Fast: a Transducer-Based Language for Tree Manipulation

LORIS D’ANTONI, University of Pennsylvania
MARGUS VEADES, Microsoft Research
BENJAMIN LIVSHITS, Microsoft Research
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 above 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:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

This paper introduces 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 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, 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 [Veanees and Bjorner 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 \textit{symbolic tree transducer} (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], that 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 \textit{symbolic tree transducers with regular look-ahead} (STTRs). The main theoretical result of this paper 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 \textsc{Fast} as a frontend for STAs and STTRs. \textsc{Fast} (Functional Abstraction of Symbolic Transducers) 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 non-linear real arithmetic. We use \textsc{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 \textsc{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 \textsc{Fast} with previous DSLs for tree manipulation.

\textbf{Contributions summary:}

(1) a \textit{theory of symbolic tree transducers with regular look-ahead} (STTR), that non-trivially extends the classic theory of tree transducers (§3);
(2) a new algorithm for composing STTRs together with a proof of correctness (§4);
(3) \textsc{Fast}, a domain-specific language for tree manipulations founded on the theory of STTRs (§3); and
(4) five concrete applications of \textsc{Fast} showing how composition of STTR can be beneficial in practical settings (§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 DOM (Document Object Model) tree, 2) the DOM is modified by a sequence of sanitization functions \( f_1, \ldots, f_n \), and 3) the modified DOM tree is transformed back into an HTML document\(^1\). In the following paragraphs we use FAST to describe some of the functions used during step 2. Each function \( f_i \) takes as input a DOM tree received from the browser’s parser and transforms it into an updated DOM tree. As an example, the FAST program \texttt{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 "\texttt{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 datatype \texttt{HtmlE} of our trees.\(^2\) Each node of type \texttt{HtmlE} contains an attribute \texttt{tag} of type \texttt{string} and is built using one of the constructors \texttt{nil}, \texttt{val}, \texttt{attr}, or \texttt{node}. Each constructor has a number of children associated with it (2 for \texttt{attr}) and all such children are \texttt{HtmlE} nodes. We use the type \texttt{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 \texttt{HtmlE} element \texttt{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) \texttt{tag} contains the node type of \( n \) (\texttt{div}, etc.). Each HTML attribute \( a \) with value \( s \) is encoded as an \texttt{HtmlE} element \texttt{attr}(\( x_1, x_2 \)) with two children \( x_1, x_2 \) where: 1) \( x_1 \) encodes the value \( s \) (\texttt{nil} if \( s \) is the empty string), 2) \( x_2 \) encodes the list of attributes following \( a \) (\texttt{nil} if \( a \) is the last attribute), and 3) \texttt{tag} contains the name of \( a \) (\texttt{id}, etc.). Each non-empty string \( w = s_1 \ldots s_n \) is encoded as an \texttt{HtmlE} element \texttt{val}(\( x_1 \)) where \texttt{tag} contains the string "\texttt{s}_1\"", and \( x_1 \) encodes the suffix \texttt{s}_2 \ldots \texttt{s}_n. Each element \texttt{nil} has \texttt{tag} "\texttt{"}, 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, \texttt{nodeTree} (lines 4-6) is the language of correct HTML encodings (nodes): 1) the tree \texttt{node}(\( x_1, x_2, x_3 \)) is in the language \texttt{nodeTree} if \( x_1 \) is in the language \texttt{attrTree}, \( x_2 \) is in the language \texttt{nodeTree}, and \( x_3 \) is in the language \texttt{nodeTree}; 2) the tree \texttt{nil} is in \texttt{nodeTree} if its \texttt{tag} contains the empty string. The other language definitions are similar.

We now describe the sanitization functions. The transformation \texttt{remScript} (lines 14-18) takes an input tree \( t \) of type \texttt{HtmlE} and produces an output tree of type \texttt{HtmlE}: 1) if \( t = \texttt{node}(x_1, x_2, x_3) \) and its \texttt{tag} is different from "\texttt{script}"", \texttt{remScript} outputs a copy of \( t \) in which \( x_2 \) and \( x_3 \) are replaced by the results of invoking \texttt{remScript} on \( x_2 \) and \( x_3 \) respectively; 2) if \( t = \texttt{node}(x_1, x_2, x_3) \) and its \texttt{tag} is equal to "\texttt{script}"", \texttt{remScript} outputs a copy of \( x_3 \), 3) if \( t = \texttt{nil} \), \texttt{remScript} outputs a copy \( t \). The transformation \texttt{esc} (lines 19-24) of type \texttt{HtmlE} \texttt{->} \texttt{HtmlE} escapes the characters '"' and "", and it outputs a copy of the input tree in which each node \texttt{val} with tag '"' or "" is pre-pended a node \texttt{val} with tag "\"". The transformations \texttt{remScript} and \texttt{esc} are then composed into a single transformation \texttt{rem_esc} (line 26). This is done using transducer composition. The square bracket syntax is used to represent the assignments to the attribute \texttt{tag}. One might notice that \texttt{rem_esc}
Fig. 2. Implementation and analysis of an HTML sanitizer in FAST.

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 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". Next, using pre-image computation, we compute the language bad_inputs
(line 34) of inputs that produce a bad output. Finally, if bad_inputs is the empty language,
sani never produces bad outputs. When running this program in FAST this checking (line 35)
fails, and FAST provides the following counterexample:

\[
\text{node } ["\text{script}"] \text{ nil nil } (\text{node } ["\text{script}"] \text{ nil nil nil})
\]

\footnote{This definition illustrates the nondeterministic semantics of FAST: a tree }
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 rem\texttt{Script} on \(x_3\). After fixing this bug
the assertion becomes valid. In this example, we showed how in \texttt{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

3. SYMBOLIC TREE TRANSDUCERS AND \texttt{FAST}

The concrete syntax of \texttt{Fast} is shown in Figure 4. \texttt{Fast} is designed for describing trees,
tree languages and functions from trees to trees. These are supported using
symbolic tree automata (STAs), and symbolic tree transducers with regular look-ahead (STTRs). This
section covers these objects and how they describe the semantics of \texttt{Fast}.

3.1. Background

All definitions are parametric with respect to a given background theory, called a \textit{label theory},
over a fixed background structure with a recursively enumerable universe of elements. Such
a theory is allowed to support auxiliary 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

We use \(\lambda\)-expressions for defining anonymous functions called \textit{\(\lambda\)-terms} without having to
name them explicitly. In general, we use standard first-order logic and follow the notational
conventions that are consistent with \cite{Veanes et al. 2012}. We write \(f(v)\) for the functional
application of the \(\lambda\)-term \(f\) to the term \(v\). We write \(\sigma\) for a type and the universe of elements
of type \(\sigma\) is denoted by \(\sigma\). A \(\sigma\)-predicate is a \(\lambda\)-term \(\lambda x.\varphi(x)\) where \(x\) has type \(\sigma\), and \(\varphi\)
is a formula for which the free variables \(FV(\varphi)\) are contained in \(\{x\}\). Given a \(\sigma\)-predicate \(\varphi\),
\([\varphi]\) denotes the set of all \(a \in \sigma\) such that \(\varphi(a)\) holds. The set of \(\sigma\)-predicates is denoted by
\(\Psi(\sigma)\). Given a type \(\sigma\) (such as \texttt{int}), we extend the universe with \(\sigma\)-labeled finite trees as an
algebraic datatype \(T_\Sigma^\sigma\) where \(\Sigma\) is a finite set of \textit{tree constructors} \(f\) with \textit{rank} \(\zeta(f) \geq 0\): \(f\)
has type \(\sigma \times (T_\Sigma^\sigma)^{\zeta(f)} \to T_\Sigma^\sigma\). We call \(T_\Sigma^\sigma\) a \textit{tree type}. Let \(\Sigma(k) \overset{\text{def}}{=} \{f \in \Sigma \mid \zeta(f) = k\}\). We
require that \(\Sigma(0)\) is non-empty so that \(T_\Sigma^\sigma\) is non-empty. We write \(f[a](\vec{u})\) for \(f(a, \vec{u})\)
and abbreviate \(f[a]()\) by \(f[a]\).

\footnote{When \(\zeta(f) = 0\) then \(f\) has type \(\sigma \to T_\Sigma^\sigma\).}
Fig. 4. Concrete syntax of FAST. Nonterminals and meta-symbols are in italic. Constant expressions for strings and numbers use C# syntax [Heijlsberg et al. 2003]. Additional well-formedness conditions (such as well-typed terms) are assumed to hold.

Example 3.1. The FAST program in Figure 2, declares $\text{Html}E = T_{\Sigma}^{\text{String}}$ over $\Sigma = \{\text{nil}, \text{val}, \text{attr}, \text{node}\}$, where $\zeta(\text{nil}) = 0$, $\zeta(\text{val}) = 1$, $\zeta(\text{attr}) = 2$, and $\zeta(\text{node}) = 3$. For example $\text{attr}[\text{a}][\text{b}][\text{c}]$ is in $T_{\Sigma}^{\text{String}}$.

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 $\bar{X} \sqcup \bar{Y} \equiv (X_i \cup Y_i)_{i=1}^k$. If $\bar{X}$ is a $k$-tuple of sets, $j \in \{1, \ldots, k\}$ and $Y$ is a set then $(\bar{X} \sqcup_j Y)$ is the $k$-tuple $(\text{if} i = j \text{ then } X_i \text{ or } Y \text{ else } X_i)_{i=1}^k$.

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 [Veane and Björner 2012]. We decide to use alternating STAs instead of their non-alternating 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 $(Q, T_\Sigma^\tau, \delta)$, where $Q$ is a finite set of states, $T_\Sigma^\tau$ is a tree type, and $\delta \subseteq \bigcup_{k \geq 0} (Q \times \Sigma(k) \times \Psi(\sigma) \times (2^{|\cdot|})^{k})$ is a finite set of rules $(q, f, \varphi, \ell)$, where $q$ is the source state, $f$ the symbol, $\varphi$ the guard, and $\ell$ the look-ahead.
Next, we define the semantics of an STA $A = (Q, T^A, \delta)$.

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

$$L^A_q \overset{\text{def}}{=} \{ f[a](\bar{t}) \in T^A_q \mid (q, f, \varphi, \bar{t}) \in \delta, a \in \{\varphi\}, \bigwedge_{i=1}^{z(f)} t_i \in L^A_{q_i} \}$$

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

$$L^A_q \overset{\text{def}}{=} \begin{cases} \bigcap_{q \in q} L^A_{q}, & \text{if } q \neq \emptyset; \\ T^A_q, & \text{otherwise.} \end{cases}$$

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

$$\text{lang } q : \tau \{ c(y) \text{ where } \varphi(x) \text{ given } \bar{t}(y) \mid \ldots \}$$

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

**Example 3.2.** Consider the following FAST program.

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

An equivalent STA $A$ over $T^\text{bst}$ has states $\{o, p, q\}$ and rules

$$\{ (p, \text{L}, \lambda x. x > 0, ()), (p, \text{N}, \lambda x. \text{true}, (\{p\}, \{p\})),
(0, \text{L}, \lambda x. \text{odd}(x), ()), (0, \text{N}, \lambda x. \text{true}, (\{o\}, \{o\})),
(q, \text{N}, \lambda x. \text{true}, (\emptyset, \{p, o\})) \}.$$

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 $\text{L}$, so there is no rule.

In the following we say STA for alternating STA.$^5$

**Definition 3.** $A$ is normalized if for all $(p, f, \varphi, \bar{t}) \in \delta$, and all $i$, $1 \leq i \leq z(f)$, $\ell_i$ 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.

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

$$\delta^f(\emptyset) = \{(\emptyset, f, \emptyset, (\emptyset)^{z(f)})\}$$
$$\delta^f(p \cup q) = \{ r \land s \mid r \in \delta^f(p), s \in \delta^f(q) \}$$
$$\delta^f(p) = \{(p, f, \varphi, \bar{p}) \mid (p, f, \varphi, \bar{p}) \in \delta \}$$

where merge operation $\land$ over merged rules is defined as follows:

$$(p, f, \varphi, \bar{p}) \land (q, f, \psi, \bar{q}) \overset{\text{def}}{=} (p \cup q, f, \varphi \cup \psi, p \cup q)$$

$^5$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.

Theorem 3.3. For all \( q \subseteq Q \), \( L_A^q = L_{\mathcal{N}(A)}^q \).

Proof. The case when \( q = \emptyset \) is clear because the state \( 0 \) in \( \mathcal{N}(A) \) has the same semantics as \( L_A^0 \). Assume \( q \neq \emptyset \). We show (1) for all \( t \in T_A^2 \):

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

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

\[
f[a] \in L_A^q \iff \forall q \in q(f[a] \in L_A^q)
\]

\[
\iff \forall q \in q(\exists \varphi((q, f, \varphi, ()), a) \in \delta, a \in [\varphi])
\]

\[
\iff \forall q \in q(\exists \varphi((q, f, \varphi, ()), a) \in \delta([q]), a \in [\varphi])
\]

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

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

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

\[
\iff \bigwedge_{i=1}^2 \exists \varphi_i, p_1^i, p_2^i : (q_1, f, \varphi_i, (p_1^i, p_2^i)) \in \delta_A, a \in [\varphi_i], t_1 \in L_A^{p_1^i}, t_2 \in L_A^{p_2^i}
\]

\[
\iff \bigwedge_{i=1}^2 \exists \varphi_i, p_1^i, p_2^i : (q_1, f, \varphi_i, (p_1^i, p_2^i)) \in \delta([q]), a \in [\varphi_i], t_1 \in L_{\mathcal{N}(A)}^{p_1^i}, t_2 \in L_{\mathcal{N}(A)}^{p_2^i}
\]

\[
\iff \exists \varphi, p_1, p_2 : t_1 \in L_A^{p_1^i} \land L_A^{p_2^i}, t_2 \in L_A^{p_1^i} \land L_A^{p_2^i}
\]

\[
(t, q, f, \varphi, ((p_1^i \cup p_2^i), p_1^i \cup p_2^i)) \in \delta([\varphi], a \in [\varphi])
\]

\[
\iff \exists \varphi, p_1, p_2 : t_1 \in L_A^{p_1^i} \setminus L_A^{p_1^i}, t_2 \in L_A^{p_2^i} \setminus L_A^{p_2^i}, (q, f, \varphi, ((p_1^i \setminus \{ p_2^i \}), \{ p_2^i \})) \in \delta(A, a \in [\varphi])
\]

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

The theorem follows by the induction principle. □

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

\footnote{In practice, merged rules are computed lazily starting from the initial state. Merged rules with unsatisfiable guards \( \varphi \) are eliminated eagerly. New concrete states are created for all the reachable merged states. Finally, the normalized STA is cleaned by eliminating states that accept no trees, e.g., by using elimination of useless symbols from a context-free grammar [Hopcroft and Ullman 1979, p. 88–89].}
Proposition 3.4. The non-emptiness 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, i.e., when $\sigma = \{()\}$.

Proposition 3.5. The non-emptiness problem of alternating STAs without attributes is ExpTime-complete.

Proof. For inclusion in ExpTime, consider an STA $A = (Q, T, \delta)$ and $q \in Q$. Here $\sigma = \{()\}$, i.e., there are no attributes. Construct an alternating tree automaton $A' = (\hat{Q}, \Sigma, \{q\}, \Delta)$ over $\Sigma$ with state set $\hat{Q}$, initial state $q$, and mapping $\Delta$ such that for $(q, f) \in Q \times \Sigma$,

$$\Delta(q, f) \overset{\text{def}}{=} \bigvee_{(q, f, r, t) \in \delta(q)} \bigwedge_{i=1}^{t(f)} (p, i).$$

Then $L(A)$ is non-empty iff $L^q_A$ is non-empty. 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 $\Delta$. Instead, let $A_i = (Q_i, T_i, \delta_i)$ 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 $\Sigma$ to $\Sigma' = \Sigma \cup \{f\}$ where $f$ is a fresh symbol of rank 1. Let $A$ be the STA $((\{q\} \cup \bigcup_i Q_i, T_{\Sigma'}, \delta, \bigcup_i \{(q, f, \lambda x.\text{true}, \{(q_i)_{1 \leq i \leq n}\})\})$ where $q$ is a new state. It follows from the definitions that $L^q_A \neq \emptyset$ iff $\bigcap_i L^q_{A_i} \neq \emptyset$. ExpTime-hardness follows now from the intersection non-emptiness 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

Symbolic tree transducers (STTs) augment STAs with outputs. Symbolic tree transducers with regular look-ahead further augment STTs by allowing rules to be guarded by symbolic tree automata. Intuitively, a rule is applied to a node if and only if its children are accepted by some symbolic tree automata. We first define terms that are used below as output components of transformation rules. We assume that we have a given tree type $T_{\Sigma'}$ for both the input trees as well as the output trees. In the case that the input tree type and the output tree type are intended to be different, we assume that $T_{\Sigma'}$ 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 $T_{\Sigma \cup \{\text{State}\}}$ where State $\notin \Sigma$ is a new fixed symbol of rank 1. A term $\text{State}[q](t)$ is always used with a concrete value $q$ and $\text{State}[q]$ is also written as $\bar{q}$. The idea is that, in $\bar{q}$ the value $q$ is always viewed as a state.

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

- for all $q \in Q$, and all $i, 1 \leq i \leq k$, $\lambda(x, \bar{y}).\bar{q}(y_i) \in S$;
- for all $f \in \Sigma$, all $e : \sigma \rightarrow \sigma$ and all $t_1, \ldots, t_\ell(f) \in S$, $\lambda(x, \bar{y}).f[e(x)][t_1(x, \bar{y}), \ldots, t_\ell(f)(x, \bar{y})] \in S.$
A rule \((q, f, \varphi, \bar{\ell}, t)\) is also denoted by \(q \xrightarrow{f,\bar{\ell}} t\). The open view of a rule \(q \xrightarrow{f,\bar{\ell}} t\) is \(\bar{q}(f[x](\tilde{y})) \xrightarrow{\bar{\varphi}(x)} t(x, \tilde{y})\). The open view is technically more convenient and more intuitive for term rewriting. The look-ahead, when omitted, is \(\emptyset\) by default. Figure 5 illustrates an open view of a linear rule over the tree type \(T^\Sigma_{2^3}\) over \(\Sigma_1 = \{f, g, h\}\), where \(\bar{z}(f) = 2\), \(\bar{z}(g) = 3\), and \(\bar{z}(h) = 0\).

Let \(T\) be an STTR \((Q, q^0, \mathcal{T}_q^\Sigma, \Delta)\). 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 \(\bar{q}(y_i)\) occurs in \(t\).

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

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 \((q, g, x. x < 4, \Sigma) \rightarrow_{\varphi}(f)\) where \(\varphi(x) = 2\), \(\bar{z}(g) = 3\), and \(\bar{z}(h) = 0\).

We recall that given a lambda term \(u = \lambda(x, \tilde{y}).v\), the term \(u(a, \tilde{s})\) is the function application of \(u\) to \((a, \tilde{s})\), where \(a\) and \(\tilde{s}\) substitute \(x\) and \(\tilde{y}\) respectively. In the following let \(T\) be the STTR, and for \(\ell \subseteq Q\), let \(L^\ell_T \equiv L^\ell_{d(T)}\).

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

\[
T^q_T(t) \overset{\text{def}}{=} \psi_T(q(t))
\]

\[
\psi_T(q(t)) = \bigcup \{\psi_T(u(a, \tilde{s})) \mid (q, f, \varphi, \bar{\ell}, u) \in \Delta_T, a \in [\varphi], \bigwedge_{i=1}^{\bar{z}(f)} s_i \in L^\ell_T\}
\]

\[
\psi_T(t) = \{f[a](\tilde{v}) \mid \bigwedge_{i=1}^{\bar{z}(f)} v_i \in \psi_T(s_i)\}
\]

The transduction of \(T\) is \(T_T \overset{\text{def}}{=} T^0_T\). The definitions are lifted to sets using union.

---

For \(k = 0\) we assume that \((2^\Sigma)^k = \{\}\), i.e., a rule for \(c \in \Sigma(0)\) has the form \((q, c, \varphi, (\lambda x. t(x))\) where \(t(x)\) is a tree term.
We omit $T$ from $T_q^s$ and $\phi_T$ 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 $i$ for a state that outputs the identity transformation. The “safe” case is

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

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

In FAST, a transformation $T^q$ is defined by the statement

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

where $\ell(\bar{y})$ denotes the look-ahead $\{(r \mid (r y_i) \in \ell(\bar{y}))\}_{i=1}^{\ell(f)}$. The semantics of a FAST transformations 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(t_1, t_2) where (odd x) |
  L() where (odd x) 
}
def evenRoot : BT := (complement oddRoot)
trans h : BT \rightarrow BT { 
  N(t_1, t_2) given (oddRoot t_1) to (N [\neg x] (h t_1) (h t_2)) |
  N(t_1, t_2) given (evenRoot t_1) to (N [x] (h t_1) (h t_2)) |
  L() to (L [x]) 
}
```

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

Definition 9. $T$ is single-valued if $\forall (t \in \mathcal{T}_Q^s, q \in Q_T) : |T_q^s(t)| \leq 1$.

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 $q \xrightarrow{f, \varphi, \bar{r}} t$ and $q \xrightarrow{\bar{f}, \psi, \bar{r}} u$ in $\Delta_T$, if $\varphi \cap \psi \neq \emptyset$ and, for all $i \in \{1, \ldots, \bar{z}(f)\}$, $L^f_i \cap L^{\bar{r}_i} \neq \emptyset$, 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

```
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 $t$ iff 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 $L[true]$ iff 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 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 iff for every input in $A_1$, $T$ only produces outputs in $A_2$.

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

Proposition 3.9. The following operations can be expressed as

\[
\begin{align*}
\text{restrict } T \ A &= \ \text{compose } I_A \ T \\
\text{restrict-out } T \ A &= \ \text{compose } T \ I_A \\
\text{pre-image } T \ A &= \ \text{domain} (\text{restrict-out } T \ A) \\
\text{type-check } A_1 \ T \ A_2 &= \ \text{is-empty} (\text{intersect } A_1 \ (\text{pre-image } T \ (\text{complement } A_2)))
\end{align*}
\]

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 pre-image computation and output restriction. Unfortunately, as shown in Example 3.8 and in [Fülop 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 paper 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.

\[
\text{trans } f : BT \rightarrow BT \{ \\
\text{ } L() \ \text{to} \ (L \ [i]) \\
\text{ } L() \ \text{to} \ (L \ [5]) \\
\text{ } N(x, y) \ \text{to} \ (N \ [i] \ (f \ x) \ (f \ y)) \\
\}
\]

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

\[
\text{trans } g : BT \rightarrow BT \{ \\
\text{ } L() \ \text{to} \ (N \ [0] \ (L \ [i]) \ (L \ [i])) \\
\text{ } N(x, y) \ \text{to} \ (N \ [0] \ (N \ [i] \ x \ y) \ (N \ [i] \ x \ y)) \\
\}
\]

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 transducer. 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, $T_{S \circ T} = T_S \circ T_T$. The composition $T_S \circ T_T$ is defined as $(T_S \circ T_T)(x) = \bigcup_{y \in T_S(x)} T_T(y)$, 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 $p.q \notin Q_S \cup Q_T$. In a nutshell, we use a least fixed point construction starting with the initial state $q_0^S,q_0^T$. Given a reached (unexplored) pair state $p.q$ and symbol $f \in \Sigma$, the rules from $p.q$ and $f$ are constructed by considering all possible constrained rewrite reductions of the form

$$(true, (\emptyset)_{i=1}^{|f|}, \bar{q}(\bar{p}(f[x](y)))) \xrightarrow{\cdot \leftarrow} (p.q, \bar{q}(\bar{p}(y)))$$

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

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

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

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

The rewriting steps are done modulo attribute constraints. To this end, a $k$-configuration is a triple $(\gamma, L, u)$ where $\gamma$ is a formula with $\text{FV}(\gamma) \subseteq \{x\}$, $L$ is a $k$-tuple of sets of pair states $p.q$ where $p \in Q_S$ and $q \in Q_T$, and $u$ is an extended tree term. We use configurations to describe reductions of $T$. Formally, given two STTRs $S = (Q_S, q_0^S, T_S^f, \Delta_S)$ and $T = (Q_T, q_0^T, T_T^f, \Delta_T)$, the composition of $S$ and $T$ is defined as follows

$$S \circ T \eqdef \left( Q_S \cup \{p.q \mid p \in Q_S, q \in Q_T \}, q_0^S, q_0^T, T_S^f, T_T^f, \Delta_S \cup \bigcup_{p \in Q_S, q \in Q_T, f \in \Sigma} \text{Compose}(p.q, f) \right)$$

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

**Compose**(p,q,f) \eqdef

1. **choose** (p,f,φ,ℓ,u) from $\Delta_S$
2. **choose** (ψ,P,t) from Reduce(φ,(∅)_{i=1}^{|f|},q(u))
3. **return** (p,q,f,ψ,ℓ⊎P,t)

The procedure **Reduce** uses a procedure **Look**(φ, L, q, t) that, given an attribute formula $φ$ with $\text{FV}(φ) \subseteq \{x\}$, 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 $d(T)$ is normalized. We define a function $\text{sin}$, such that $\text{sin}(\{e\}) \eqdef \text{undefined}$ for any singleton set $\{e\}$, and $\text{defined}$ otherwise. This function extracts the only element in a singleton set. Notice that since we operate over normalized transducers, $\text{sin}$ is always defined.

Loris D’Antoni et al.
Lemma. this means that rewrite steps of $T$ that allows states of $S$
moreover, the satisfiability check in 2(a) ensures that $\varphi$ 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 $\text{Look}(x>0, \emptyset, q, t)$ for $t = g[x+1](\gamma_1(y_1, y_2))$ where $g \in \Sigma(1)$. Suppose there is a rule $(g, g, \lambda x.odd(x), \{q\}) \in \delta_{\text{dt}}(T)$ that requires that all attributes of $g$ are odd and assume that there is no other rule for $g$. The term $t$ itself may arise as an output of a rule $\bar{p}(f[x](y_1, y_2)) \rightarrow g[x+1](\gamma_1(y_1, y_2))$ 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 $\text{Look}$ procedure.

In $\text{Look}(x>0, \emptyset, q, t)$ line 2(c) we have the recursive call $\text{Look}(x>0 \land odd(x+1), \emptyset, g, q[x-2](\bar{p}_1(y_2)))$. Inside the recursive call we have the failing satisfiability check of $\text{IsSat}(x>0 \land odd(x+1) \land odd(x-2))$ 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 $\text{Look}(x>0, \emptyset, q, t)$ is empty.

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

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

(1) if $v = \bar{q}(\bar{p}(y_i))$ where $q \in Q_T$ and $p \in Q_S$ then return $(\gamma, L, \bar{p}, \bar{q}(y_i))$

(2) if $v = \bar{g}[u_0](\bar{u})$ where $q \in Q_T$ and $g \in \Sigma$ then
(a) choose $\tau = (q, g, \ldots)$ from $\Delta_T$
(b) choose $(\gamma_1, L_1)$ from $\text{Look}(\gamma, L, q, g[u_0](\bar{u}))$
(c) choose $\chi$ from $\text{Reduce}(\gamma_1, L_1, t(u_0, \bar{u}))$ return $\chi$

(3) if $v = g[t_0](\bar{t})$ where $g \in \Sigma$ then
(a) $\gamma_0 := \gamma$, $L_0 := L$
(b) for $(i = 1; i \leq \bar{g}(g); i++)$
   choose $(\gamma_i, L_i, u_i)$ from $\text{Reduce}(\gamma_{i-1}, L_{i-1}, t_i)$
   return $(\gamma_{i}(g), L_{i}(g), g[t_0](\bar{u}))$

There is a close relationship between $\text{Reduce}$ and Definition 8. We include the case

$$\text{T}_p \overset{\text{def}}{=} \text{T}_p(T_p(t))$$

for $p \in Q_S$ and $t \in T_S$, (2)

that allows states of $S$ to occur in the input trees to $\text{T}_p$ in a non-nested manner. Intuitively this means that rewrite steps of $T$ are carried out first while rewrite steps of $S$ are being postponed (called by name). In order to justify the extension (2) we need the following Lemma.
LEMMA 4.4. For all \( t \in \Lambda(T_S^\varnothing, Q_S, k) \), \( a \in \sigma \), and \( \bar{u} \in T_S^\varnothing \):

1. \( T_F^\varnothing(t_S(t(a, \bar{u}))) \subseteq T_F^\varnothing(t(a, \bar{u})) \), and
2. \( T_F^\varnothing(t_S(t(a, \bar{u}))) = T_F^\varnothing(t(a, \bar{u})) \) when \( S \) is single-valued or \( T \) is linear.

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

\[
T_F^\varnothing(t_S(p(u_1))) = T_F^\varnothing(T_S^\varnothing(u_1)) = T_F^\varnothing(\bar{p}(u_1))
\]

where the last equality holds by using equation (2). The induction case is as follows. Let \( t = \lambda(x, \bar{y}).f[t_0(x)](t_1(x, \bar{y})^{(i)}_{i=1}) \). Suppose \( z(f) = 1 \), the proof of the general case is analogous.

\[
T_F^\varnothing(t_S(f[t_0(a)](t_1(a, \bar{u}))))
\]

\[
\text{Def} ~ t_S \{ f[t_0(a)](v) ~|~ v \in t_S(t_1(a, \bar{u})) \}
\]

\[
\text{Def} ~ T_F^\varnothing \{ w(t_0(a), (u_i)_{i=1}^m) ~|~ (\exists \varphi, \bar{t}, \bar{q}) t_0(a) \in \llbracket \varphi \rrbracket \\
q \frac{f, \varphi, \bar{t}}{\lambda(x, y).w(x, (\bar{q}_i(y))_{i=1}^m) \in \Delta_T} \\
\bigwedge_{i=1}^m u_i \in T_F^\varnothing(t_1(a, \bar{u})) \}
\]

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

Example 4.5. The example shows a case when

\[
T_F^\varnothing(t_S(t(a, \bar{u}))) \neq T_F^\varnothing(t(a, \bar{u})�)
\]

Suppose \( p \xrightarrow{c, T} \bullet \), \( p \xrightarrow{c, T} \triangle \), and \( q \xrightarrow{g, T} \lambda xy.f[x](\bar{q}(y), \bar{q}(y)) \). Let \( f = f[0], c = c[0], g = g[0]. \)

Then

\[
\bar{q}(g(p(c))) \xrightarrow{T} f(\bar{q}(\bar{p}(c))) \\
\xrightarrow{\text{Def}} \{ f(\bar{q}(\bullet)), f(\bar{q}(\triangle)) \cup \{ f(\bar{q}(\bullet)), f(\bar{q}(\triangle)) \}
\]

Lemma 4.4. It implies that, in general, \( T \) becomes the composition result proven in Theorem 2.11 of [Engelfriet 1977]. Theorem 4.6 uses Lemma 4.4. 

It does not directly follow from classic results because, for example, \( f \) behaves as a partial function.

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 \( \Sigma \times \sigma \) 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, \( T_{S \circ T} \) is an overapproximation of \( T_{S} \circ T_{T} \), and that \( T_{S \circ T} \) captures \( T_{S} \circ T_{T} \) 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 \( t \in T_{S}^{p,q}, T_{S \circ T}^{p,q}(t) \supseteq T_{T}^{p}(T_{S}^{p}(t)) \), and if \( S \) is single-valued or if \( T \) is linear then \( T_{S \circ T}^{p,q}(t) \subseteq T_{T}^{p}(T_{S}^{p}(t)) \).

**Proof.** We start by introducing auxiliary definitions and by proving additional properties that help us to formalize our arguments precisely. For \( p \in Q_{S} \) and \( q \in Q_{T} \), given that \( L_{p-q} \) is the language accepted at the pair state \( p.q \), we have the following relationship that is used below:

\[
L_{p-q} \stackrel{\text{def}}{=} \{ t \mid T_{T}^{p}(T_{S}^{p}(t)) \neq \emptyset \}
\]

\[
= \{ t \mid \exists u \in T_{S}^{p}(t) \land T_{T}^{p}(u) \neq \emptyset \}
\]

\[
= \{ t \mid \exists u \in T_{S}^{p}(t) \land u \in L_{T} \}
\]

\[
= \{ t \mid T_{S}^{p}(t) \cap L_{T} \neq \emptyset \}
\]

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

\[
L_{p} \stackrel{\text{def}}{=} \bigcap_{p.q \in P} L_{p-q}
\]

\[
L_{\bar{P}} \stackrel{\text{def}}{=} \{ \bar{u} \mid \bigwedge_{i=1}^{k} u_{i} \in L_{P_{i}} \}
\]

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

\[
[\text{Look}(\varphi, \bar{P}, q, t)]_{\Sigma} \stackrel{\text{def}}{=} \{(a,\bar{u}) \mid a \in [\varphi], \bar{u} \in L_{\bar{P}}, \downarrow_{S}(t(a,\bar{u})) \cap L_{T} \neq \emptyset \}
\]

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

\[
[(\varphi, \bar{P})]_{\Sigma} \stackrel{\text{def}}{=} \{(a,\bar{u}) \mid a \in [\varphi], \bar{u} \in L_{\bar{P}} \}
\]

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

\[
\bigcup \{[\chi] \mid \text{Look}(\varphi, \bar{P}, q, t) \text{ returns } \chi \} = [\text{Look}(\varphi, \bar{P}, q, t)]_{\Sigma}
\]
We prove (4) by induction over \( t \). The base case is when \( t = \overline{p}(y_i) \) for some \( p \in Q_S \) and \( y_i \) for some \( i \in \{1, \ldots, k\} \):

\[
\bigcup\{\{x\}| \text{Look}(\varphi, \overline{P}, q, \overline{p}(y_i)) \text{ returns } x\} = [[(\varphi, \overline{P} \cup_i p \cdot q)]] \\
= \{(a, \bar{u}) | a \in \varphi, \bar{u} \in L^{P}, u_i \in L^{P-q}\} \\
= \{(a, \bar{u}) | a \in \varphi, \bar{u} \in L^{P}, T^p(u_i) \cap L^{P-q} \neq \emptyset\} \\
= \{(a, \bar{u}) | a \in \varphi, \bar{u} \in L^{P}, \downarrow_S(\overline{p}(u_i)) \cap L^{P-q} \neq \emptyset\} \\
= [[\text{Look}(\varphi, \overline{P}, q, \overline{p}(u_i))]]
\]

The induction case is when \( t = f[t_0](\bar{t}) \). Assume \( \sharp(f) = 2 \). IH is that (4) holds for \( t_1 \) and \( t_2 \). Assume, without loss of generality, that \( d(T) \) is normalized. We have for all \( a \in \sigma \) and
Fast: a Transducer-Based Language for Tree Manipulation

\[ \bar{u} \in (\mathcal{T}_\Sigma)^k, \]

\[ (a, \bar{u}) \in \bigcup \{ [\chi] \mid \text{Look}(\varphi, \bar{P}, f[t_0][\bar{t}]) \text{ returns } \chi \} \]

\begin{align*}
(\text{Def Look}) & \iff (\exists \psi, q_1, q_2) (q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_d(T), \\
& \quad \text{IsSat}(\varphi \land \psi(t_0)), \\
& \quad (\exists \varphi', \bar{P}', \varphi'', \bar{P}'') \\
& \quad \text{Look}(\varphi \land \psi(t_0), \bar{P}, q_1, t_1) \text{ returns } (\varphi', \bar{P}'), \\
& \quad \text{Look}(\varphi', \bar{P}', q_2, t_2) \text{ returns } (\varphi'', \bar{P}''), \\
& \quad (a, \bar{u}) \in \llbracket (\varphi'', \bar{P}'') \rrbracket \\
(\text{IH}) & \iff (\exists \psi, q_1, q_2) (q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_d(T), \\
& \quad \text{IsSat}(\varphi \land \psi(t_0)), \\
& \quad (\exists \varphi', \bar{P}'), \\
& \quad \text{Look}(\varphi \land \psi(t_0), \bar{P}, q_1, t_1) \text{ returns } (\varphi', \bar{P'}), \\
& \quad (a, \bar{u}) \in \llbracket \text{Look}(\varphi', \bar{P}', q_2, t_2) \rrbracket \\
(\text{Eq} (3)) & \iff (\exists \psi, q_1, q_2) (q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_d(T), \\
& \quad \text{IsSat}(\varphi \land \psi(t_0)), \\
& \quad (a, \bar{u}) \in \llbracket \text{Look}(\varphi \land \psi(t_0), \bar{P}, q_1, t_1) \rrbracket, \\
& \quad \downarrow_S(t_2(a, \bar{u})) \cap L_{\bar{P}}^{\bar{t}} \neq \emptyset \\
(\text{Eq} (3)) & \iff (\exists \psi, q_1, q_2) (q, f, \psi, (\{q_1\}, \{q_2\})) \in \delta_d(T), \\
& \quad \text{IsSat}(\varphi \land \psi(t_0)), \\
& \quad a \in \llbracket \varphi \rrbracket \cap \llbracket \psi(t_0) \rrbracket, \\
& \quad \downarrow_S(t_1(a, \bar{u})) \cap L_{\bar{P}}^{\bar{t}} \neq \emptyset, \\
& \quad \downarrow_S(t_2(a, \bar{u})) \cap L_{\bar{P}}^{\bar{t}} \neq \emptyset \\
(\text{Def} 2) & \iff a \in \llbracket \varphi \rrbracket, \\
& \quad \downarrow_S(f[t_0(a)](t_1(a, \bar{u}), t_2(a, \bar{u}))) \cap L_{\bar{P}}^{\bar{t}} \neq \emptyset \\
\iff a \in \llbracket \varphi \rrbracket, \\
& \quad \downarrow_S(t(a, \bar{u})) \cap L_{\bar{P}}^{\bar{t}} \neq \emptyset \\
(\text{Eq} (3)) & \iff (a, \bar{u}) \in \llbracket \text{Look}(\varphi, \bar{P}, q, t) \rrbracket \\
\end{align*}

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 \textbf{Reduce} denotes the function \( \partial_{(\varphi, \bar{P}, t)} \) such that, for all \( a \in \sigma \) and \( u_i \in \mathcal{T}_\Sigma^{\bar{P}} \),

\[ \partial_{(\varphi, \bar{P}, t)}(a, \bar{u}) \overset{\text{def}}{=} \begin{cases} \\ \downarrow_T(t(a, \bar{u})), & \text{if } (a, \bar{u}) \in \llbracket (\varphi, \bar{P}) \rrbracket; \\ \emptyset, & \text{otherwise.} \end{cases} \quad (5) \]
Next, we prove (6) under the assumption that $S$ is single-valued or $T$ is linear. For all $a \in \sigma$, $u_i \in T^G$ and $v \in T^G$,

$$\exists \alpha \in \sigma, u_i \in T^\sigma \Sigma \text{ and } v \in T^\sigma \Sigma, (\exists \alpha) v \in \partial_{\alpha}(a, u_i).$$ 

(6)

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

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

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

$$\uparrow_T(q(f[a](\tilde{t}))) \overset{\text{def}}{=} \bigcup \{ \uparrow_T(u(a, \tilde{t})) \mid (q, f, \varphi, \tilde{\ell}, u) \in \Delta_T, a \in [\varphi], \\
\wedge (\forall i \in \ell(f)) \uparrow_S(t_i) \cap L^i_T \neq \emptyset \}$$

(7)

Observe that when $t_i$ does not contain any states of $S$ then $\uparrow_S(t_i) = \{ t_i \}$ and thus the condition $\uparrow_S(t_i) \cap L^i_T \neq \emptyset$ simplifies to the condition $t_i \in L^i_T$ used in the original version of Definition 8.
There are two induction cases. The first induction case is $t = \overline{q}(f[t_0](\overline{t}))$ where $q \in Q_T$ and $f \in \Sigma$. Let $t' = f[t_0](\overline{t})$. For all $a \in \sigma, u_i \in T^a_T$ and $v \in T^v_T$,

$$\exists \alpha \quad v \in \partial_\alpha(a, \overline{u}), \text{Reduce}(\varphi, \overline{P}, \overline{q}(t')) \quad \text{return} \alpha$$

$$\forall \psi, \overline{R} \quad \text{Look}(\overline{P}, q_\tau, t') \quad \text{return} \psi, \overline{R}$$

$$\exists \beta \quad \text{Reduce}(\psi, \overline{R}, u(t_0, \overline{t})) \quad \text{return} \beta$$

$$v \in \partial_\beta(a, \overline{u})$$

$$\exists \tau, u, \gamma, \overline{t} \quad \tau = q f, \overline{\gamma} \quad u \in \Delta_T$$

$$\exists \psi, \overline{R} \quad \text{Look}(\overline{P}, q_\tau, t') \quad \text{return} \psi, \overline{R}$$

$$v \in \psi_T(u(t_0(a), t(a, \overline{u}))), (a, \overline{u}) \in \psi_T((\psi, \overline{R}))$$

$$\exists \psi, \overline{R} \quad \text{Look}(\overline{P}, q_\tau, t') \quad \text{return} \psi, \overline{R}$$

$$v \in \psi_T(u(t_0(a), t(a, \overline{u})))$$

$$\exists \tau, u, \gamma, \overline{t} \quad \tau = q f, \overline{\gamma} \quad u \in \Delta_T$$

$$a \in \psi, \overline{u} \in L^\overline{P}, \psi_T(t'(a, \overline{u})) \cap L^\psi_T \neq \emptyset$$

$$v \in \psi_T(u(t_0(a), t(a, \overline{u})))$$

$$\exists \tau, u, \gamma, \overline{t} \quad \tau = q f, \overline{\gamma} \quad u \in \Delta_T$$

$$t_0(a) \in \psi, \overline{u} \in L^\overline{P}, \psi_T(t_i(a, \overline{u})) \cap L^\psi_T \neq \emptyset$$

$$v \in \psi_T(u(t_0(a), t(a, \overline{u})))$$

The second induction case is $t = f[t_0](\overline{t})$. Assume $\zeta(f) = 2$. Generalization to arbitrary ranks is straightforward by repeating IH steps below $\zeta(f)$ times. For all $a \in \sigma, u_i \in T^a_T$ and
\( \forall v \in T^v_2, \)

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

\[\begin{align*}
\text{Def \ Reduc}_{\emptyset} & \iff (\exists \varphi', \bar{P}', v_1, \varphi'', \bar{P}'', v_2) \\
& \text{Reduce}(\varphi, \bar{P}, t_1) \text{ returns } (\varphi', \bar{P}', v_1) \\
& \text{Reduce}(\varphi', \bar{P}', t_2) \text{ returns } (\varphi'', \bar{P}'', v_2) \\
& \forall v \in \partial((\varphi'', \bar{P}'', f([t_0])(v_1, v_2)))(a, \bar{u})
\end{align*}\]

\[\begin{align*}
\text{Def } \emptyset & \iff (\exists \varphi', \bar{P}', w_1, \varphi'', \bar{P}'', w_2) \\
& \text{Reduce}(\varphi, \bar{P}, t_1) \text{ returns } (\varphi', \bar{P}', w_1) \\
& \text{Reduce}(\varphi', \bar{P}', t_2) \text{ returns } (\varphi'', \bar{P}'', w_2) \\
& \forall v \in \mathcal{R}(f([t_0](a))(w_1(a, \bar{u}), w_2(a, \bar{u}))), \\
& a \in \mathcal{L}(\varphi''), \bar{u} \in L^{\emptyset}''
\end{align*}\]

\[\begin{align*}
\text{H} & \iff (\exists \varphi', \bar{P}', w_1) \\
& \text{Reduce}(\varphi, \bar{P}, t_1) \text{ returns } (\varphi', \bar{P}', w_1) \\
& (\exists v_1, v_2) v = f([t_0](a))(v_1, v_2) \\
& v_1 \in \mathcal{R}(w_1(a, \bar{u})), v_2 \in \mathcal{R}(w_2(a, \bar{u})) \\
& a \in \mathcal{L}(\varphi'), \bar{u} \in L^{\emptyset}''
\end{align*}\]

\[\begin{align*}
\text{Def } \emptyset & \iff (\exists \varphi', \bar{P}', w_1) \\
& \text{Reduce}(\varphi, \bar{P}, t_1) \text{ returns } (\varphi', \bar{P}', w_1) \\
& (\exists v_1, v_2) v = f([t_0](a))(v_1, v_2) \\
& v_1 \in \mathcal{R}(w_1(a, \bar{u})), v_2 \in \mathcal{R}(w_2(a, \bar{u})) \\
& a \in \mathcal{L}(\varphi'), \bar{u} \in L^{\emptyset}''
\end{align*}\]

Equation (6) follows by the induction principle.
Finally, we prove $T^{p\rightarrow q}_{S\rightarrow T} = T^p_S \circ T^q_T$. Let $p \in Q_S$, $q \in Q_T$ and $f[a](\bar{u}), w \in T^q_T$ be fixed.

\[
\begin{align*}
w &\in T^{p\rightarrow q}_{S\rightarrow T}(f[a](\bar{u})) \\
&\text{Def Compose} \implies (\exists \phi, \bar{t}, t) (p, f, \phi, \bar{t}, t) \in \Delta_S \\
&\quad (\exists \alpha) \text{Reduce}(\phi, \bar{t}, \bar{q}(t)) \text{ returns } \alpha \\
&\quad w \in \partial_\alpha(a, \bar{u}), \bar{u} \in L^\bar{t}_S \\
&\text{Eq (4)} \implies (\exists \phi, \bar{t}, t) (p, f, \phi, \bar{t}, t) \in \Delta_S \\
&\quad w \in \partial_{(\phi, \bar{t}, \bar{q}(t))}(a, \bar{u}), \bar{u} \in L^\bar{t}_S \\
&\text{Def } \theta \implies (\exists \phi, \bar{t}, t) (p, f, \phi, \bar{t}, t) \in \Delta_S \\
&\quad a \in [\phi], \bar{u} \in L^a, w \in \downarrow_T(\bar{q}(t(a, \bar{u}))), \bar{u} \in L^\bar{t}_S \\
&\text{Def } T^{\theta}_S \implies (\exists \phi, \bar{t}, t) (p, f, \phi, \bar{t}, t) \in \Delta_S \\
&\quad a \in [\phi], \bar{u} \in L^\bar{t}_S, w \in T^q_T(t(a, \bar{u})) \\
&\text{Eq (5)} \implies (\exists \phi, \bar{t}, t) (p, f, \phi, \bar{t}, t) \in \Delta_S \\
&\quad a \in [\phi], \bar{u} \in L^\bar{t}_S, w \in T^q_T(t(a, \bar{u})) \\
&\text{Def } \psi_S \implies w \in T^q_T(\psi_S(\bar{p}(f[a](\bar{u})))) \\
&\text{Def } T^{\psi}_S \implies w \in T^q_T(\psi_S(f[a](\bar{u})))
\end{align*}
\]

Step (\star) uses Lemma 4.4.2. It holds only when $S$ is single-valued or $T$ is linear. Otherwise, only ‘⇐’ holds. $\square$

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 scenarios above, there have been high-profile vulnerabilities stemming from careless sanitization of specially crafted HTML input leading to the creation of the infamous Samy worm for MySpace (http://namb.la/popular/) and the Yamanner worm for the Yahoo Mail system. In fact, MySpace has repeatedly failed to properly sanitize their HTML inputs, leading to the Month of MySpace Bugs initiative (http://momby.livejournal.com/586.html).

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

We show how FAST is expressive enough to model HTML sanitizers, and we argue that writing such programs is easier with FAST than with current tools. Our version of an HTML sanitizer written in FAST and automatically translated by the FAST compiler into C# is partially described in Section 2. Although we can’t 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 above HTML sanitization is implemented as a monolithic function in order to achieve reasonable performance. In the case of FAST each sanitization routine can be written as a single function and all 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 20 KB (Bing) to 409 KB in size (Facebook). For speed, the FAST-based sanitizer is comparable to HTML Purify. In terms of maintainability, FAST wins on two counts. First, unlike sanitizers written in PHP, FAST programs can be analyzed statically. Second, our sanitizer is only 200 lines of FAST code instead of 10000 lines of PHP. While these are different languages, we argue that our approach is more maintainable because FAST captures the high level semantics of HTML sanitization, as well as being fewer lines of code to understand. We manually spot-checked the outputs to determine that both produce reasonable sanitizations.
5.2. Conflicting Augmented Reality Applications

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

We call a tagger an AR application that labels elements of a given set with a piece of information based on the properties of such elements. As an example, consider a tagger that assigns to every city a set of tags representing the monuments in such 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 \( \langle p_1, p_2 \rangle \):

1. **composition.** We compute \( p' \), composition of \( p_1 \) and \( p_2 \);
2. **input restriction.** We compute \( p'_1 \), a restriction of \( p \) that is only defined on trees where each node contains no tags;
3. **output restriction.** We compute \( p'' \), a restriction of \( p' \) that only outputs trees in which some node contains two tags;
4. **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, two attributes of type real, and an attribute of type int. In our encoding the left-child of each element was the list of tags, while the right child was the next element. Each tagger we generated conforms to the following properties: 1) it is non-empty; 2) it tags on average 3 nodes; and 3) it tags each node at most once.

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

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

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 be-
tween 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 analysed the performance gain obtained by deforestation in Fast.

**Evaluation:** We considered the function `map caesar` from Figure 8 that replaces each value \( x \) of a integer list with \((x+5)\mod 26\). We composed the function `map caesar` with itself several times to see how the performance changed when using Fast. Let’s call `map^n` the composition of `map caesar` with itself \( n \) times. 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 into 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 a integer list with \((x+5)\mod 26\). The function `filter ev` removes all the odd elements from a list.

One might wonder what happens when such functions are composed. Consider the case in which we execute the map followed by the filter, followed by the map, and again by the filter. This transformation is equivalent to deleting all the elements in the list! This property can be statically checked in Fast. We first compute `comp2` as the composition described above. As show in Figure 8, the language of non-empty lists can be expressed using the construct `not_emp_list`. Finally, we can use the output restriction to restrict `comp2` to only output non-empty lists and show that such function is empty. In this example the whole analysis can be done in less than 10 ms.
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 comp2 never outputs a non-empty list. Example available at http://rise4fun.com/Fast/Jv.

5.5. CSS Analysis

Cascading style-sheets (CSS) is a language that allows 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 doesn’t exists 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 modelling CSS programs as symbolic tree transducers we can overcome this limitation. This analysis relies on the alphabet being symbolic, and we plan on extending Fast with primitives for simplifying CSS modelling.

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 Fig. 2 can be represented by classic finite state transducers with regular lookahead. 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 Fig. 3. In our representation each string value is modelled 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, e.g., as unary function symbols $f_c$ for the characters $c$. If we want to support full Unicode, e.g., 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, e.g., Z3 [De Moura and Bjørner 2008]), and involve fairly simple arithmetic operations.

We need to add lookahead 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 lookahead 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 $6 \times (2^{16} - 1)$ rules. The other string constraints are handled similarly. Besides the additional lookahead 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 [Fülpö 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; Fülpö 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 don’t consider extending them in this paper.

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 [Botičan 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 [Fülpö and Vogler 1998] which is only stated for total deterministic finite tree transducers. In [Fülpö 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 transducers 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 transducers 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 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. Table 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 [Künnemann 1999]. In this setting deforestation is done via Macro Tree Transducers (MTT) [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 [Künnemann 1999]. Many models of tree transducers have been introduced to 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 of operations in this setting orthogonal to symbolic automata and transducers. Higher-Order Multi-Parameter Tree Transducers (HMTT) [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
<table>
<thead>
<tr>
<th>Language</th>
<th>$\sigma$</th>
<th>Analysis</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>$\infty$</td>
<td>composition; typechecking, pre-image,</td>
<td>Tree-manipulating programs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>language equivalence, determinization,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>complement, intersection</td>
<td></td>
</tr>
<tr>
<td>VATA</td>
<td>ff</td>
<td>union, intersection, language inclusion</td>
<td>Tree-automata</td>
</tr>
<tr>
<td>Tiburon</td>
<td>ff</td>
<td>composition; type-checking; training;</td>
<td>NLP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>weights; language equivalence, determiniz-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>tion, complement, intersection</td>
<td></td>
</tr>
<tr>
<td>TTT</td>
<td>ff</td>
<td>-</td>
<td>NLP</td>
</tr>
<tr>
<td>ASF+SDF</td>
<td>$\infty$</td>
<td>-</td>
<td>Parsing</td>
</tr>
<tr>
<td>XPath</td>
<td>$\infty$</td>
<td>emptiness for a fragment</td>
<td>XML query (only selection)</td>
</tr>
<tr>
<td>XDuce</td>
<td>$\infty$</td>
<td>type-checking for navigational part (finite</td>
<td>XML query</td>
</tr>
<tr>
<td></td>
<td></td>
<td>alphabet)</td>
<td></td>
</tr>
<tr>
<td>XQuery, XSLT, STX</td>
<td>$\infty$</td>
<td>-</td>
<td>XML transformations</td>
</tr>
</tbody>
</table>

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

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 non-emptiness problem of a set of normalized STAs as the emptiness of a single un-normalized STA. In the classic case, the non-emptiness problem of tree automata is P-co, while the intersection non-emptiness problem is ExpTIME-co [Comon et al. 2007, Thm 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/.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback that helped us improving the quality of our paper. 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.

REFERENCES


