Fast: A Transducer-Based Language for Tree Manipulation

LORIS D'ANTONI, University of Pennsylvania
MARGUS VEANES, BENJAMIN LIVSHITS, and DAVID MOLNAR, Microsoft Research

Tree automata and transducers are used in a wide range of applications in software engineering. While these formalisms are of immense practical use, they can only model finite alphabets. To overcome this problem we augment tree automata and transducers with symbolic alphabets represented as parametric theories. Admitting infinite alphabets makes these models more general and succinct than their classic counterparts. Despite this, we show how the main operations, such as composition and language equivalence, remain computable given a decision procedure for the alphabet theory. We introduce a high-level language called Fast that acts as a front-end for the preceding formalisms.

Categories and Subject Descriptors: F.1.1 [**Theory of Computation**]: Models of Computation, Automata

General Terms: Algorithms, Languages, Verification

Additional Key Words and Phrases: Symbolic tree transducers, FAST

ACM Reference Format:
Loris D'Antoni, Margus Veanes, Benjamin Livshits, and David Molnar. 2015. Fast: A transducer-based language for tree manipulation. ACM Trans. Program. Lang. Syst. 38, 1, Article 1 (October 2015), 32 pages.
DOI: http://dx.doi.org/10.1145/2791292

1. INTRODUCTION

This article introduces Functional Abstraction of Symbolic Transducers (Fast), a new language for analyzing and modeling programs that manipulate trees over potentially infinite domains. Fast builds on top of satisfiability modulo theory solvers, tree automata, and tree transducers. Tree automata are used in a variety of applications in software engineering, from analysis of XML programs [Hosoya and Pierce 2003] to language type checking [Seidl 1994b]. Tree transducers extend tree automata to model functions over trees, and appear in fields such as natural language processing [Maletti et al. 2009; Purtee and Schubert 2012; May and Knight 2008] and XML transformations [Maneth et al. 2005]. While these formalisms are of immense practical use, they suffer from a major drawback: in the most common forms they can only handle finite alphabets. Moreover, in practice existing models do not scale well even for finite but large alphabets.

In order to overcome this limitation, *Symbolic Tree Automata* (STAs) and *Symbolic Tree Transducers* (STTs) extend these classic objects by allowing transitions to be labeled with formulas in a specified theory. While the concept is straightforward,

Loris D'Antoni did this work as part of an internship at Microsoft Research, and he is supported by NSF Expeditions in Computing award CCF 1138996.

Authors' addresses: L. D'Antoni, University of Pennsylvania 3330 Walnut St. Philadelphia, PA, 19104; email: lorisdan@seas.upenn.edu; M. Veanes, B. Livshits, and D. Molnar, Microsoft Research, One Microsoft Way, Redmond, WA; emails: margus@microsoft.com, livshits@microsoft.com, dmolnar@microsoft.com.

DOI: http://dx.doi.org/10.1145/2791292

	Composition	Equivalence	Pre-image
Augmented reality	✓		✓
HTML sanitization	✓		✓
Deforestation	✓		
Program analysis	✓	✓	✓
CSS analysis	✓	✓	✓

Fig. 1. Representative applications of Fast discussed in Section 5. For each application we show which analyses of Fast are needed.

traditional algorithms for constructing composition, deciding equivalence, and other properties of finite automata and transducers *do not* immediately generalize. A notable example appears in D'Antoni and Veanes [2013a] where it is shown that while in the classic case allowing finite automata transitions to read subsequent inputs does not add expressiveness, in the symbolic case this extension makes most problems, such as checking equivalence, undecidable. Symbolic tree automata still enjoy the closure and decidability properties of classic tree automata [Veanes and Bjørner 2012] under the assumption that the alphabet theory forms a Boolean algebra (i.e., closed under Boolean operations) and it is decidable. In particular, STAs can be minimized and are closed under complement, and intersection, and it is therefore decidable to check whether two STAs are equivalent.

Taking a step further, tree transducers model transformations from trees to trees. A STT traverses the input tree in a top-down fashion, processes one node at a time, and produces an output tree. This simple model can capture several scenarios; however, in most useful cases it is not closed under sequential composition [Fülöp and Vogler 2014]. In the case of finite alphabets this problem is solved by augmenting the transducer's rules with regular look-ahead [Engelfriet 1977], which is the capability of checking whether the subtrees of each processed node belong to some regular tree languages. We extend STTs in a similar way, and introduce *Symbolic Tree Transducers with Regular look-ahead* (STTRs). The main theoretical result of this article is a new composition algorithm for STTRs together with a proof of its correctness. Similarly to the classic case, we show that two STTRs A and B can be composed into a single STTR $A \circ B$ if either A is single-valued (for every input produces at most one output), or B is linear (traverses each node in the tree at most once). Remarkably, the algorithm works modulo any decidable alphabet theory that is an effective Boolean algebra.

We introduce the language Fast as a front-end for STAs and STTRs. Fast is a functional language that integrates symbolic automata and transducers with Z3 [De Moura and Bjørner 2008], a state-of-the-art solver able to support complex theories that range from data types to nonlinear real arithmetic. We use Fast to model several real world scenarios and analysis problems: we demonstrate applications to HTML sanitization, interference checking of augmented reality applications submitted to an app store, deforestation in functional language compilation, and analysis of functional programs over trees. We also sketch how Fast can capture simple CSS analysis tasks. All such problems require the use of symbolic alphabets. Figure 1 summarizes our applications and the analyses enabling each one. In Section 7 we further contrast Fast with previous Domain-Specific Languages (DSLs) for tree manipulation.

Contributions summary:

(1) a *theory of symbolic tree transducers with regular look-ahead* (STTR), that nontrivially extends the classic theory of tree transducers (Section 3);
(2) a new algorithm for composing STTRs together with a proof of correctness (Section 4);

(3) Fast, a DSL for tree manipulations founded on the theory of STTRs (Section 3); and
(4) five concrete applications of Fast showing how composition of STTR can be beneficial in practical settings (Section 5).

2. MOTIVATING EXAMPLE

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

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

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

We now describe the sanitization functions. The transformation *remScript* (lines 14–18) takes an input tree t of type *HtmlE* and produces an output tree of type *HtmlE*: (1) if $t = node(x_1, x_2, x_3)$ and its *tag* is different from `"script"`, *remScript* outputs a copy of t in which x_2 and x_3 are replaced by the results of invoking *remScript* on x_2 and x_3 respectively; (2) if $t = node(x_1, x_2, x_3)$ and its *tag* is equal to `"script"`, *remScript* outputs a copy of x_3, (3) if $t = nil$, *remScript* outputs a copy t. The transformation *esc*

[1] Some sanitizers process the input HTML as a string, often causing the output not to be standards compliant.
[2] Section 6 discusses why classic tree transducers do not scale in this case.

```
// Datatype definition for HTML encoding
type HtmlE[tag : String]{nil(0), val(1), attr(2), node(3)}
// Language of well-formed HTML trees
lang nodeTree:HtmlE {
      node(x1, x2, x3) given (attrTree x1) (nodeTree x2) (nodeTree x3)
   | nil() where (tag = "")   }
lang attrTree:HtmlE {
      attr(x1, x2) given (valTree x1) (attrTree x2)
   | nil() where (tag = "")   }
lang valTree:HtmlE {
      val(x1) where (tag ≠ "") given (valTree x1)
   | nil() where (tag = "")   }
// Sanitization functions
trans remScript:HtmlE->HtmlE {
      node(x1, x2, x3) where (tag ≠ "script") to
                                             (node [tag] x1 (remScript x2) (remScript x3))
   | node(x1, x2, x3) where (tag = "script") to x3
   | nil() to (nil [tag])   }
trans esc:HtmlE->HtmlE {
      node(x1, x2, x3) to (node [tag] (esc x1) (esc x2) (esc x3))
   | attr(x1, x2) to (attr [tag] (esc x1) (esc x2))
   | val(x1) where (tag = "'" ∨ tag = """) to (val ["\"](val [tag] (esc x1)))
   | val(x1) where (tag ≠ "'" ∧ tag ≠ """) to (val [tag] (esc x1))
   | nil() to (nil [tag])   }
// Compose remScript and esc, restrict to well-formed trees
def rem_esc:HtmlE->HtmlE := (compose remScript esc)
def sani:HtmlE->HtmlE := (restrict rem_esc nodeTree)
// Language of bad outputs that contain a "script" node
lang badOutput:HtmlE {
      node(x1, x2, x3) where (tag = "script")
   | node(x1, x2, x3) given (badOutput x2)
   | node(x1, x2, x3) given (badOutput x3)   }
// Check that no input produces a bad output
def bad_inputs:HtmlE := (pre-image sani badOutput)
assert-true (is-empty bad_inputs)
```

Fig. 2. Implementation and analysis of an HTML sanitizer in Fast.

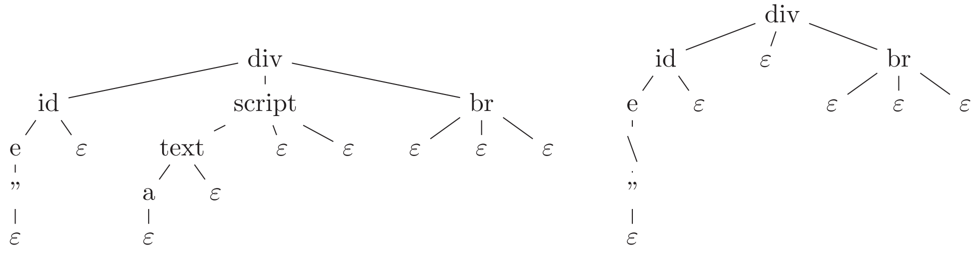

Fig. 3. (Left) *HtmlE* encoding of the HTML tree `<div id='e"'><script>a</script></div><br />`. `div`, `script`, and `br` are built using the constructor *node*. Nodes with tag `id`, and `text`, are built using *attr*. Single character nodes are built using *val*, and ε's using *nil*. The strings appearing in the figure are the *tags* of each node. Sanitizing this tree with the function *sani* of Figure 2 yields the *HtmlE* tree corresponding to `<div id='e\"'></div><br />`. (Right) *HtmlE* encoding of the HTML tree resulting from applying the transformation *sani* to the tree on the left of the figure.

(lines 19–24) of type *HtmlE->HtmlE* escapes the characters `’` and `"`, and it outputs a copy of the input tree in which each node *val* with tag `"’"` or `"""` is prepended a node *val* with tag `"\"`. The transformations *remScript* and *esc* are then composed into a single transformation *rem_esc* (line 26). This is done using transducer composition. The square bracket syntax is used to represent the assignments to the attribute tag. One might notice that *rem_esc* also accepts input trees that are not in the language *nodeTree* and do not correspond to correct encodings. Therefore, we compute the transformation *sani* (line 27), which is the same as *rem_esc*, but restricted to only accept inputs in the language *nodeTree*.

We can now use FAST to analyze the program *sani*. First, we define the language *bad_output* (lines 29–32), which accepts all the trees containing at least one node with tag `"script"`.[3] Next, using preimage computation, we compute the language *bad_inputs* (line 34) of inputs that produce a bad output. Finally, if *bad_inputs* is the empty language, *sani* never produces bad outputs. When running this program in FAST this checking (line 35) fails, and FAST provides the following counterexample:

node [`"script"`] *nil nil* (*node* [`"script"`] *nil nil nil*)

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

3. SYMBOLIC TREE TRANSDUCERS AND FAST

The concrete syntax of FAST is shown in Figure 4. FAST is designed for describing trees, tree languages, and functions from trees to trees. These are supported using *STAs*, and *STTRs*. This section covers these objects and how they describe the semantics of FAST.

3.1. Background

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

We use λ expressions for defining anonymous functions called λ *terms* without having to name them explicitly. In general, we use standard first-order logic and follow the notational conventions that are consistent with Veanes et al. [2012]. We write $f(v)$ for the functional application of the λ term f to the term v. We write σ for a type and the universe of elements of type σ is denoted by $\boldsymbol{\sigma}$. A σ predicate is a λ term $\lambda x.\varphi(x)$ where x has type σ, and φ is a formula for which the free variables $FV(\varphi)$ are contained in $\{x\}$. Given a σ predicate φ, $[\![\varphi]\!]$ denotes the set of all $a \in \boldsymbol{\sigma}$ such that $\varphi(a)$ holds. The set of σ predicates is denoted by $\Psi(\sigma)$. Given a type σ (such as *int*), we extend the universe with σ-labeled finite trees as an algebraic data type $\mathcal{T}^{\sigma}_{\Sigma}$ where Σ is a finite set of *tree constructors* f with *rank* $\natural(f) \geq 0$; f has type $\sigma \times (\mathcal{T}^{\sigma}_{\Sigma})^{\natural(f)} \rightarrow \mathcal{T}^{\sigma}_{\Sigma}$.[4] We call $\mathcal{T}^{\sigma}_{\Sigma}$ a *tree*

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

[4] When $\natural(f) = 0$ then f has type $\sigma \rightarrow \mathcal{T}^{\sigma}_{\Sigma}$.

```
Identifiers               ID : (a..z|A..Z|_)(a..z|A..Z|_|.|0..9)*
Basic types                σ : String | Int | Real | Bool ...
Built-in operators        op : < | > | = | + | and | or | ...
  Constructors             c : ID       Natural numbers    k : ℕ
  Tree types               τ : ID       Language states    p : ID
  Transformation states    q : ID       Attribute fields   x : ID
  Subtree variables        y : ID

Main definitions :
  Fast ::=  type τ [(x:σ)*] {(c(k))+} | tree t : τ := TR
         |  lang p : τ { Lrule+ } | trans q : τ -> τ { Trule+ }
         |  def p : τ := L | def q : τ -> τ := T
         |  assert-true A | assert-false A
 Lrule ::=  c(y1,...,yn) (where Aexp)? (given (( p y ))+)?
 Trule ::=  Lrule to Tout
  Tout ::=  y | ( q y ) | ( c[ Aexp+] Tout*)
  Aexp ::=  ID | Const | (op Aexp+)

Operations over languages, transductions, and trees :
     L ::=  (intersect L L) | (union L L) | (complement L) |
            (difference L L) | (L) | (domain T)
         |  (pre-image T L) | p
     T ::=  (compose T T) | (restrict T L) | (restrict-out T L) | q
    TR ::=  t | ( c[ Aexp*] TR*) | (apply T TR) | (get-witness L)
     A ::=  L == L | (is-empty L) | (is-empty T) | TR ∈ T
         |  (type-check L T L)
```

Fig. 4. Concrete syntax of Fast. Nonterminals and metasymbols are in italic. Constant expressions for strings and numbers use C# syntax [Hejlsberg et al. 2003]. Additional well-formedness conditions (such as well-typed terms) are assumed to hold.

type. Let $\Sigma(k) \stackrel{\text{def}}{=} \{f \in \Sigma \mid \natural(f) = k\}$. We require that $\Sigma(0)$ is *nonempty* so that $\mathcal{T}_\Sigma^\sigma$ is nonempty. We write $f[a](\bar{u})$ for $f(a, \bar{u})$ and abbreviate $f[a]()$ by $f[a]$.

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

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

3.2. Alternating Symbolic Tree Automata

We introduce and develop the basic theory of alternating symbolic tree automata, which adds a form of alternation to the basic definition originally presented in Veanes and Bjørner [2012]. We decide to use alternating STAs instead of their nonalternating counterpart because they are succinct and arise naturally when composing tree transducers.

Definition 1. An *Alternating Symbolic Tree Automaton (Alternating STA)* A is a tuple $(Q, \mathcal{T}^{\sigma}_{\Sigma}, \delta)$, where Q is a finite set of *states*, $\mathcal{T}^{\sigma}_{\Sigma}$ is a *tree type*, and $\delta \subseteq \bigcup_{k\geq 0}(Q \times \Sigma(k) \times \Psi(\sigma) \times (2^{Q})^{k})$ is a finite set of *rules* $(q, f, \varphi, \bar{\ell})$, where q is the *source state*, f the *symbol*, φ the *guard*, and $\bar{\ell}$ the *look-ahead*.

Next, we define the semantics of an STA $A = (Q, \mathcal{T}^{\sigma}_{\Sigma}, \delta)$.

Definition 2. For every state $q \in Q$ the *language of* A *at* q, is the set

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

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

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

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

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

The semantics of a Fast language is given by the induced STA.

Example 3.2. Consider the following Fast program:

$$\begin{array}{l}
\textbf{type}\ BT[i : \text{Int}]\{L(0), N(2)\} \\
\textbf{lang}\ p{:}BT\ \{L()\ \textbf{where}\ (i > 0) \mid N(x, y)\ \textbf{given}\ (p\ x)\ (p\ y)\} \\
\textbf{lang}\ o{:}BT\ \{L()\ \textbf{where}\ (odd\ i) \mid N(x, y)\ \textbf{given}\ (o\ x)\ (o\ y)\} \\
\textbf{lang}\ q{:}BT\ \{N(x, y)\ \textbf{given}\ (p\ y)\ (o\ y)\}.
\end{array}$$

An equivalent STA A over $\mathcal{T}^{\mathtt{Int}}_{\mathrm{BT}}$ has states $\{o, p, q\}$ and rules

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

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

In the following we say STA for alternating STA.[5]

Definition 3. A is *normalized* if for all $(p, f, \varphi, \bar{\ell}) \in \delta$, and all i, $1 \leq i \leq \natural(f)$, ℓ_i is a singleton set.

For example, the STA in Example 3.2 is not normalized because of the rule with source q. Normalization is a practically useful operation of STAs that is used on several occasions.

[5] When compared to the model in Comon et al. [2007], the STAs defined above are "almost" alternating, in the sense that they can only allow disjunctions of conjunctions, rather than arbitrary positive Boolean combinations. Concretely, the look-ahead of a rule r corresponds to a *conjunction* of states, while several rules from the same source state provide a *disjunction* of cases.

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

$$\begin{aligned}
\boldsymbol{\delta}^f(\emptyset) &= \{(\emptyset, f, \emptyset, (\emptyset)_{i=1}^{\natural(f)})\}, \\
\boldsymbol{\delta}^f(\boldsymbol{p} \cup \boldsymbol{q}) &= \{r \mathbin{\wedge\!\!\!\wedge} s \mid r \in \boldsymbol{\delta}^f(\boldsymbol{p}), s \in \boldsymbol{\delta}^f(\boldsymbol{q})\}, \\
\boldsymbol{\delta}^f(\{p\}) &= \{(\{p\}, f, \{\varphi\}, \bar{\rho}) \mid (p, f, \varphi, \bar{\rho}) \in \delta\},
\end{aligned}$$

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

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

Definition 4. The *normalized form of A* is the STA

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

The original rules of the normalized form are precisely the ones for which the states are singleton sets in 2^Q.[6] As expected, normalization preserves the language semantics of STAs.

THEOREM 3.3. *For all* $\mathbf{q} \subseteq Q$, $\mathbf{L}^{\mathbf{q}}_A = \mathbf{L}^{\mathbf{q}}_{\mathcal{N}(A)}$.

PROOF. The case when $\mathbf{q} = \emptyset$ is clear because the state $\emptyset$ in $\mathcal{N}(A)$ has the same semantics as $\mathbf{L}^{\emptyset}_A$. Assume $\mathbf{q} \neq \emptyset$. We show (1) for all $t \in \mathcal{T}^{\sigma}_{\Sigma}$:

$$t \in \mathbf{L}^{\mathbf{q}}_A \Leftrightarrow t \in \mathbf{L}^{\mathbf{q}}_{\mathcal{N}(A)}. \tag{1}$$

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

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

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

$$\begin{aligned}
t \in \mathbf{L}^{\mathbf{q}}_A &\Leftrightarrow \bigwedge_{i=1}^{2} t \in \mathbf{L}^{q_i}_A \\
&\Leftrightarrow \bigwedge_{i=1}^{2} \exists \varphi^i, \boldsymbol{p}^i_1, \boldsymbol{p}^i_2 : (q_i, f, \varphi^i, (\boldsymbol{p}^i_1, \boldsymbol{p}^i_2)) \in \delta_A, a \in [\![\varphi^i]\!], t_1 \in \mathbf{L}^{\boldsymbol{p}^i_1}_A, t_2 \in \mathbf{L}^{\boldsymbol{p}^i_2}_A \\
&\Leftrightarrow \bigwedge_{i=1}^{2} \exists \varphi^i, \boldsymbol{p}^i_1, \boldsymbol{p}^i_2 : (\{q_i\}, f, \{\varphi^i\}, (\boldsymbol{p}^i_1, \boldsymbol{p}^i_2)) \in \boldsymbol{\delta}^f, a \in [\![\varphi^i]\!], t_1 \in \mathbf{L}^{\boldsymbol{p}^i_1}_A, t_2 \in \mathbf{L}^{\boldsymbol{p}^i_2}_A
\end{aligned}$$

[6] In practice, merged rules are computed lazily starting from the initial state. Merged rules with unsatisfiable guards $\boldsymbol{\varphi}$ are eliminated eagerly. New concrete states are created for all the reachable merged states. Finally, the normalized STA is cleaned by eliminating states that accept no trees, for example, by using elimination of useless symbols from a context-free grammar [Hopcroft and Ullman 1979, pp. 88–89].

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

The theorem follows by the induction principle. □

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

PROPOSITION 3.4. *The nonemptiness problem of STAs is decidable if the label theory is decidable.*

While normalization is always possible, an STA may be *exponentially* more succinct than the equivalent normalized STA. This is true already for the *classic* case, that is, when $\boldsymbol{\sigma} = \{()\}$.

PROPOSITION 3.5. *The nonemptiness problem of alternating STAs without attributes is* EXPTIME*-complete.*

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

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

Then $L(\mathcal{A})$ is nonempty if and only if $\boldsymbol{L}_{\boldsymbol{A}}^{\boldsymbol{q}}$ is nonempty. For inclusion in EXPTIME use Comon et al. [2007, Theorem 7.5.1].

For EXPTIME-hardness a converse reduction is not as simple because alternating tree automata allow general (positive) Boolean combinations of $Q \times \Sigma$ in the mapping Δ. Instead, let $A_i = (Q_i, \mathcal{T}_\Sigma, \delta_i)$ be classic top-down tree automata with initial states $q_i \in Q_i$ for $1 \leq i \leq n$ [Comon et al. 2007]. Consider all these automata as STAs without attributes and with pairwise disjoint Q_i. In particular, all A_i are normalized. Expand Σ to $\Sigma' = \Sigma \cup \{f\}$ where f is a fresh symbol of rank 1. Let A be the STA $(\{q\} \cup \bigcup_i Q_i, \mathcal{T}_{\Sigma'}, \bigcup_i \delta_i \cup \{(q, f, \lambda x.true, (\{q_i\}_{1 \leq i \leq n}))\})$ where q is a new state. It follows from the definitions that $\boldsymbol{L}_{\boldsymbol{A}}^{\boldsymbol{q}} \neq \emptyset$ if and only if $\bigcap_i \boldsymbol{L}_{\boldsymbol{A_i}}^{\boldsymbol{q_i}} \neq \emptyset$. EXPTIME-hardness follows now from the intersection nonemptiness problem of tree automata [Frühwirth et al. 1991] (already restricted to the top-down-deterministic case [Seidl 1994b]). □

3.3. Symbolic Tree Transducers with Regular Look-Ahead

STTs augment STAs with outputs. STTs with regular look-ahead further augment STTs by allowing rules to be guarded by STAs. Intuitively, a rule is applied to a node if and only if its children are accepted by some STAs. We first define terms that are used in the following as output components of transformation rules. We assume that we have a given tree type $\mathcal{T}_\Sigma^\sigma$ for both the input trees as well as the output trees. In the case that the input tree type and the output tree type are intended to be different, we assume

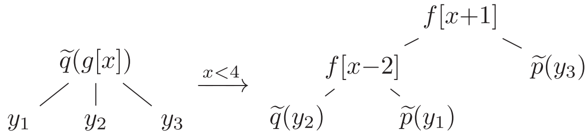

Fig. 5. A depiction of a linear rule of rank 3.

that $\mathcal{T}_{\Sigma}^{\sigma}$ is a combined tree type that covers both. The guards and the look-aheads can be used to restrict the types as needed.

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

Definition 5. Given a tree type $\mathcal{T}_{\Sigma}^{\sigma}$, a finite set Q of states, and $k \geq 0$, the set $\Lambda(\mathcal{T}_{\Sigma}^{\sigma}, Q, k)$ is defined as the least set S of λ terms called *k-rank tree transformers* that satisfies the following conditions, let $\bar{y}$ be a k tuple of variables of type $\mathcal{T}_{\Sigma\cup\{State\}}^{\sigma}$ and let x be a variable of type σ,

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

Definition 6. A *STTR T* is a tuple $(Q, q^0, \mathcal{T}_{\Sigma}^{\sigma}, \Delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $\mathcal{T}_{\Sigma}^{\sigma}$ is the *tree type*, $\Delta \subseteq \bigcup_{k\geq 0}(Q \times \Sigma(k) \times \Psi(\sigma) \times (2^Q)^k \times \Lambda(\mathcal{T}_{\Sigma}^{\sigma}, Q, k))$ is a finite set of *rules* $(q, f, \varphi, \bar{\ell}, t)$, where t is the *output*.[7] A rule is *linear* if its output is $\lambda(x, \bar{y}) \cdot u$ where each y_i occurs at most once in u. T is *linear* when all rules of T are linear.

A rule $(q, f, \varphi, \bar{\ell}, t)$ is also denoted by $q \xrightarrow{f,\varphi,\bar{\ell}} t$. The *open view* of a rule $q \xrightarrow{f,\varphi,\bar{\ell}} t$ is $\widetilde{q}(f[x](\bar{y})) \xrightarrow{\varphi(x),\bar{\ell}} t(x, \bar{y})$. The open view is technically more convenient and more intuitive for *term rewriting*. The look-ahead, when omitted, is $\bar{\emptyset}$ by default. Figure 5 illustrates an open view of a linear rule over the tree type $\mathcal{T}_{\Sigma_1}^{\mathrm{Int}}$ over $\Sigma_1 = \{f, g, h\}$, where $\natural(f) = 2$, $\natural(g) = 3$, and $\natural(h) = 0$.

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

Definition 7. The *domain automaton of* T, $\mathbf{d}(T)$, is the STA $(Q, \mathcal{T}_{\Sigma}^{\sigma}, \{(q, f, \varphi, (\ell_i \cup St(i, t))_{i=1}^{\natural(f)}) \mid q \xrightarrow{f,\varphi,\bar{\ell}} t \in \Delta\})$.

The rules of the domain automaton also take into account the states that occur in the outputs in addition to the look-ahead states. For example, the rule in Figure 5 yields the domain automaton rule $(q, g, \lambda x.x < 4, (\{p\}, \{q\}, \{p\}))$.

[7] For $k = 0$ we assume that $(2^Q)^k = \{()\}$, that is, a rule for $c \in \Sigma(0)$ has the form $(q, c, \varphi, (), \lambda x.t(x))$ where $t(x)$ is a tree term.

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

Definition 8. For all $q \in Q_T$, the *transduction of* T *at* q is the function $\mathbf{T}_T^q$ from $\mathcal{T}_\Sigma^\sigma$ to $2^{\mathcal{T}_\Sigma^\sigma}$ such that, for all $t = f[a](\bar{s}) \in \mathcal{T}_\Sigma^\sigma$,

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

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

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

Example 3.6. Recall the transformation *remScript* in Figure 2. These are the corresponding rules. We use q for the state of *remScript*, and ι for a state that outputs the identity transformation. The "safe" case is

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

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

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

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

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

Example 3.7. The following STTR describes the function h that negates a node value when the value in its left child is odd, leaves it unchanged otherwise, and is then invoked recursively on the children.

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

The following property of STTRs will be used in Section 4.

Definition 9. T is *single valued* if $\forall(t \in \mathcal{T}^{\sigma}_{\Sigma}, q \in Q_T) : |\mathbf{T}^{q}_{T}(t)| \leq 1$.

Determinism, as defined next, implies single-valuedness and determinism is easy to decide. Intuitively, determinism means that there are no two distinct transformation rules that are enabled for the same node of any input tree. Although single-valuedness can be decided in the classic case [Esik 1980], decidability of single-valuedness of STTRs is an open problem.

Definition 10. T is *deterministic* when, for all $q \in Q$, $f \in \Sigma$, and all rules $q \xrightarrow{f,\varphi,\bar{\ell}} t$ and $q \xrightarrow{f,\psi,\bar{r}} u$ in Δ_T, if $[\![\varphi]\!] \cap [\![\psi]\!] \neq \emptyset$ and, for all $i \in \{1,\ldots,\natural(f)\}$, $\mathbf{L}^{\ell_i} \cap \mathbf{L}^{r_i} \neq \emptyset$, then $t = u$.

3.4. The Role of Regular Look-Ahead

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

Example 3.8. Consider the following Fast program:

```
type BBT[b : Bool]{L(0), N(2)}
trans s1 :BBT ->BBT{
    L() where b to (L[b])
  | N(x, y) where b to (N[b] (s1 x) (s1 y))
}
trans s2 :BBT->BBT {
    L() to (L[true])
  | N(x, y) to (L[true])
}.
```

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

Example 3.7 shows that STTRs are sometimes more convenient to use than STTs. Although the transformation h can be expressed using a nondeterministic STT that guesses if the attribute of the left child is odd or even, using a deterministic STTR is a more natural solution.

3.5. Operations on Automata and Transducers

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

—*Operations that compute new languages:*
intersect A_1 A_2, ***complement*** A, *etc.* Operations over STAs [Veanes and Bjørner 2015];
domain T. Computes the domain of the STTR T using the operation from Definition 7;

pre-image T A. Computes an STA accepting all the inputs for which T produces an output belonging to A.

—*Operations that compute new transformations:*

restrict T A. Constructs a new STTR that behaves like T, but is only defined on the inputs that belong to A.

restrict-out T A. Constructs a new STTR that behaves like T, but is only defined on the inputs for which T produces an output that belongs to A.

compose T_1 T_2. Constructs a new STTR that computes the functional composition $T_1 \circ T_2$ of T_1 and T_2 (algorithm described in Section 4).

—*Assertions:*

$a \in A$, $A_1 = A_2$, ***is-empty*** A. Decision procedures for STAs. In the order, membership, language equivalence, and emptiness (Proposition 3.4 and Veanes and Bjørner [2015]).

type-check A_1 T A_2. True if and only if for every input in A_1, T only produces outputs in A_2.

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

Proposition 3.9. *The following operations can be expressed as*

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

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

4. COMPOSITION OF STTRS

Closure under composition is a fundamental property for transducers. Composition is needed as a building block for many operations, such as preimage computation and output restriction. Unfortunately, as shown in Example 3.8 and in Fülöp and Vogler [2014], STTs are not closed under composition. Particularly, when tree rules may *delete* and/or *duplicate* input subtrees, the composition of two STT transductions might not be expressible as an STT transduction. This is already known for classic tree transducers and can be avoided either by considering restricted fragments, or by instead adding regular look-ahead [Engelfriet 1975; Baker 1979; Engelfriet 1980]. In this article, we consider the latter option. Intuitively, regular look-ahead acts as an additional child guard that is carried over in the composition so that even when a subtree is deleted, the child guard remains in the composed transducer and is not “forgotten.” While deletion can be handled by STTRs, duplication is a much more difficult feature to support. When duplication is combined with nondeterminism, as shown in the next example, it is still not possible to compose STTRs. In practice this case is unusual, and it can only appear when programs produce more than one output for a given input.

Example 4.1. Let f be the function that, given a tree of type BT (see Example 3.2) transforms it by nondeterministically replacing some leaves with the value 5.

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

Let g be the function that given any tree t always outputs $N[0](t, t)$.

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

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

4.1. Composition Algorithm

Algorithms for composing transducers with regular look-ahead have been studied extensively [Fülöp and Vágvölgyi 1989]. However, as shown in Fülöp and Vogler [2014], extending classic transducers results to the symbolic setting is a far from trivial task. The key property that makes symbolic transducers semantically different and much more challenging than classic tree transducers, apart from the complexity of the label theory itself, is the *output computation*. In symbolic transducers the output attributes depend *symbolically* on the input attribute. Effectively, this breaks the application of some well-established classic techniques that no longer carry over to the symbolic setting. For example, while for classic tree transducers the output language is always regular, this is not the case for symbolic transducers. Such anomaly is caused by the fact that the input attribute can appear more than once in the output of a rule.

Let S and T be two STTRs with disjoint sets of states Q_S and Q_T, respectively. We want to construct a composed STTR $S \circ T$ such that, $\mathbf{T}_{S \circ T} = \mathbf{T}_S \circ \mathbf{T}_T$. The composition $\mathbf{T}_S \circ \mathbf{T}_T$ is defined as $(\mathbf{T}_S \circ \mathbf{T}_T)(x) = \bigcup_{y \in \mathbf{T}_S(x)} \mathbf{T}_T(y)$, following the convention in Fülöp and Vogler [1998].

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

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

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

Example 4.2. Suppose $\widetilde{p}(f[x](y_1, y_2)) \xrightarrow[S]{x>0} \widetilde{p}(y_2)$. Assume also that $q \in Q_T$ and that $p \cdot q$ has been reached. Then

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

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

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

and $T = (Q_T, q^0_T, \mathcal{T}^\sigma_\Sigma, \Delta_T)$, the composition of S and T is defined as follows:

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

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

```
$\mathbf{Compose}(p, q, f) \stackrel{\text{def}}{=}$
(1) $\textit{\textbf{choose}}\ (p, f, \varphi, \bar{\ell}, u)\ \textit{\textbf{from}}\ \Delta_S$
(2) $\textit{\textbf{choose}}\ (\psi, \bar{P}, t)\ \textit{\textbf{from}}\ \mathbf{Reduce}(\varphi, (\emptyset)_{i=1}^{\natural(f)}, \widetilde{q}(u))$
(3) $\textit{\textbf{return}}(p.q, f, \psi, \bar{\ell} \uplus \bar{P}, t)$
```

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

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

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

Example 4.3. Consider the instance $\mathbf{Look}(x{>}0, \bar{\emptyset}, q, t)$ for $t = g[x{+}1](g[x{-}2](\widetilde{p}_1(y_2)))$ where $g \in \Sigma(1)$. Suppose there is a rule $(q, g, \lambda x \cdot odd(x), \{q\}) \in \delta_{\mathbf{d}(T)}$ that requires that all attributes of g are odd and assume that there is no other rule for g from q. The term t itself may arise as an output of a rule $\widetilde{p}(f[x](y_1, y_2)) \rightarrow g[x{+}1](g[x{-}2](\widetilde{p}_1(y_2)))$ of S. Clearly, this outrules t as a valid input of T at q because of the cross-level dependency between attributes due to x, implying that both attributes cannot be odd at the same time. Let us examine how this is handled by the **Look** procedure.

In $\mathbf{Look}(x{>}0, \bar{\emptyset}, q, t)$ line 2(c) we have the recursive call $\mathbf{Look}(x{>}0 \wedge odd(x{+}1), \bar{\emptyset}, q, g[x{-}2](\widetilde{p}_1(y_2)))$. Inside the recursive call we have the failing satisfiability check of $\textit{IsSat}(x{>}0 \wedge odd(x{+}1) \wedge odd(x{-}2))$ in line 2(a). So that there exists no choice for which 2(d) is reached in the original call so the set of return values of $\mathbf{Look}(x{>}0, \bar{\emptyset}, q, t)$ is empty.

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

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

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

There is a close relationship between **Reduce** and Definition 8. We include the case

$$\mathbf{T}^q_T(\widetilde{p}(t)) \stackrel{\text{def}}{=} \mathbf{T}^q_T(\mathbf{T}^p_S(t)) \quad \text{for } p \in Q_S \text{ and } t \in \mathcal{T}^\sigma_\Sigma, \tag{2}$$

that allows states of S to occur in the input trees to $\mathbf{T}^q_T$ in a nonnested manner. Intuitively this means that rewrite steps of T are carried out first while rewrite steps of S are being postponed (called by name). In order to justify the extension (2) we need the following Lemma.

Lemma 4.4. *For all $t \in \Lambda(\mathcal{T}^\sigma_\Sigma, Q_S, k)$, $\mathtt{a} \in \boldsymbol{\sigma}$, and $\mathtt{u}_i \in \mathcal{T}^\sigma_\Sigma$:*

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

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

$$\mathbf{T}^q_t(\Downarrow_S(\widetilde{p}(\mathtt{u_i}))) = \mathbf{T}^q_t(\mathbf{T}^p_S(\mathtt{u_i})) = \mathbf{T}^q_t(\widetilde{p}(\mathtt{u_i})),$$

where the last equality holds by using Equation (2). The induction case is as follows. Let $t = \lambda(x, \bar{y}) \cdot f[t_0(x)](t_i(x, \bar{y})_{i=1}^{\natural(f)})$. Suppose $\natural(f) = 1$, the proof of the general case is analogous.

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

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

The step ($\star$) becomes “ ” when either $|\Downarrow_S(t_1(\mathtt{a}, \bar{\mathtt{u}}))| \leq 1$ or when $m \leq 1$. The first case holds if S is single-valued. The second case holds if T is linear in which case also the induction step becomes “=.” Both statements of the lemma follow by using the induction principle. □

Example 4.5. The example shows a case when

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

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

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

but

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

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

The assumptions on S and T given in Lemma 4.4 are the same as in the classic setting; however, the proof of Lemma 4.4 does not directly follow from classic results because the concrete alphabet $\Sigma \times \boldsymbol{\sigma}$ can be infinite. Theorem 4.6 generalizes to symbolic alphabets the composition result proven in Theorem 2.11 of Engelfriet [1977]. Theorem 4.6 uses Lemma 4.4. It implies that, in general, $\mathbf{T}_{S\circ T}$ is an overapproximation of $\mathbf{T}_S \circ \mathbf{T}_T$ and that $\mathbf{T}_{S\circ T}$ captures $\mathbf{T}_S \circ \mathbf{T}_T$ precisely when either S behaves as a partial function or when T does not duplicate its tree arguments.

THEOREM 4.6. *For all $p \in Q_S$, $q \in Q_T$ and $t \in \mathcal{T}_\Sigma^\sigma$, $\mathbf{T}_{S\circ T}^{p.q}(t) \supseteq \mathbf{T}_t^q(\mathbf{T}_S^p(t))$, and if S is single-valued or if T is linear, then $\mathbf{T}_{S\circ T}^{p.q}(t) \subseteq \mathbf{T}_t^q(\mathbf{T}_S^p(t))$.*

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

$$\begin{aligned}
\mathbf{L}^{p\cdot q} &\overset{\text{def}}{=} \{t \mid \mathbf{T}_t^q(\mathbf{T}_S^p(t)) \neq \emptyset\} \\
&= \{t \mid \exists u(u \in \mathbf{T}_S^p(t) \wedge \mathbf{T}_t^q(u) \neq \emptyset)\} \\
&= \{t \mid \exists u(u \in \mathbf{T}_S^p(t) \wedge u \in \mathbf{L}_T^q)\} \\
&= \{t \mid \mathbf{T}_S^p(t) \cap \mathbf{L}_T^q \neq \emptyset\}.
\end{aligned}$$

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

$$\mathbf{L}^{P} \stackrel{\text{def}}{=} \bigcap_{p.q \in P} \mathbf{L}^{p \cdot q},$$

$$\mathbf{L}^{\bar{P}} \stackrel{\text{def}}{=} \{\bar{u} \mid \bigwedge_{i=1}^{k} u_i \in \mathbf{L}^{P_i}\}.$$

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

$$\begin{aligned}&\llbracket \mathbf{Look}(\varphi, \bar{P}, q, t) \rrbracket \stackrel{\text{def}}{=} \\ &\{(a, \bar{u}) | a \in \llbracket \varphi \rrbracket, \bar{u} \in \mathbf{L}^{\bar{P}}, \Downarrow_S(t(a, \bar{u})) \cap \mathbf{L}^{q}_{T} \neq \emptyset\}.\end{aligned} \tag{3}$$

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

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

We now prove (4). It is the link between the symbolic and the concrete semantics of **Look** and Definition 2.

$$\bigcup\{\llbracket \chi \rrbracket | \mathbf{Look}(\varphi, \bar{P}, q, t) \ \boldsymbol{returns}\ \chi\} = \llbracket \mathbf{Look}(\varphi, \bar{P}, q, t) \rrbracket. \tag{4}$$

We prove (4) by induction over t. The base case is when $t = \widetilde{p}(y_i)$ for some $p \in Q_S$ and y_i for some $i \in \{1, \ldots, k\}$:

$$\begin{aligned}&\bigcup\{\llbracket \chi \rrbracket | \mathbf{Look}(\varphi, \bar{P}, q, \widetilde{p}(y_i)) \ \boldsymbol{returns}\ \chi\} \\ &\quad = \llbracket (\varphi, \bar{P} \uplus_i p.q) \rrbracket \\ &\quad = \{(a, \bar{u}) \mid a \in \llbracket \varphi \rrbracket, \bar{u} \in \mathbf{L}^{\bar{P}}, u_i \in \mathbf{L}^{p \cdot q}\} \\ &\quad = \{(a, \bar{u}) \mid a \in \llbracket \varphi \rrbracket, \bar{u} \in \mathbf{L}^{\bar{P}}, \mathbf{T}^{p}_{S}(u_i) \cap \mathbf{L}^{q}_{T} \neq \emptyset\} \\ &\quad = \{(a, \bar{u}) \mid a \in \llbracket \varphi \rrbracket, \bar{u} \in \mathbf{L}^{\bar{P}}, \Downarrow_S(\widetilde{p}(u_i)) \cap \mathbf{L}^{q}_{T} \neq \emptyset\} \\ &\quad = \llbracket \mathbf{Look}(\varphi, \bar{P}, q, \widetilde{p}(u_i)) \rrbracket.\end{aligned}$$

The induction case is when $t = f[t_0](\bar{t})$. Assume $\natural(f) = 2$. IH is that (4) holds for t_1 and t_2. Assume, without loss of generality, that $\mathbf{d}(T)$ is normalized. We have for all $a \in \boldsymbol{\sigma}$ and $\bar{u} \in (\boldsymbol{\mathcal{T}}^{\sigma}_{\Sigma})^k$,

$$\begin{aligned}&(a, \bar{u}) \in \bigcup\{\llbracket \chi \rrbracket \mid \mathbf{Look}(\varphi, \bar{P}, q, f[t_0](\bar{t})) \ \boldsymbol{returns}\ \chi\} \\ &\overset{(\text{Def}\,\mathbf{Look})}{\Leftrightarrow} \ (\text{exists } \psi, q_1, q_2)\, (q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)}, \\ &\qquad IsSat(\varphi \wedge \psi(t_0)), \\ &\qquad (\text{exists } \varphi', \bar{P}', \varphi'', \bar{P}'') \\ &\qquad \mathbf{Look}(\varphi \wedge \psi(t_0), \bar{P}, q_1, t_1) \ \boldsymbol{returns}\ (\varphi', \bar{P}'), \\ &\qquad \mathbf{Look}(\varphi', \bar{P}', q_2, t_2) \ \boldsymbol{returns}\ (\varphi'', \bar{P}''), \\ &\qquad (a, \bar{u}) \in \llbracket (\varphi'', \bar{P}'') \rrbracket\end{aligned}$$

$$
\begin{aligned}
\overset{\text{(IH)}}{\Leftrightarrow}\quad & (\text{exists } \psi, q_1, q_2)\,(q,\ f,\ \psi,\ (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)},\\
& \mathit{IsSat}(\varphi \wedge \psi(t_0)),\\
& (\text{exists } \varphi', \bar{P}')\\
& \mathbf{Look}(\varphi \wedge \psi(t_0), \bar{P}, q_1, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}'),\\
& (a, \bar{u}) \in [\![\mathbf{Look}(\varphi', \bar{P}', q_2, t_2)]\!]\\
\overset{\text{(Eq (3))}}{\Leftrightarrow}\quad & (\text{exists } \psi, q_1, q_2)\,(q,\ f,\ \psi,\ (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)},\\
& \mathit{IsSat}(\varphi \wedge \psi(t_0)),\\
& (\text{exists } \varphi', \bar{P}')\\
& \mathbf{Look}(\varphi \wedge \psi(t_0), \bar{P}, q_1, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}'),\\
& a \in [\![\varphi']\!], \bar{u} \in \mathbf{L}^{\bar{P}'}, \Downarrow_S(t_2(a, \bar{u})) \cap \mathbf{L}_T^{q_2} \neq \emptyset\\
\overset{\text{(IH)}}{\Leftrightarrow}\quad & (\text{exists } \psi, q_1, q_2)\,(q,\ f,\ \psi,\ (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)},\\
& \mathit{IsSat}(\varphi \wedge \psi(t_0)),\\
& (a, \bar{u}) \in [\![\mathbf{Look}(\varphi \wedge \psi(t_0), \bar{P}, q_1, t_1)]\!],\\
& \Downarrow_S(t_2(a, \bar{u})) \cap \mathbf{L}_T^{q_2} \neq \emptyset\\
\overset{\text{(Eq (3))}}{\Leftrightarrow}\quad & (\text{exists } \psi, q_1, q_2)\,(q,\ f,\ \psi,\ (\{q_1\}, \{q_2\})) \in \delta_{\mathbf{d}(T)},\\
& \mathit{IsSat}(\varphi \wedge \psi(t_0)),\\
& a \in [\![\varphi]\!] \cap [\![\psi(t_0)]\!], \bar{u} \in \mathbf{L}^{\bar{P}},\\
& \Downarrow_S(t_1(a, \bar{u})) \cap \mathbf{L}_T^{q_1} \neq \emptyset, \Downarrow_S(t_2(a, \bar{u})) \cap \mathbf{L}_T^{q_2} \neq \emptyset\\
\overset{\text{(Def 2)}}{\Leftrightarrow}\quad & a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}},\\
& \Downarrow_S(f[t_0(a)](t_1(a, \bar{u}), t_2(a, \bar{u}))) \cap \mathbf{L}_T^{q} \neq \emptyset\\
\Leftrightarrow\quad & a \in [\![\varphi]\!], \bar{u} \in \mathbf{L}^{\bar{P}}, \Downarrow_S(t(a, \bar{u})) \cap \mathbf{L}_T^{q} \neq \emptyset\\
\overset{\text{(Eq (3))}}{\Leftrightarrow}\quad & (a, \bar{u}) \in [\![\mathbf{Look}(\varphi, \bar{P}, q, t)]\!]
\end{aligned}
$$

Equation (4) follows by the induction principle. Observe that, so far, no assumptions on S or T were needed.

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

$$
\boldsymbol{\partial}_{(\varphi,\bar{P},t)}(\mathsf{a}, \bar{\mathsf{u}}) \overset{\text{def}}{=} \begin{cases} \Downarrow_T(t(\mathsf{a}, \bar{\mathsf{u}})), & \text{if } (\mathsf{a}, \bar{\mathsf{u}}) \in [\![(\varphi, \bar{P})]\!];\\ \emptyset, & \text{otherwise.} \end{cases} \tag{5}
$$

Next, we prove (6) under the assumption that S is single-valued or T is linear. For all $\mathsf{a} \in \boldsymbol{\sigma}$, $\mathsf{u}_i \in \boldsymbol{\mathcal{T}}^{\sigma}_{\boldsymbol{\Sigma}}$, and $\mathsf{v} \in \boldsymbol{\mathcal{T}}^{\sigma}_{\boldsymbol{\Sigma}}$,

$$
\begin{aligned}
& (\exists\alpha)\, \mathsf{v} \in \boldsymbol{\partial}_{\alpha}(\mathsf{a}, \bar{\mathsf{u}}), \mathbf{Reduce}(\varphi, \bar{P}, t)\ \boldsymbol{returns}\ \alpha\\
& \Leftrightarrow \mathsf{v} \in \boldsymbol{\partial}_{(\varphi,\bar{P},t)}(\mathsf{a}, \bar{\mathsf{u}}).
\end{aligned} \tag{6}
$$

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

The base case is $t = \widetilde{q}(\widetilde{p}(y_i))$ where $q \in Q_T$, $p \in Q_S$, and (6) follows because $\mathbf{Reduce}(\varphi, \bar{P}, \widetilde{q}(\widetilde{p}(y_i)))$ returns $(\varphi, \bar{P}, \widetilde{p \cdot q}(y_i))$ and $\lambda y \cdot \widetilde{p.q}(y)$ denotes, by definition, the composition $\lambda y \cdot \widetilde{q}(\widetilde{p}(y))$.

We use the extended case (7) of Definition 8 that allows states of S to occur in $\bar{\mathtt{t}}$. This extension is justified by Lemma 4.4. For $q \in Q_T$:

$$\begin{aligned}&\Downarrow_T(\widetilde{q}(f[\mathtt{a}](\bar{\mathtt{t}}))) \stackrel{\text{def}}{=}\\ &\quad\bigcup\{\Downarrow_T(u(\mathtt{a},\bar{\mathtt{t}})) \mid (q,\ f,\ \varphi,\ \bar{\ell},\ u)\in\Delta_T,\ \mathtt{a}\in[\![\varphi]\!],\\ &\qquad\qquad \bigwedge_{i=1}^{\natural(f)} \Downarrow_S(\mathtt{t}_i)\cap \mathbf{L}_T^{\ell_i}\neq\emptyset\}.\end{aligned} \tag{7}$$

Observe that when $\mathtt{t}_i$ does not contain any states of S then $\Downarrow_S(\mathtt{t}_i) = \{\mathtt{t}_i\}$ and thus the condition $\Downarrow_S(\mathtt{t}_i)\cap\mathbf{L}_T^{\ell_i} \neq \emptyset$ simplifies to the condition $\mathtt{t}_i \in \mathbf{L}_T^{\ell_i}$ used in the original version of Definition 8.

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

$$\begin{aligned}
&(\exists\alpha)\ \mathtt{v} \in \boldsymbol{\partial}_\alpha(\mathtt{a},\bar{\mathtt{u}}),\ \mathbf{Reduce}(\varphi,\bar{P},\widetilde{q}(t'))\ \textbf{\textit{returns}}\ \alpha\\
\overset{\text{Def }\mathbf{Reduce}}{\Leftrightarrow}\ &(\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T\\
&(\exists\psi,\bar{R})\ \mathbf{Look}(\varphi,\bar{P},q_\tau,t')\ \textbf{\textit{returns}}\ (\psi,\bar{R})\\
&(\exists\beta)\ \mathbf{Reduce}(\psi,\bar{R},u(t_0,\bar{t}))\ \textbf{\textit{returns}}\ \beta\\
&\mathtt{v} \in \boldsymbol{\partial}_\beta(\mathtt{a},\bar{\mathtt{u}})\\
\overset{\text{IH}}{\Leftrightarrow}\ &(\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T\\
&(\exists\psi,\bar{R})\ \mathbf{Look}(\varphi,\bar{P},q_\tau,t')\ \textbf{\textit{returns}}\ (\psi,\bar{R})\\
&\mathtt{v} \in \boldsymbol{\partial}_{(\psi,\bar{R},u(t_0,\bar{t}))}(\mathtt{a},\bar{\mathtt{u}})\\
\overset{\text{Eq (5)}}{\Leftrightarrow}\ &(\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T\\
&(\exists\psi,\bar{R})\ \mathbf{Look}(\varphi,\bar{P},q_\tau,t')\ \textbf{\textit{returns}}\ (\psi,\bar{R})\\
&\mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}}))),\ (\mathtt{a},\bar{\mathtt{u}}) \in [\![(\psi,\bar{R})]\!]\\
\overset{\text{Eq (4)}}{\Leftrightarrow}\ &(\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T\\
&(\mathtt{a},\bar{\mathtt{u}}) \in [\![\mathbf{Look}(\varphi,\bar{P},q_\tau,t')]\!]\\
&\mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}})))\\
\overset{\text{Eq (3)}}{\Leftrightarrow}\ &(\exists\tau,u,\gamma,\bar{\ell})\ \tau = q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T\\
&\mathtt{a} \in [\![\varphi]\!],\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}},\ \ \Downarrow_S(t'(\mathtt{a},\bar{\mathtt{u}})) \cap \mathbf{L}_T^{q_\tau} \neq \emptyset\\
&\mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}})))\\
\overset{\text{Def }q_\tau}{\Leftrightarrow}\ &\mathtt{a} \in [\![\varphi]\!],\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}}\\
&(\exists u,\gamma,\bar{\ell})\ q \xrightarrow{f,\gamma,\bar{\ell}} u \in \Delta_T\\
&t_0(\mathtt{a}) \in [\![\gamma]\!],\\
&\bigwedge_{i=1}^{\natural(f)} \Downarrow_S(t_i(\mathtt{a},\bar{\mathtt{u}})) \cap \mathbf{L}_T^{\ell_i} \neq \emptyset\\
&\mathtt{v} \in \Downarrow_T(u(t_0(\mathtt{a}),\bar{t}(\mathtt{a},\bar{\mathtt{u}})))\\
\overset{\text{Eq (7)}}{\Leftrightarrow}\ &\mathtt{a} \in [\![\varphi]\!],\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}},\ \ \mathtt{v} \in \Downarrow_T(t(\mathtt{a},\bar{\mathtt{u}}))\\
\overset{\text{Def }\boldsymbol{\partial}}{\Leftrightarrow}\ &\mathtt{v} \in \boldsymbol{\partial}_{(\varphi,\bar{P},t)}(\mathtt{a},\bar{\mathtt{u}}).
\end{aligned}$$

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

$$
\begin{aligned}
& (\exists \alpha)\ \mathtt{v} \in \boldsymbol{\partial}_{\alpha}(\mathtt{a}, \bar{\mathtt{u}}), \mathbf{Reduce}(\varphi, \bar{P}, f[t_0](t_1, t_2))\ \boldsymbol{returns}\, \alpha \\
\overset{\text{Def } \mathbf{Reduce}}{\Leftrightarrow}\ & (\exists\, \varphi', \bar{P}', v_1, \varphi'', \bar{P}'', v_2) \\
& \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', v_1) \\
& \mathbf{Reduce}(\varphi', \bar{P}', t_2)\ \boldsymbol{returns}\ (\varphi'', \bar{P}'', v_2) \\
& \mathtt{v} \in \boldsymbol{\partial}_{(\varphi'', \bar{P}'', f[t_0](v_1, v_2))}(\mathtt{a}, \bar{\mathtt{u}}) \\
\overset{\text{Def } \boldsymbol{\partial}}{\Leftrightarrow}\ & (\exists\, \varphi', \bar{P}', w_1, \varphi'', \bar{P}'', w_2) \\
& \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', w_1) \\
& \mathbf{Reduce}(\varphi', \bar{P}', t_2)\ \boldsymbol{returns}\ (\varphi'', \bar{P}'', w_2) \\
& \mathtt{v} \in \Downarrow_T(f[t_0(\mathtt{a})](w_1(\mathtt{a}, \bar{\mathtt{u}}), w_2(\mathtt{a}, \bar{\mathtt{u}}))), \\
& \mathtt{a} \in [\![\varphi'']\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}''} \\
\overset{\text{Def } \Downarrow_T}{\Leftrightarrow}\ & (\exists\, \varphi', \bar{P}', w_1, \varphi'', \bar{P}'', w_2) \\
& \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', w_1) \\
& \mathbf{Reduce}(\varphi', \bar{P}', t_2)\ \boldsymbol{returns}\ (\varphi'', \bar{P}'', w_2) \\
& (\exists\, \mathtt{v}_1, \mathtt{v}_2)\mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2) \\
& \mathtt{v}_1 \in \Downarrow_T(w_1(\mathtt{a}, \bar{\mathtt{u}})),\ \mathtt{v}_2 \in \Downarrow_T(w_2(\mathtt{a}, \bar{\mathtt{u}})) \\
& \mathtt{a} \in [\![\varphi'']\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}''} \\
\overset{\text{IH}}{\Leftrightarrow}\ & (\exists\, \varphi', \bar{P}', w_1) \\
& \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', w_1) \\
& (\exists\, \mathtt{v}_1, \mathtt{v}_2)\mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2) \\
& \mathtt{v}_1 \in \Downarrow_T(w_1(\mathtt{a}, \bar{\mathtt{u}})) \\
& \mathtt{v}_2 \in \boldsymbol{\partial}_{(\varphi', \bar{P}', t_2)}(\mathtt{a}, \bar{\mathtt{u}}) \\
\overset{\text{Def } \boldsymbol{\partial}}{\Leftrightarrow}\ & (\exists\, \varphi', \bar{P}', w_1) \\
& \mathbf{Reduce}(\varphi, \bar{P}, t_1)\ \boldsymbol{returns}\ (\varphi', \bar{P}', w_1) \\
& (\exists\, \mathtt{v}_1, \mathtt{v}_2)\mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2) \\
& \mathtt{v}_1 \in \Downarrow_T(w_1(\mathtt{a}, \bar{\mathtt{u}})) \\
& \mathtt{a} \in [\![\varphi']\!],\ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}'},\ \mathtt{v}_2 \in \Downarrow_T(t_2(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{\text{IH}}{\Leftrightarrow}\ & (\exists\, \mathtt{v}_1, \mathtt{v}_2)\mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2) \\
& \mathtt{v}_1 \in \boldsymbol{\partial}_{(\varphi, \bar{P}, t_1)}(\mathtt{a}, \bar{\mathtt{u}}) \\
& \mathtt{v}_2 \in \Downarrow_T(t_2(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{\text{Def } \boldsymbol{\partial}}{\Leftrightarrow}\ & (\exists\, \mathtt{v}_1, \mathtt{v}_2)\mathtt{v} = f[t_0(\mathtt{a})](\mathtt{v}_1, \mathtt{v}_2) \\
& \mathtt{a} \in [\![\varphi]\!], \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}}, \mathtt{v}_1 \in \Downarrow_T(t_1(\mathtt{a}, \bar{\mathtt{u}})) \\
& \mathtt{v}_2 \in \Downarrow_T(t_2(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{\text{Def } \Downarrow_T}{\Leftrightarrow}\ & \mathtt{a} \in [\![\varphi]\!], \bar{\mathtt{u}} \in \mathbf{L}^{\bar{P}} \\
& \mathtt{v} \in \Downarrow_T(f[t_0(\mathtt{a})](t_1(\mathtt{a}, \bar{\mathtt{u}}), t_2(\mathtt{a}, \bar{\mathtt{u}}))) \\
\overset{\text{Def } \boldsymbol{\partial}}{\Leftrightarrow}\ & \mathtt{v} \in \boldsymbol{\partial}_{(\varphi, \bar{P}, f[t_0](t_1, t_2))}
\end{aligned}
$$

Equation (6) follows by the induction principle.

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

$$
\begin{aligned}
& \mathtt{w} \in \mathbf{T}^{p.q}_{S\circ T}(f[\mathtt{a}](\bar{\mathtt{u}})) \\
\overset{\text{Def }\mathbf{Compose}}{\Leftrightarrow}\quad & (\exists\, \varphi, \bar{\ell}, t)\,(p,\ f,\ \varphi, \bar{\ell}, t) \in \Delta_S \\
& (\exists\, \alpha)\mathbf{Reduce}(\varphi, \bar{\emptyset}, \widetilde{q}(t))\ \boldsymbol{returns}\ \alpha \\
& \mathtt{w} \in \boldsymbol{\partial}_{\alpha}(\mathtt{a}, \bar{\mathtt{u}}),\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S} \\
\overset{\text{Eq (6)}}{\Leftrightarrow}\quad & (\exists\, \varphi, \bar{\ell}, t)\,(p,\ f,\ \varphi, \bar{\ell}, t) \in \Delta_S \\
& \mathtt{w} \in \boldsymbol{\partial}_{(\varphi,\bar{\emptyset},\widetilde{q}(t))}(\mathtt{a}, \bar{\mathtt{u}}), \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S} \\
\overset{\text{Def }\boldsymbol{\partial}}{\Leftrightarrow}\quad & (\exists\, \varphi, \bar{\ell}, t)\,(p,\ f,\ \varphi, \bar{\ell}, t) \in \Delta_S \\
& \mathtt{a} \in [\![\varphi]\!],\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\emptyset}},\ \ \mathtt{w} \in \Downarrow_T(\widetilde{q}(t(\mathtt{a}, \bar{\mathtt{u}}))),\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S} \\
\overset{\text{Def }\mathbf{T}^{q}_{t}}{\Leftrightarrow}\quad & (\exists\, \varphi, \bar{\ell}, t)\,(p,\ f,\ \varphi, \bar{\ell}, t) \in \Delta_S \\
& \mathtt{a} \in [\![\varphi]\!],\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S},\ \ \mathtt{w} \in \mathbf{T}^{q}_{t}(t(\mathtt{a}, \bar{\mathtt{u}})) \\
\overset{(\star)}{\Leftrightarrow}\quad & (\exists\, \varphi, \bar{\ell}, t)\,(p,\ f,\ \varphi, \bar{\ell}, t) \in \Delta_S \\
& \mathtt{a} \in [\![\varphi]\!],\ \ \bar{\mathtt{u}} \in \mathbf{L}^{\bar{\ell}}_{S},\ \ \mathtt{w} \in \mathbf{T}^{q}_{t}(\Downarrow_S(t(\mathtt{a}, \bar{\mathtt{u}}))) \\
\overset{\text{Def }\Downarrow_S}{\Leftrightarrow}\quad & \mathtt{w} \in \mathbf{T}^{q}_{t}(\Downarrow_S(\widetilde{p}(f[\mathtt{a}](\bar{\mathtt{u}})))) \\
\overset{\text{Def }\mathbf{T}^{p}_{S}}{\Leftrightarrow}\quad & \mathtt{w} \in \mathbf{T}^{q}_{t}(\mathbf{T}^{p}_{S}(f[\mathtt{a}](\bar{\mathtt{u}}))).
\end{aligned}
$$

Step $(\star)$ uses Lemma 4.4.2. It holds only when S is single-valued or T is linear. Otherwise, only "$\Leftarrow$" holds. □

5. EVALUATION

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

5.1. HTML Sanitization

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

This has led to the emergence of a range of libraries attempting to do HTML sanitization, including PHP Input Filter, HTML_Safe, kses, htmLawed, Safe

HTML Checker, and HTML Purifier. Among these, the last one, HTML Purifier (`http://htmlpurifier.org`), is believed to be most robust, so we choose it as a comparison point for our experiments. Note that HTML Purifier is a tree-based rewriter written in PHP, which uses the HTMLTidy library to parse the input.

We show how FAST is expressive enough to model HTML sanitizers, and we argue that writing such programs is easier with FAST than with current tools. Our version of an HTML sanitizer written in FAST and automatically translated by the FAST compiler into C# is partially described in Section 2. Although we cannot argue for the correctness of our implementation (except for the basic analysis shown in Section 2), sanitizers are much simpler to write in FAST thanks to composition. In all the libraries mentioned previously HTML sanitization is implemented as a monolithic function in order to achieve reasonable performance. In the case of FAST each sanitization routine can be written as a single function and all such routines can be then composed preserving the property of traversing the input HTML only once.

Evaluation: To compare different sanitization strategies in terms of performance, we chose 10 Web sites and picked an HTML page from each content, ranging from 20KB (Bing) to 409KB in size (Facebook). For speed, the FAST-based sanitizer is comparable to HTML Purify. In terms of maintainability, FAST wins on two counts. First, unlike sanitizers written in PHP, FAST programs can be analyzed statically. Second, our sanitizer is only 200 lines of FAST code instead of 10,000 lines of PHP. While these are different languages, we argue that our approach is more maintainable because FAST captures the high-level semantics of HTML sanitization, as well as being fewer lines of code to understand. We manually spot-checked the outputs to determine that both produce reasonable sanitizations.

5.2. Conflicting AR Applications

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

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

composition. We compute p, composition of p_1 and p_2.

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

output restriction. We compute p'', a restriction of p' that only outputs trees in which some node contains two tags.

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

Evaluation: Figure 6 shows the timing results for conflict analysis. To collect this data, we randomly generated 100 taggers in FAST and checked whether they conflicted with each other. Each element in the input of a tagger contained a name of type string,

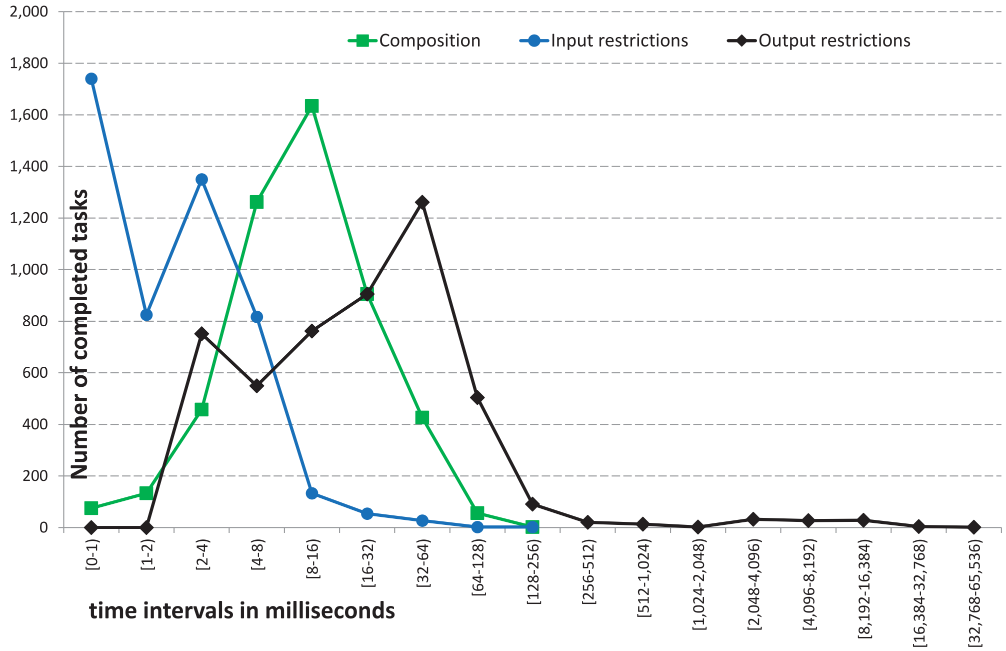

Fig. 6. Augmented reality: running times for operations on transducers. The x axis represents time intervals in ms. The y axis shows how many cases run in a time belonging to an interval. For example, about 1,600 compositions were completed between 8 and 16 ms.

two attributes of type real, and an attribute of type int. In our encoding the left child of each element was the list of tags, while the right child was the next element. Each tagger we generated conforms to the following properties: (1) it is nonempty; (2) it tags on average three nodes; and (3) it tags each node at most once.

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

All the compositions are computed in less than 250ms, and the average time is 15ms. All the input restrictions are computed in less than 150ms. The average time is 3.5ms. All the output restrictions are computed in less than 33,000ms. The average time is 175ms. The output restriction takes longer to compute in some cases, due to the following two factors: (1) the input sizes are always bigger: the size of the composed transducers after the input restriction (p' in the list before) vary from 5 to 300 states and 10 to 4,000 rules. This causes the restricted output to have up to 5,000 states and 100,000 rules; and (2) since the conditions in the example are randomly generated, some of them may be complex causing the SMT solver to slow down the computation. The 33,000ms example contains nonlinear (cubic) constraints over reals. The average time of 193ms per pairwise conflict check is quite acceptable: indeed, adding a new app to a store already containing 10,000 apps will incur an average checking overhead of about 35min.

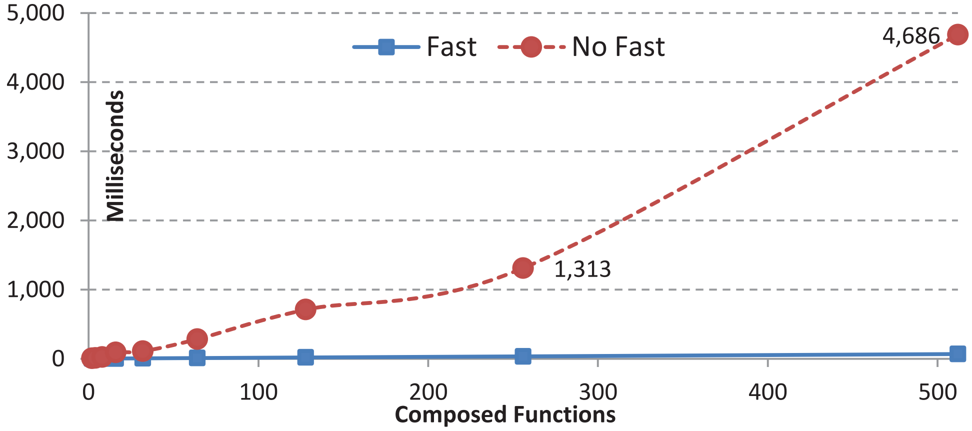

Fig. 7. Deforestation advantage for a list of 4,096 integers.

5.3. Deforestation

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

Evaluation: We considered the function *map_caesar* from Figure 8 that replaces each value x of an integer list with $(x + 5)\%26$. We composed the function *map_caesar* with itself several times to see how the performance changed when using FAST. Let us call map^n the composition of *map_caesar* with itself n times. Unlike in Wadler [1988], we do not represent numbers using their unary encoding. We run the experiments on lists containing randomly generated elements and we consider up to 512 composed functions. Figure 7 shows the running time of FAST with and without deforestation for a list of 4,096 integers used as the input. The running time of the version that uses transducer composition is almost unchanged, even for 512 compositions, while the running time of the naively composed functions degrades linearly in the number of composed functions. This is due to the fact that the composed version results in a single function that processes the input list in a single left-to-right pass, while the naive composition causes the input list to be read multiple times.

5.4. Analysis of Functional Programs

FAST can also be used to perform static analysis of simple functional programs over lists and trees. Consider again the functions from Figure 8. As we described in the previous experiment the function *map_caesar* replaces each value x of an integer list with $(x + 5)$ *mod* 26. The function *filter_ev* removes all the odd elements from a list.

One might wonder what happens when such functions are composed. Consider the case in which we execute the map followed by the filter, followed by the map, and

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

Fig. 8. Analysis of functional programs in FAST. The final assertion shows that *comp*2 never outputs a nonempty list. Example available at `http://rise4fun.com/Fast/Jv`.

again by the filter. This transformation is equivalent to deleting all the elements in the list! This property can be statically checked in FAST. We first compute *comp*2 as the composition described previously. As shown in Figure 8, the language of nonempty lists can be expressed using the construct *not_emp_list*. Finally, we can use the output restriction to restrict *comp*2 to only output nonempty lists and show that such function is empty. In this example the whole analysis can be done in less than 10ms.

5.5. CSS Analysis

Cascading Style-Sheets (CSS) is a language that allows one to stylize and format HTML documents. A CSS program is a sequence of CSS rules, where each rule contains a selector and an assignment. The selector decides which nodes are affected by the rule and the assignment is responsible for updating the selected nodes. The following is a typical CSS rule: `div p {word-spacing:30px;}`. In this case `div p` is the selector, while `word-spacing:30px` is the assignment. This rule sets the attribute `word-spacing` to `30px` for every `p` node inside a `div` node. We call $C(H)$ be the updated HTML resulting from applying a CSS program C to an HTML document H. In Geneves et al. [2012] CSS programs are analyzed using tree logic. For example, one can check whether given a CSS program C, there does not exist an HTML document H such that $C(H)$ contains a node *n* for which the attributes `color` and `background-color` both have value `black`. This property ensures that black text is never written on a black background, causing the text not to be readable. Ideally, one would want to check that `color` and `background-color` never have the same value, but, since tree logic explicitly models the alphabet, the corresponding formula would be too large. By modeling CSS programs as symbolic tree transducers, we can overcome this limitation. This analysis relies on the alphabet being symbolic, and we plan on extending FAST with primitives for simplifying CSS modeling.

6. A COMPARISON WITH CLASSIC TREE TRANSDUCERS

As we mentioned in the previous section, the HTML sanitization and CSS analysis problems could, in principle, be expressed using existing classic models and do not require symbolic alphabets. In both of these domains the alphabet is finite and, for example, the sanitizer in Figure 2 can be represented by classic finite state transducers with regular look-ahead. In the next paragraphs we show the benefit of the symbolic representation of the alphabet and argue that the use of classic transducers does not scale in this case.

The HTML sanitization example illustrates some core differences between the symbolic and the classic case. In some respect, the situation is analogous to going from SAT to SMT solving [de Moura and Bjørner 2011], where many of the core propositional techniques remain similar but where a theory specific component adds additional succinctness and expressiveness. Consider our encoding of HTML documents presented in Figure 3. In our representation each string value is modeled as a list of characters, and this means that each possible character should belong to the input alphabet. The input alphabet Σ therefore needs to include the UTF16 representation of Unicode characters, because UTF16 is used as the standard runtime representation of characters and is the basic building block of strings. Thus, Σ has at least 2^{16} elements, for example, as unary function symbols f_c for the characters c. If we want to support full Unicode, for example, including emoticons [The Unicode Consortium], we need to add additional rules that ensure that consecutive characters $\ldots(f_c(f_d(\ldots)))$ where c and d are *surrogates* are indeed valid as surrogate pairs. This adds yet another layer of complexity and there are 2^{20} valid surrogate pairs. In contrast, at the level of strings, that are defined as lists of 16-bit bit vectors, such checks are straightforward (given a solver that supports lists and bit-vector arithmetic; for example, Z3 [de Moura and Bjørner 2008]), and involve fairly simple arithmetic operations.

We need to add look-ahead automata to all the rules so that the *tag* subtree does not include other symbols besides the character symbols. Such an automaton needs 2^{16} transitions. The where condition $tag =$ "script" can be represented by a look-ahead automaton, say A, with six transitions. The constraint $tag \neq$ "script" can be represented by the *complement* A^c of A. Observe that complementation of classic automata over large alphabets is expensive: while A needs six rules, one per character in the string "script", A^c needs $6 * (2^{16} - 1)$ rules. The other string constraints are handled similarly. Besides the additional look-ahead tests, transformation rules remain the same, where *tag* is treated as the first subtree. Observe that, a further blowup would occur if we wanted to apply transformations (other than the identity mapping, such as HtmlEncoding) to *tag*, in which case we would need explicit rules for all of the 2^{16} symbols.

The bottom line is that tags are independent of the rest of the tree structure and the two should, if possible, not be mixed. Similar arguments already hold for symbolic finite (word) transducers as a special case of symbolic tree transducers, where a symbolic representation may avoid an exponential blow-up compared to an equivalent classic transducer, as demonstrated by the symbolic word transducer implementing UTF8 encoding in D'Antoni and Veanes [2013b]. The same argument holds for the domain of CSS analysis.

7. RELATED WORK

Tree transducers. Tree transducers have been long studied; surveys and books are available on the topic [Fülöp and Vogler 1998; Comon et al. 2007; Raoult 1992]. The first models were top-down and bottom-up tree transducers [Engelfriet 1975; Baker 1979], later extended to top-down transducers with regular look-ahead in order to achieve closure under composition [Engelfriet 1977; Fülöp and Vágvölgyi 1989; Engelfriet 1980]. Extended top-down tree transducers [Maletti et al. 2009] (XTOP) were introduced in the context of program inversion and allow rules to read more than one node at a time, as long as such nodes are adjacent. When adding look-ahead such a model is equivalent to top-down tree transducers with regular look-ahead. More complex models, such as *macro tree transducers* [Engelfriet and Vogler 1985], and *streaming tree transducers* [Alur and D'Antoni 2012] have been introduced to improve the expressiveness at the cost of higher complexity. Due to this reason we do not consider extending them in this article.

Symbolic transducers. Symbolic finite transducers (SFTs) over lists, together with a front-end language Bek, were originally introduced in Hooimeijer et al. [2011] with a focus on security analysis of string sanitizers. The main SFT algorithms, in particular an algorithm for deciding equivalence of SFTs modulo a decidable background theory, are studied in Veanes et al. [2012]. Variants of SFTs in which multiple input symbols can be read by a single transition are studied in D'Antoni and Veanes [2013a] and in Botinčan and Babić [2013]. Symbolic tree transducers are originally introduced in Veanes and Bjørner [2012], where it is wrongly claimed that STTs are closed under composition by referring to a generalization of a proof of the classic case in Fülöp and Vogler [1998] that is only stated for total deterministic finite tree transducers. In Fülöp and Vogler [2014] this error is discovered and other properties of STTs are investigated. The main result of Veanes and Bjørner [2012] is an algorithm for checking equivalence of single-valued linear STTs. For classic transducers, equivalence has been shown to be decidable for deterministic or finite-valued tree transducers [Seidl 1994a], streaming tree transducers [Alur and D'Antoni 2012], and MSO tree transformations [Engelfriet and Maneth 2006]. We are currently investigating the problem of checking equivalence of single-valued STTRs.

DSL for tree manipulation. Domain specific languages for tree transformation have been studied in several different contexts. VATA [Lengal et al. 2012] is a tree automata library for analyzing tree languages over large alphabets. In VATA transitions are represented symbolically using BDDs; however, the library does not support transducers and it is limited to nondeterministic automata over finite (although large) alphabets. TTT [Purtee and Schubert 2012] and Tiburon [May and Knight 2008], are transducer-based languages used in natural language processing. TTT allows complex forms of pattern matching, but does not enable any form of analysis. Tiburon supports probabilistic transitions and several weighted tree transducer algorithms. Although they support weighted transitions, both the languages are limited to finite input and output alphabets. ASF+SDF [van den Brand et al. 2002] is a term-rewriting language for manipulating parsing trees. ASF+SDF is simple and efficient, but does not support any analysis. In the context of XML processing numerous languages have been proposed for querying (XPath, XQuery [Walmsley 2007]), stream processing (STX [Becker 2003]), and manipulating (XSLT, XDuce [Hosoya and Pierce 2003]) XML trees. While being very expressive, these languages support very limited forms of analysis. Emptiness has been shown decidable for restricted fragments of XPath [Bojańczyk et al. 2006]. XDuce [Hosoya and Pierce 2003] allows one to define basic XML transformations, and supports a tree automata based type checking that is limited to finite alphabets. We plan to extend Fast to better handle XML processing and to identify a fragment of XPath expressible in Fast. However, to the best of our knowledge, Fast is the first language for tree manipulations that supports infinite input and output alphabets while preserving decidable analysis. Figure 9 summarizes the relations between Fast and the other domain-specific languages for tree transformations.

Applications. The connection between tree transducers and deforestation was first investigated in Wadler [1988], and then further investigated in Kühnemann [1999]. In this setting, deforestation is done via *Macro Tree Transducers* (MTTs) [Engelfriet and Vogler 1985]. While being more expressive than Top-Down Transducers with regular look-ahead, MTTs only support finite alphabets and their composition algorithm has very high complexity. We are not aware of an actual implementation of the techniques in Kühnemann [1999]. Many models of tree transducers have been introduced to

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

Fig. 9. Summary of main domain specific languages for tree-manipulating programs and their properties; σ indicates whether the language supports finite (ff) or infinite (∞) alphabets.

analyze and execute XML transformations. Most notably, K-pebble transducers [Milo et al. 2000] enjoy decidable type checking and can capture fairly complex XSLT and XML transformations. Macro forest transducer [Perst and Seidl 2004] extend MTT to operate over unranked trees and therefore naturally capture XML transformations. Recently, this model has been used to efficiently execute XQuery transformations [Hakuta et al. 2014]. The models we just discussed only operate over finite alphabets. Many models of automata and transducers have been applied to verifying functional programs. The equivalence problems has been shown to be decidable for some fragments of ML using Class Memory Automata [Cotton-Barratt et al. 2015]. This model allows values over infinite alphabets to be compared using equality, but does not support predicates arbitrary label theories. This restriction is common in the so-called data languages and makes other models operating in this setting orthogonal to symbolic automata and transducers. Higher-Order Multiparameter Tree Transducers (HMTTs) [Kobayashi et al. 2010] are used for type checking higher-order functional programs. HMTTs enable sound but incomplete analysis of programs which takes multiple trees as input, but only support finite alphabets. Extending our theory to multiple input trees and higher-order functions is an open research direction.

Open problems. Several complexity related questions for STAs and STTRs are open and depend on the complexity of the label theory, but some lower bounds can be established using known results for finite tree automata and transducers. For example, an STA may be exponentially more succinct than the equivalent *normalized* STA because one can directly express the intersection nonemptiness problem of a set of normalized STAs as the emptiness of a single unnormalized STA. In the classic case, the nonemptiness problem of tree automata is P-CO, while the intersection nonemptiness problem is EXPTIME-CO [Comon et al. 2007, Theorem 1.7.5]. Recently, new techniques based on antichains have been proposed to check universality and inclusion for nondeterministic tree automata [Bouajjani et al. 2008]. Whether such techniques translate to our setting is an open research direction. Concrete open problems are decidability of *single-valuedness of STTRs*, *equivalence of single-valued STTRs*, and *finite-valuedness of STTRs*. Classically, these problems are decidable, but some proofs are mathematically quite challenging [Seidl 1994a]. Novel algorithms for minimizing and learning symbolic automata over lists have been recently proposed in D'Antoni and Veanes [2014] and Botinčan and Babić [2013]. Extending such results to STAs are also unexplored topics.

8. CONCLUSIONS

We introduce FAST, a new domain-specific language for tree manipulation based on symbolic tree automata and symbolic tree transducers. To allow FAST to perform useful program analysis, we design a novel algorithm for composing symbolic tree transducers with regular look-ahead and we prove its correctness. FAST strikes a delicate balance between precise analysis and expressiveness, and we show how multiple applications benefit from this analysis. A running version of FAST can be accessed at http://rise4fun.com/Fast/.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback that helped us improve the quality of our article.

REFERENCES

Rajeev Alur and Loris D'Antoni. 2012. Streaming tree transducers. In *Automata, Languages, and Programming*, Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer (Eds.). Lecture Notes in Computer Science, Vol. 7392. Springer, Berlin, 42–53. DOI:http://dx.doi.org/10.1007/978-3-642-31585-5_8

Brenda S. Baker. 1979. Composition of top-down and bottom-up tree transductions. *Inform. Control* 41, 2 (1979), 186–213.

Oliver Becker. 2003. Streaming transformations for XML-STX. In *XMIDX 2003 (LNI)*, Rainer Eckstein and Robert Tolksdorf (Eds.), Vol. 24. GI, 83–88.

Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2006. Two-variable logic on data trees and XML reasoning. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'06)*. ACM, New York, NY, 10–19. DOI:http://dx.doi.org/10.1145/1142351.1142354

Matko Botinčan and Domagoj Babić. 2013. Sigma*: Symbolic learning of input-output specifications. *SIGPLAN Not.* 48, 1 (Jan. 2013), 443–456. DOI:http://dx.doi.org/10.1145/2480359.2429123

Ahmed Bouajjani, Peter Habermehl, Lukas Holik, Tayssir Touili, and Tomas Vojnar. 2008. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA'08*. Lecture Notes in Computer Science, Vol. 5148. Springer, Berlin, 57–67. DOI:http://dx.doi.org/10.1007/978-3-540-70844-5_7

Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. 2007. Tree Automata Techniques and Applications.

Conrad Cotton-Barratt, David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. 2015. Fragments of ML decidable by nested data class memory automata. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015. Proceedings*. 249–263. DOI:http://dx.doi.org/10.1007/978-3-662-46678-0_16

Loris D'Antoni and Margus Veanes. 2013a. Equivalence of extended symbolic finite transducers. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Springer-Verlag, Berlin, 624–639. DOI:http://dx.doi.org/10.1007/978-3-642-39799-8_41

Loris D'Antoni and Margus Veanes. 2013b. Static analysis of string encoders and decoders. In *VMCAI 2013 (LNCS)*, R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.), Vol. 7737. Springer, 209–228.

Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In *POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 541–554.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction* *and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Leonardo de Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: Introduction & applications. *Commun. ACM* 54, 9 (2011), 69–77.

Joost Engelfriet. 1975. Bottom-up and top-down tree transformations—A comparison. *Math. Syst. Theory* 9 (1975), 198–231.

Joost Engelfriet. 1977. Top-down tree transducers with regular look-ahead. *Math. Syst. Theory* 10 (1977), 289–303.

Joost Engelfriet. 1980. Some open questions and recent results on tree transducers and tree languages. In *Formal Language Theory*. Academic Press, 241–286.

Joost Engelfriet and Sebastian Maneth. 2006. The equivalence problem for deterministic MSO tree transducers is decidable. *Inf. Process. Lett.* 100, 5 (Dec. 2006), 206–212. DOI:http://dx.doi.org/10.1016/j.ipl.2006.05.015

Joost Engelfriet and Heiko Vogler. 1985. Macro tree transducers. *J. Comput. Syst. Sci.* 31 (1985), 71–146.

Z. Esik. 1980. Decidability results concerning tree transducers. *Acta Cybern.* 5 (1980), 1–20.

Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. 1991. Logic programs as types for logic programs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91)*. 300–309. DOI:http://dx.doi.org/10.1109/LICS.1991.151654

Zoltán Fülöp and Sándor Vágvölgyi. 1989. Variants of top-down tree transducers with look-ahead. *Math. Syst. Theory* 21, 3 (1989), 125–145.

Zoltán Fülöp and Heiko Vogler. 2014. Forward and backward application of symbolic tree transducers. *Acta Inf.* 51, 5 (Aug. 2014), 297–325. DOI:http://dx.doi.org/10.1007/s00236-014-0197-7

Zoltán Fülöp and H. Vogler. 1998. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer.

Pierre Geneves, Nabil Layaida, and Vincent Quint. 2012. On the analysis of cascading style sheets. In *WWW'12*. ACM, New York, NY, 809–818. DOI:http://dx.doi.org/10.1145/2187836.2187946

Shizuya Hakuta, Sebastian Maneth, Keisuke Nakano, and Hideya Iwasaki. 2014. XQuery streaming by forest transducers. In *IEEE 30th International Conference on Data Engineering (ICDE'14)*. 952–963. DOI:http://dx.doi.org/10.1109/ICDE.2014.6816714

Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. 2003. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, 1–1.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc.

Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.* 3, 2 (May 2003), 117–148. DOI:http://dx.doi.org/10.1145/767193.767195

Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. *SIGPLAN Not.* 45, 1 (Jan. 2010), 495–508. DOI:http://dx.doi.org/10.1145/1707801.1706355

Armin Kühnemann. 1999. Comparison of deforestation techniques for functional programs and for tree transducers. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*.

Ond Lengal, Ji Simáček, and Tom Vojnar. 2012. VATA: A library for efficient manipulation of nondeterministic tree automata. In *TACAS'12*. Lecture Notes in Computer Science, Vol. 7214. Springer, Berlin, 79–94. DOI:http://dx.doi.org/10.1007/978-3-642-28756-5_7

Andreas Maletti, Jonathan Graehl, Mark Hopkins, and Kevin Knight. 2009. The power of extended top-down tree transducers. *SIAM J. Comput.* 39, 2 (June 2009), 410–430.

Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. 2005. XML type checking with macro tree transducers. In *PODS'05*. ACM, New York, NY, 283–294. DOI:http://dx.doi.org/10.1145/1065167.1065203

Jonathan May and Kevin Knight. 2008. A Primer on Tree Automata Software for Natural Language Processing. http://www.isi.edu/licensed-sw/tiburon/tiburon-tutorial.pdf.

Tova Milo, Dan Suciu, and Victor Vianu. 2000. Typechecking for XML transformers. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS'00)*. ACM, 11–22.

Thomas Perst and Helmut Seidl. 2004. Macro forest transducers. *Inf. Process. Lett.* 89, 3 (Feb. 2004), 141–149. DOI:http://dx.doi.org/10.1016/j.ipl.2003.05.001

Adam Purtee and Lenhart Schubert. 2012. TTT: A tree transduction language for syntactic and semantic processing. In *Proceedings of the Workshop on Application of Tree Automata Techniques in NLP*.

Jean-Claude Raoult. 1992. A survey of tree transductions. In *Tree Automata and Languages*. , 311–326.

Helmut Seidl. 1994a. Equivalence of finite-valued tree transducers is decidable. *Math. Syst. Theory* 27 (1994), 285–346.

Helmut Seidl. 1994b. Haskell overloading is DEXPTIME-complete. *Inf. Process. Lett.* 52, 2 (1994), 57–60.

The Unicode Consortium. *The Unicode Standard 6.3, Emoticons*. The Unicode Consortium. http://unicode.org/charts/PDF/U1F600.pdf.

Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. 2002. Compiling language definitions: The ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.* 24, 4 (July 2002), 334–368. DOI:http://dx.doi.org/10.1145/567097.567099

Margus Veanes and Nikolaj Bjørner. 2012. Symbolic tree transducers. In *Proceedings of the 8th International Conference on Perspectives of System Informatics (PS'11)*. Springer-Verlag, Berlin, 377–393. DOI:http://dx.doi.org/10.1007/978-3-642-29709-0_32

Margus Veanes and Nikolaj Bjørner. 2015. Symbolic tree automata. *Inform. Process. Lett.* 115, 3 (2015), 418–424. DOI:http://dx.doi.org/10.1016/j.ipl.2014.11.005

Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. 2012. Symbolic finite state transducers: Algorithms and applications. *SIGPLAN Not.* 47, 1 (Jan. 2012), 137–150. DOI:http://dx.doi.org/10.1145/2103621.2103674

Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.* 73, 2 (Jan. 1988), 231–248. DOI:http://dx.doi.org/10.1016/0304-3975(90)90147-A

Priscilla Walmsley. 2007. *XQuery*. O'Reilly Media, Inc.

Received July 2014; accepted June 2015