engelfriet, "The complexity of typechecking tree-walking tree transducers." Unpublished manuscript, 2006.

[11] J. Engelfriet and H. J. Hoogeboom, "Tree-walking pebble automata," in *Jewels are Forever*, pp. 72–83, 1999.

[12] J. Engelfriet, H. J. Hoogeboom, and B. Samwel, "XML transformation by tree-walking transducers with invisible pebbles." Unpublished manuscript, 2006.

[13] J. Engelfriet and S. Maneth, "A comparison of pebble tree transducers with macro tree transducers," *Acta Informatica*, vol. 39, pp. 613–698, 2003.

[14] D. C. Fallside, "XML schema part 0: Primer," W3C recommendation, World Wide Web Consortium (W3C), May 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[15] F. Gécseg and M. Steinby, *Tree Automata*. Budapest: Akadémiai Kiadó, 1984.

[16] M. Kay, "XSL transformations (XSLT) version 2.0," W3C proposed recommendation, World Wide Web Consortium (W3C), Nov. 2006. http://www.w3.org/TR/2006/PR-xslt20-20061121/.

[17] T. Milo, D. Suciu, and V. Vianu, "Typechecking for XML transformers," *J. Comput. Syst. Sci.*, vol. 66, no. 1, pp. 66–97, 2003. (Originally appeared in the Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2000).

[18] M. Murata, D. Lee, M. Mani, and K. Kawaguchi, "Taxonomy of XML schema languages using formal language theory," *ACM Trans. Internet Techn.*, vol. 5, no. 4, pp. 660–704, 2005.

[19] F. Neven and T. Schwentick, "Automata- and logic-based pattern languages for tree-structured data," in *Semantics in Databases 2001, LNCS 2582*, pp. 160–178.

[20] J. Robie, A. Berglund, D. Chamberlin, M. Kay, M. F. Fernández, J. Siméon, and S. Boag, "XML path language (XPath) 2.0," W3C proposed recommendation, World Wide Web Consortium (W3C), Nov. 2006. http://www.w3.org/TR/2006/PR-xpath20-20061121/.

[21] B. Samwel, "Pebble scope and the power of pebble tree transducers," Master's thesis, LIACS, Leiden University, 2007.

[22] J. Siméon, J. Robie, M. F. Fernández, S. Boag, D. Chamberlin, and D. Florescu, "XQuery 1.0: An XML query language," W3C proposed recommendation, World Wide Web Consortium (W3C), Nov. 2006. http://www.w3.org/TR/2006/PR-xquery-20061121/.

[23] C. M. Sperberg-McQueen, T. Bray, and J. Paoli, "XML 1.0," W3C recommendation, World Wide Web Consortium (W3C), Feb. 1998. http://www.w3.org/TR/1998/REC-xml-19980210.