# Fast: a Transducer-Based Language for Tree Manipulation

Loris D'Antoni
University of Pennsylvania
lorisdan@cis.upenn.edu

Margus Veanes Benjamin Livshits
David Molnar
Microsoft Research
{margus,livshits,dmolnar}@microsoft.com

## Abstract

Tree automata and tree transducers are used in a wide range of applications in software engineering, from XML processing to language type-checking. While these formalisms are of immense practical use, they can only model finite alphabets, and since many real-world applications operate over infinite domains such as integers, this is often a limitation. 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 classical 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. Fast supports symbolic alphabets through tight integration with state-of-the-art satisfiability modulo theory (SMT) solvers. We demonstrate our techniques on practical case studies, covering a wide range of applications.

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

***Keywords*** Symbolic Tree Transducers, FAST

## 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 [27] to language type-checking [37]. Tree transducers extend tree automata to model functions over trees, and appear in fields such as natural language processing [31, 33, 34] and XML transformations [32]. 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.

In order to overcome this limitation, *symbolic tree automata* (STAs) and *symbolic tree transducers* (STTs) extend these classical objects by allowing transitions to be labeled with formulas in a specified theory. While the concept is straightforward, traditional algorithms for deciding composition, equivalence, and other properties of finite automata and transducers *do not* immediately generalize. A notable example appears in [9] where it is shown that while in the classical 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 classical tree automata [39] 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 *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 [22]. In the case of finite alphabets this problem is solved by augmenting the transducer's rules with regular lookahead [15], 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 *symbolic tree transducers with regular lookahead* (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 classical 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 frontend for STAs and STTRs. Fast (Functional Abstraction of Symbolic Transducers) is a functional language that integrates symbolic automata and transducers with Z3 [12], a state-of-the-art solver able to support complex theories that range from data-types to non-linear real arithmetic. We use Fast to model several real world scenarios and analysis problems: we demonstrate applications to HTML sanitization, interference checking of augmented reality applications submitted to an app store, deforestation in functional language compilation, and analysis of functional programs over trees. We also sketch how Fast can capture simple CSS analysis tasks. All 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 DSLs for tree manipulation.



*PLDI '14*, June 9 - 11 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594309

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

**Figure 1:** Representative applications of FAST discussed in Section 5. For each application we show which analyses of FAST are needed.

**Contributions summary:**

1. a *theory of symbolic tree transducers with regular lookahead* (STTR), that non-trivially extends the classical theory of tree transducers (§3);
2. a new algorithm for composing STTRs together with a proof of correctness (§4);
3. FAST, a domain-specific language for tree manipulations founded on the theory of STTRs (§3); and
4. five concrete applications of 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. 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 usually the following: 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 we us 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 of it in which 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 `"\"`.

We informally describe each component of Figure 2. Line 2 defines the data-type *HtmlE* of our trees.[2] Each node of type *HtmlE* contains a *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 classical 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) $x_1$

[1] Some sanitizers process the input HTML as a string, often causing the output not to be standards compliant.

[2] Section 6 discusses why classical tree transducers do not scale in this case.

```
1  // Datatype definition for HTML encoding
2  type HtmlE[tag : String]{nil(0), val(1), attr(2), node(3)}
3  // Language of well-formed HTML trees
4  lang nodeTree:HtmlE {
5        node(x1, x2, x3) given
6                    (attrTree x1) (nodeTree x2) (nodeTree x3)
7       |nil() where (tag = "")   }
8  lang attrTree:HtmlE {
9        attr(x1, x2) given (valTree x1) (attrTree x2)
10      |nil() where (tag = "")   }
11 lang valTree:HtmlE {
12       val(x1) where (tag ≠ "") given (valTree x1)
13      |nil() where (tag = "")   }
14 // Sanitization functions
15 trans remScript:HtmlE->HtmlE {
16       node(x1, x2, x3) where (tag ≠ "script")
17             to (node [tag] x1 (remScript x2) (remScript x3))
18      |node(x1, x2, x3) where (tag = "script") to x3
19      |nil() to (nil [tag])   }
20 trans esc:HtmlE->HtmlE {
21       node(x1, x2, x3) to (node [tag] (esc x1) (esc x2) (esc x3))
22      |attr(x1, x2) to (attr [tag] (esc x1) (esc x2))
23      |val(x1) where (tag = "'" ∨ tag = """)
24             to (val ["\"](val [tag] (esc x1)))
25      |val(x1) where (tag ≠ "'" ∧ tag ≠ """)
26             to (val [tag] (esc x1))
27      |nil() to (nil [tag])   }
28 // Compose remScript and esc, restrict to well-formed trees
29 def rem_esc:HtmlE->HtmlE := (compose remScript esc)
30 def sani:HtmlE->HtmlE := (restrict rem_esc nodeTree)
31 // Language of bad outputs that contain a "script" node
32 lang badOutput:HtmlE {
33       node(x1, x2, x3) where (tag = "script")
34      |node(x1, x2, x3) given (badOutput x2)
35      |node(x1, x2, x3) given (badOutput x3)   }
36 // Check that no input produces a bad output
37 def bad_inputs:HtmlE := (pre-image sani badOutput)
38 assert-true (is-empty bad_inputs)
```

**Figure 2:** Implementation and analysis of an HTML sanitizer in FAST.

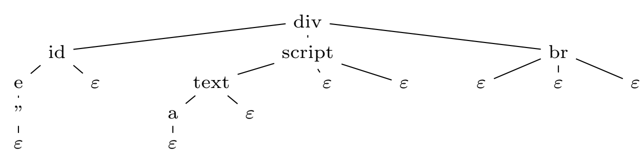


**Figure 3:** *HtmlE* encoding of the HTML tree `<div id='e"'><script>a</script></div><br />`. `div`, `script`, and `br` are built using the constructor *node*. Nodes labeled with `id`, and `text`, are built using *attr*. Single character nodes are built using *val*, and $\varepsilon$'s using *nil*. 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 />`.

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 non-empty string $w = s_1 \ldots s_n$ is encoded as an *HtmlE* element $val(x_1)$ where *tag* contains the string "$s_1$", and $x_1$ encodes the suffix $s_2 \ldots s_n$. Each element *nil* has *tag* `""`, and can be seen as a termination operator for lists, strings, and trees. This encoding can be expressed in FAST (lines 4-13). For example, *nodeTree* (lines 4-7) 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 15-19) 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* (lines 20-27) 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 pre-pended a node *val* with tag `"\"`. The transformations *remScript* and *esc* are then composed into a single transformation *rem_esc* (line 29). One might notice that *rem_esc* also accepts input trees that are not in the language *nodeTree*, and therefore do not correspond to correct encodings. Therefore, we compute the transformation *sani* (line 31), 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 33-36), which accepts all the trees containing at least one node labeled with `"script"`.[3] Next, using transducers composition, we compute the language *bad_inputs* (line 38) 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 40) 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 18, where the rule does not recursively invoke the transformation *remScript* on $x_3$. When 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 *symbolic tree automata* (*STAs*), and *symbolic tree transducers with regular lookahead* (*STTRs*). This section describes 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 $\lambda$-expressions for defining anonymous functions called $\lambda$-*terms* without having to name them explicitly. In general, we use standard first-order logic and follow the notational conventions that are consistent with [40]. We write $\sigma$ for a type and the universe of elements of type $\sigma$ is denoted by $\boldsymbol{\sigma}$. A $\sigma$-predicate is a $\lambda$-term $\lambda x.\varphi(x)$ where $x$ has type $\sigma$, and $\varphi$ is a formula whose free variables $FV(\varphi)$ are contained in $\{x\}$. Given a $\sigma$-predicate $\varphi$, $[\![\varphi]\!]$ denotes the set of all $a \in \boldsymbol{\sigma}$ such that $\varphi(a)$ holds. The set of $\sigma$-predicates

[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.

```
Indentifiers            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)
```

**Figure 4:** Concrete syntax of FAST. Nonterminals and meta-symbols are in italic. Constant expressions for strings and numbers use C# syntax [24]. Additional well-formedness conditions (such as well-typed terms) are assumed to hold.

is denoted by $\Psi(\sigma)$. Given a type $\sigma$ (such as *int*), we extend the universe with $\sigma$-labeled finite trees as an algebraic datatype $\mathcal{T}^\sigma_\Sigma$ where $\Sigma$ is a finite set of *tree constructors* $f$ with *rank* $\natural(f) \geq 0$; $f$ has type $\sigma \times (\mathcal{T}^\sigma_\Sigma)^{\natural(f)} \to \mathcal{T}^\sigma_\Sigma$.[4] Let $\Sigma(k) \stackrel{\text{def}}{=} \{f \in \Sigma \mid \natural(f) = k\}$. We require that $\Sigma(0)$ is *non-empty* so that $\boldsymbol{\mathcal{T}^\sigma_\Sigma}$ is non-empty. We write $f[t](\bar{u})$ for $f(t, \bar{u})$ and abbreviate $f[t]()$ by $f[t]$.

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

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

### 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 [39].

**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 *state*, $f$ the *symbol*, $\varphi$ the *guard*, and $\bar{\ell}$ the *lookahead*.

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

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

**Example 2.** Consider the following FAST program.

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

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

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

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

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. ⊠

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

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

Each subtree lookahead $\ell_i$ above 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 $\mathbf{q} \subseteq Q$:

$$\mathbf{L}^{\mathbf{q}}_A \stackrel{\text{def}}{=} (\bigcap_{q \in \mathbf{q}} \mathbf{L}^q_A,\ \text{if } \mathbf{q} \neq \emptyset; \quad \boldsymbol{\mathcal{T}^{\sigma}_{\Sigma}},\ \text{otherwise.})$$

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 $i$, $1 \leq i \leq \natural(f)$, $\ell_i$ is a singleton set.

For example, the STA in Example 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 $A = (Q, \mathcal{T}^{\sigma}_{\Sigma}, \delta)$ be an STA. We compute *merged rules* $(\boldsymbol{q}, f, \boldsymbol{\varphi}, \bar{\boldsymbol{\rho}})$ 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:

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

where *merge* $\wedge\!\!\!\wedge$ of rules is defined as follows:

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

We can then define *Normalize*$(A)$ as the STA

$$\left(2^Q, \mathcal{T}^{\sigma}_{\Sigma}, \{(\boldsymbol{p}, f, \bigwedge \boldsymbol{\varphi}, (\{\boldsymbol{q}_i\})_{i=1}^{\natural(f)}) \mid f \in \Sigma, (\boldsymbol{p}, f, \boldsymbol{\varphi}, \bar{\boldsymbol{q}}) \in \boldsymbol{\delta}^f\}\right)$$

where the original rules are precisely the ones whose states are singleton sets in $2^Q$.[6] Checking whether $\mathbf{L}^q_A \neq \emptyset$ can be done by first normalizing $A$, then removing unsatisfiable guards using the decision procedure of the theory $\Psi(\sigma)$, and finally using emptiness of classical tree automata.

**Proposition 1.** *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

[5] When compared to the model in [8], 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 lookahead of a rule $r$ corresponds to a *conjunction* of states, while several rules from the same source state provide a *disjunction* of cases.

[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, e.g., by using elimination of useless symbols from a context-free grammar [26, p. 88–89].

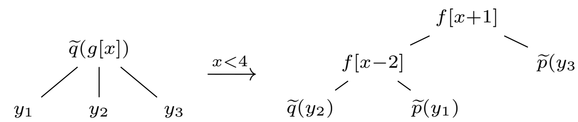


**Figure 5:** A depiction of a linear rule of rank 3.

STA. This is true already for the *classical* case, i.e., when $\boldsymbol{\sigma} = \{()\}$. Using the intersection non-emptiness problem of classical tree automata [19, 37], and emptiness of alternating tree automata [8] we have the following bound.

**Proposition 2.** *The non-emptiness problem of alternating STAs without attributes is* ExpTime*-complete.*

We decided to use alternating STAs because they are succinct and arise naturally when composing tree transducers.

### 3.3 Symbolic Tree Transducers with Regular Lookahead

Symbolic tree transducers (STTs) augment STAs with outputs. Symbolic tree transducers with regular lookahead 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 $\mathcal{T}^{\sigma}_{\Sigma}$ 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 that $\mathcal{T}^{\sigma}_{\Sigma}$ is a combined tree type that covers both. This assumption avoids a lot of cumbersome overhead of type annotations and can be made without loss of generality because we have *partial* definitions. The guards and the lookaheads can be use to restrict the types as needed.

The set of *extended tree terms* is the set of tree terms of type $\mathcal{T}^{\sigma}_{\Sigma \cup \{State\}}$ 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 4.** Given a tree type $\mathcal{T}^{\sigma}_{\Sigma}$, a finite set $Q \subseteq \boldsymbol{\sigma}$ of states, and $k \geq 0$, the set $\Lambda(\mathcal{T}^{\sigma}_{\Sigma}, Q, k)$ is defined as the least set $T$ of $\lambda$-terms called *$k$-rank tree transformers* that satisfies the following conditions, let $\bar{y}$ be a $k$-tuple of variables of type $\mathcal{T}^{\sigma}_{\Sigma \cup \{State\}}$ and let $x$ be a variable of type $\sigma$,

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

**Definition 5.** A *Symbolic Tree Transducer with Regular lookahead (STTR)* $S$ 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 $\lambda(x, \bar{y}).u$ where each $y_i$ occurs at most once in $u$. $S$ is *linear* when all rules of $S$ 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 lookahead, when omitted, is $\bar{\emptyset}$ by default. Figure 5 illustrates an open view of a linear rule.

[7] For $k = 0$ we assume that $(2^Q)^k = \{()\}$, 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.

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

**Definition 6.** The *domain automaton of* $S$, $\mathbf{d}(S)$, 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 lookahead states. For example, the rule in Figure 5 yields the domain automaton rule $(q, g, \lambda x.x < 4, (\{p\}, \{q\}, \{p\}))$.

In the following let $T$ be the STTR and let $\mathbf{L}^{\ell}_T \stackrel{\text{def}}{=} \mathbf{L}^{\ell}_{\mathbf{d}(T)}$.

**Definition 7.** For all $q \in Q_T$, the *transduction of* $T$ *at* $q$ is the function $\mathbf{T}^q_T$ such that, for all $t = f[a](\bar{t}) \in \boldsymbol{\mathcal{T}^{\sigma}_{\Sigma}}$,

$$\mathbf{T}^q_T(t) \stackrel{\text{def}}{=} \Downarrow_T \widetilde{q}(t)$$

$$\Downarrow_T \widetilde{q}(t) \stackrel{\text{def}}{=} \bigcup \{\Downarrow_T u(a, \bar{t}) \mid (q, f, \varphi, \bar{\ell}, u) \in \Delta_T, a \in [\![\varphi]\!], \bigwedge_{i=1}^{\natural(f)} t_i \in \mathbf{L}^{\ell_i}_T\}$$

$$\Downarrow_T(t) \stackrel{\text{def}}{=} \{f[a](\bar{v}) \mid \bigwedge_{i=1}^{\natural(f)} v_i \in \Downarrow_T(t_i)\}$$

The *transduction of* $T$ is $\mathbf{T}_T \stackrel{\text{def}}{=} \mathbf{T}^{q^0}_T$. The definitions are lifted to sets using *union*. We write $\mathbf{T}_T(t, u)$ for $u \in \mathbf{T}_T(t)$.

We omit $T$ from $\mathbf{T}^q_T$ and $\Downarrow_T$ when $T$ is clear from the context. 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 lookahead $(\{r \mid (r\ y_i) \in \ell(\bar{y})\})_{i=1}^{\natural(f)}$.

**Example 3.** Recall the transformation *remScript* in Figure 2. These are the corresponding rules. We use $q$ for the state of *remScript*, and $\imath$ for a state that outputs the identity transformation. The "safe" case is

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

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

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

**Definition 8.** $S$ is *single-valued* if $\forall (t \in \boldsymbol{\mathcal{T}^{\sigma}_{\Sigma}}, q \in Q_S) : |\mathbf{T}^q_S(t)| \le 1$.

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 input tree. In contrast, decidability of single-valuedness of STTRs is an open problem.

**Definition 9.** $S$ 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 $\Delta_S$, 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 4.** 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 label *true*. Given an input $t$, $s_2$ always outputs $L[true]$. Both transductions are definable using STTs since they do not use lookahead. Now consider the composed transduction $s = s_1 \circ s_2$ that outputs $L[true]$ iff all the nodes in $t$ have label *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 $false$ labels. However, $s$ can be computed using an STTR that checks that both $x$ and $y$ contain only *true* labels. ⊠

The next example shows how STTRs are simpler than STTs.

**Example 5.** 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]  }
```

This function can be expressed using a nondeterministic STT that guesses if the label 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, complement, etc.:** operations over STAs;
- **domain** $t$**:** computes the domain of the STTR $t$ using the operation from Definition 6; and
- **pre-image** $t$ $l$**:** computes an STA accepting all the inputs for which $t$ produces an output belonging to $l$.

*Operations that compute new transformations:*

- **restrict** $t$ $l$**:** constructs a new STTR that behaves like $t$, but is only defined on the inputs that belong to $l$;
- **restrict-out** $t$ $l$**:** constructs a new STTR that behaves like $t$, but is only defined on the inputs for which $t$ produces an output that belongs to $l$; and
- **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:*

- $a \in l$, $l_1 = l_2$, **is-empty:** decision procedures for STAs;
- **type-check** $l_1$ $t$ $l_2$**:** true iff for every input in $l_1$, $t$ only produces outputs in $l_2$.

Several operations are special applications of composition. For example **restrict-out** $q$ $p$ = **compose** $q$ (**restrict** $I$ $p$), where $I$ is the identity STTR.

## 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 4 and in [22], 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 classical tree transducers and can be avoided either by considering restricted fragments, or by instead adding regular lookahead [2, 14, 16]. In this paper we consider the latter option. Intuitively, regular lookahead 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 6.** Let $f$ be the function that, given a tree of type `BT` (see Example 2) transforms it by nondeterministically replacing some leaves with the value 5. Let $g$ be the function that transforms a tree $t$ into $N[0](t,t)$. So $g(f(L[1]))$ 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 lookahead have been studied extensively [20]. However, as shown in [22], extending classical 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 classical tree transducers, apart from the complexity of the label theory itself, is the *output computation*. In symbolic transducers the output labels depend *symbolically* on the input label. Effectively, this breaks the application of some well-established classical techniques that no longer carry over to the symbolic setting. For example, while for classical 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 states 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 the relation $\exists y(\mathbf{T}_S(x,y) \circ \mathbf{T}_T(y,z))$, following the convention in [21].

For $p \in Q_S$ and $q \in Q_T$, assume that '.' is an injective pairing function that constructs a new pair state $p.q \notin Q_S \cup Q_T$. In a nutshell, we use a least fixed point construction starting with the initial state $q_S^0.q_T^0$. 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

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

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

**Example 7.** 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 $p.q$ has been reached. Then

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

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

The rewriting steps are done modulo label constraints. To this end, a *k-configuration* is a triple $(\gamma, L, u)$ where $\gamma$ is a formula with $FV(\gamma) \subseteq \{x\}$, $L$ is a $k$-tuple of sets of pair states $p.q$ where $p \in Q_S$ and $q \in Q_T$, and $u$ is an extended tree term. We use configurations to describe reductions of $T$. Composition of $S$ and $T$ is defined formally as follows

$$\begin{aligned} S{\circ}T \;\; &\stackrel{\text{def}}{=} \;\; (Q_S \cup \{p.q \mid p \in Q_S, q \in Q_T\}, q_S^0.q_T^0, \mathcal{T}_\Sigma^\sigma, \\ &\quad \Delta_S \cup \bigcup_{p\in Q_S, q \in Q_T, f\in\Sigma} \mathbf{Compose}(p,q,f)) \end{aligned}$$

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.

$\mathbf{Compose}(p,q,f) \stackrel{\text{def}}{=}$

1. ***choose*** $(p, f, \varphi, \bar{\ell}, u)$ ***from*** $\Delta_S$
2. ***choose*** $(\psi, \bar{P}, t)$ ***from*** $\mathbf{Reduce}(\varphi, (\emptyset)_{i=1}^{\natural(f)}, \widetilde{q}(u))$
3. ***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 a label formula $\varphi$ with $FV(\varphi) \subseteq \{x\}$, a composed lookahead $L$ of rank $k$, a state $q \in Q_T$, and a term $t$ including states from $Q_S$, returns all possible extended contexts and lookaheads. Assume, without loss of generality, that $\mathbf{d}(T)$ is *normalized*. We let $\varepsilon(\{e\}) \stackrel{\text{def}}{=} e$ for any singleton set $\{e\}$.

$\mathbf{Look}(\varphi, L, q, t) \stackrel{\text{def}}{=}$

1. ***if*** $t = \widetilde{p}(y_i)$ ***where*** $p \in Q_S$ ***then return*** $(\varphi, L \uplus_i \{p.q\})$
2. ***if*** $t = g[u_0](\bar{u})$ ***where*** $g \in \Sigma$ ***then***
   - (a) ***choose*** $(q, g, \psi, \bar{\ell})$ ***from*** $\delta_{\mathbf{d}(T)}$ ***where*** $\mathit{IsSat}(\varphi \wedge \psi(u_0))$
   - (b) $L_0 := L$, $\varphi_0 := \varphi \wedge \psi(u_0)$
   - (c) ***for*** $(i = 1; i \leq \natural(g); i{+}{+})$

     ***choose*** $(\varphi_i, L_i)$ ***from*** $\mathbf{Look}(\varphi_{i-1}, L_{i-1}, \varepsilon(\ell_i), u_i)$
   - (d) ***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 labels, which are not expressible by classical tree transducers.

**Example 8.** 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 $(q, g, \lambda x.\mathit{odd}(x), \{q\}) \in \delta_{\mathbf{d}(T)}$ that requires that all labels 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 $\widetilde{p}(f[x](y_1,y_2)) \to g[x{+}1](g[x{-}2](\widetilde{p_1}(y_2)))$ of $S$. Clearly, this outrules $t$ as a valid input of $T$ at $q$ because of the cross-level dependency between labels due to $x$, implying that both labels 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 $\mathbf{Look}(x{>}0 \wedge \mathit{odd}(x{+}1), \bar{\emptyset}, q, g[x{-}2](\widetilde{p_1}(y_2)))$. Inside the recursive call we have the failing satisfiability check of $IsSat(x>0 \wedge odd(x+1) \wedge odd(x-2))$ 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 $q_\tau$ that uniquely identifies the rule $(q_\tau, f, \varphi, \bar{\ell}, t)$; $q_\tau$ is used to refer to the guard and the lookahead of $\tau$ chosen in line 2(a) in the call to **Look** in 2(b) below, $q_\tau$ is not used elsewhere.

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

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

There is a close relationship between **Reduce** and Definition 7. We include the case

$$\mathbf{T}_T^q(\widetilde{p}(t)) \stackrel{\text{def}}{=} \mathbf{T}_T^q(\mathbf{T}_S^p(t)) \quad \text{for } p \in Q_S \text{ and } t \in \mathcal{T}_{\boldsymbol{\Sigma}}^{\boldsymbol{\sigma}}, \qquad (1)$$

that allows states of $S$ to occur in the input trees to $\mathbf{T}_T^q$ 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 (1) we need the following Lemma.

**Lemma 3.** *For all* $t \in \Lambda(\mathcal{T}_\Sigma^\sigma, Q_S, k)$, $\mathtt{a} \in \boldsymbol{\sigma}$, *and* $\mathtt{u}_i \in \mathcal{T}_{\boldsymbol{\Sigma}}^{\boldsymbol{\sigma}}$:

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

**Example 9.** 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 $q \xrightarrow[T]{g,\top} \lambda xy.f[x](\widetilde{q}(y), \widetilde{q}(y))$. Let $\mathtt{f} = f[0]$, $\mathtt{c} = c[0]$, $\mathtt{g} = g[0]$. Then

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

but

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

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

The assumptions on $S$ and $T$ given in Lemma 3 are the same as in the classical setting, however the proof of Lemma 3 does not directly follow from classical results because either the concrete alphabet $\Sigma \times \boldsymbol{\sigma}$ is infinite, or else, if $\boldsymbol{\sigma}$ is encoded as trees, linear rules become nonlinear in the classical sense, such as the rule in Figure 5.

Theorem 4 uses Lemma 3. 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.** *For all* $p \in Q_S$, $q \in Q_T$ *and* $t \in \mathcal{T}_{\boldsymbol{\Sigma}}^{\boldsymbol{\sigma}}$, $\mathbf{T}_{S \circ T}^{p.q}(t) \supseteq \mathbf{T}_T^q(\mathbf{T}_S^p(t))$, *and if $S$ is single-valued or if $T$ is linear then* $\mathbf{T}_{S \circ T}^{p.q}(t) \subseteq \mathbf{T}_T^q(\mathbf{T}_S^p(t))$.

## 5. Evaluation

FAST can be applied 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, HTML_Safe, 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

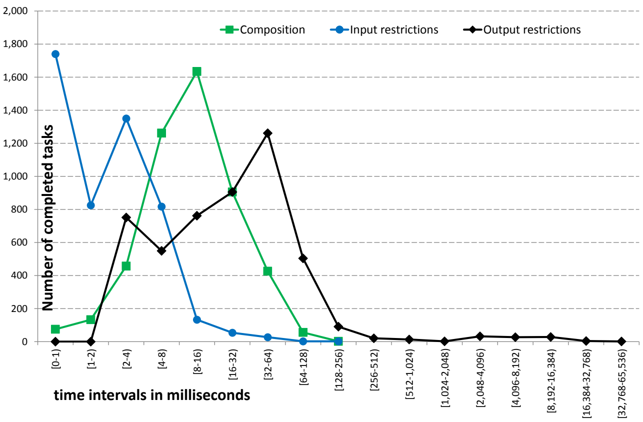


**Figure 6:** Augmented reality: running times for operations on transducers. The $x$-axis represent time intervals in *ms*. The $y$-axis shows how many cases run in a time belonging to an interval. For example about 1,600 compositions were completed between 8 and 16 ms.

of maintainability, FAST wins on two counts. First, we can apply analysis to FAST programs that is precise, unlike analyses for PHP. 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 (often called taggers) on the Layar phone AR platform applications 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 label 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 only accepts 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 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. 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 [41], 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 [41] 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 $(x+5)\%26$. 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. We run the experiments on lists containing 4,000 randomly generated elements and we consider up to 512 composed functions. Figure 7 shows the running time with and without deforestation for a list of 4,096 integers used as the input. The running time of the FAST composed version is almost unchanged, even for 512 compositions while

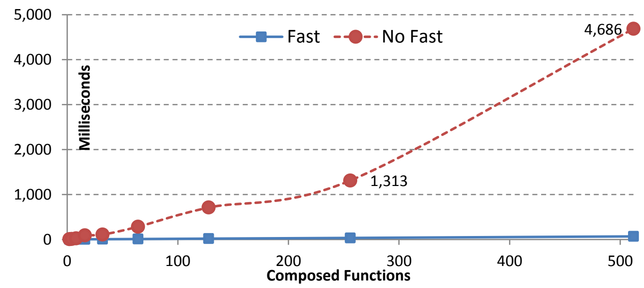


**Figure 7:** Deforestation advantage for a list of 4,096 integers.

```
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)
```

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

the running time of the naïvely composed functions degrades linearly in the number of composed functions.

### 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 *comp*2 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 *comp*2 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.

### 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 [23] 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 Classical Tree Transducers

As we mentioned in the previous section, the HTML sanitization and CSS analysis problems could, in principle, be expressed using existing classical 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 classical 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 classical transducers does not scale in this case.

The HTML sanitization example illustrates some core differences between the symbolic and the classical case. In some respect, the situation is analogous to going from SAT to SMT solving [13], 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 $\Sigma$ 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, $\Sigma$ 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 [38], 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 [12]), 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* ≠ `"script"` can be represented by the *complement* $A^c$ of $A$. Observe that complementation of classical 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 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 blowup compared to an equivalent classical transducer, as demonstrated by the symbolic word transducer implementing UTF8 encoding in [10]. 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 [8, 21, 35]. The first models were top-down and bottom-up tree transducers [2, 14], later extended to top-down transducers with regular lookahead in order to achieve closure under composition [15, 16, 20]. Extended top-down tree transducers [31] (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 lookahead such a model is equivalent to top-down tree transducers with regular lookahead. More complex models, such as *macro tree transducers* [17], and *streaming tree transducers* [1] 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 [25] 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 [40]. Variants of SFTs in which multiple input symbols can be read by a single transition are studied in [9] and in [5]. Symbolic tree transducers are originally introduced in [39], where it is wrongly claimed that STTs are closed under composition by referring to a generalization of a proof of the classical case in [21] which is only stated for total deterministic finite tree transducers. In [22] this error is discovered and other properties of STTs are investigated. The main result of [39] is an algorithm for checking equivalence of single-valued linear STTs. For classical transducers, equivalence has been shown to be decidable for deterministic or finite-valued tree transducers [36], streaming tree transducers [1], and MSO tree transformations [18]. 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 [30] 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 [34] and Tiburon [33], 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 transducers algorithms. Both the languages are limited to finite input and output alphabets. ASF+SDF [7] 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 [42]), stream processing (STX [3]), and manipulating (XSLT, XDuce [27]) XML trees. While being very expressive, these languages support very limited forms of analysis. Emptiness has been shown decidable for restricted fragments of XPath [4]. XDuce [27] 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 1 summarizes the relations between FAST and the other domain-specific languages for tree transformations.

| Language | $\sigma$ | Analysis | Domain |
|---|---|---|---|
| FAST | $\infty$ | composition; typechecking, pre-image, 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 | $\infty$ | - | Parsing |
| XPath | $\infty$ | emptiness for a fragment | XML query (only selection) |
| XDuce | $\infty$ | type-checking for navigational part (finite alphabet) | XML query |
| XQuery, XSLT, STX | $\infty$ | - | XML transformations |

**Table 1:** Summary of main domain specific languages for tree-manipulating programs and their properties; $\boldsymbol{\sigma}$ indicates whether the language supports finite (ff) or infinite ($\infty$) alphabets.

**Applications.** The connection between tree transducers and deforestation was first investigated in [41], and then further investigated in [29]. In this setting deforestation is done via *Macro Tree Transducers* (MTT) [17]. While being more expressive than Top Down Transducers with regular lookahead, MTTs only support finite alphabets and their composition is very expensive. We are not aware of an actual implementation of the techniques in [29]. Higher-Order Multi-Parameter Tree Transducers (HMTT) [28] 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 non-emptiness problem of a set of normalized STAs as the emptiness of a single unnormalized STA. In the classical case, the non-emptiness problem of tree automata is P-CO, while the intersection non-emptiness problem is EXPTIME-CO [8, Thm 1.7.5]. Recently, new techniques based on antichains have been proposed to check universality and inclusion for nondeterministic tree automata [6]. 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 [36]. Novel algorithms for minimizing and learning symbolic automata over lists have been recently proposed in [11] and [5]. 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 lookahead 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


[1] R. Alur and L. D'Antoni. Streaming tree transducers. In *ICALP'12*, pages 42–53. Springer, 2012.

[2] B. S. Baker. Composition of top-down and bottom-up tree transductions. *Inform. and Control*, 41:186–213, 1979.

[3] O. Becker. Streaming transformations for XML-STX. In R. Eckstein and R. Tolksdorf, editors, *XMIDX 2003*, volume 24 of *LNI*, pages 83–88. GI, 2003.

[4] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. In *PODS'06*, pages 10–19, 2006.

[5] M. Botinčan and D. Babić. Sigma*: symbolic learning of input-output specifications. In *POPL'13*, pages 443–456, New York, NY, USA, 2013. ACM.

[6] A. Bouajjani, P. Habermehl, L. Holik, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA'08*, volume 5148 of *Lecture Notes in Computer Science*, pages 57–67. Springer Berlin Heidelberg, 2008.

[7] M. Brand, J. Heering, P. Klint, and P. Olivier. Compiling Rewrite Systems: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4), 2002.

[8] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

[9] L. D'Antoni and M. Veanes. Equivalence of extended symbolic finite transducers. In *CAV 2013*, volume 8044 of *LNCS*, pages 624–639. Springer, 2013.

[10] L. D'Antoni and M. Veanes. Static analysis of string encoders and decoders. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI 2013*, volume 7737 of *LNCS*, pages 209–228. Springer, 2013.

[11] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 541–554. ACM, 2014.

[12] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS, 2008.

[13] L. de Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction & Applications. *Communications of the ACM*, 54(9):69–77, 2011.

[14] J. Engelfriet. Bottom-up and top-down tree transformations – a comparison. *Math. Systems Theory*, 9:198–231, 1975.

[15] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Math. Systems Theory*, 10:289–303, 1977.

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

[17] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput*, 154:34–91, 1998.

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

[19] T. W. Frühwirth, E. Y. Shapiro, M. Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *LICS'91*, pages 300–309, 1991.

[20] Z. Fülöp and S. Vágvölgyi. Variants of top-down tree transducers with look-ahead. *Math. Sys. Th.*, 21(3):125–145, 1989.

[21] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers.* EATCS. Springer, 1998.

[22] Z. Fülöp and H. Vogler. Forward and backward application of symbolic tree transducers. *CoRR*, abs/1208.5324, 2012.

[23] P. Geneves, N. Layaida, and V. Quint. On the analysis of cascading style sheets. In *WWW '12*, pages 809–818, New York, NY, USA, 2012. ACM.

[24] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[25] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *Proceedings of the USENIX Security Symposium*, 2011.

[26] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Longman Publishing Co., Inc., 1979.

[27] H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003.

[28] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL'10*, pages 495–508, 2010.

[29] A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In *Fuji Int. Symp. on Functional and Logic Programming*, 1999.

[30] O. Lengal, J. Šimáček, and T. Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *TACAS'12*, volume 7214 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, 2012.

[31] A. Maletti, J. Graehl, M. Hopkins, and K. Knight. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39:410–430, June 2009.

[32] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *PODS'05*, pages 283–294, New York, NY, USA, 2005. ACM.

[33] J. May and K. Knight. A primer on tree automata software for natural language processing, 2008.

[34] A. Purtee and L. Schubert. TTT: A tree transduction language for syntactic and semantic processing. In *Proceedings of the Workshop on App. of Tree Aut. Tech. in NLP*, 2012.

[35] J.-C. Raoult. A survey of tree transductions. In *Tree Automata and Languages*, pages 311–326. sn, 1992.

[36] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Math. Systems Theory*, 27:285–346, 1994.

[37] H. Seidl. Haskell overloading is dexptime-complete. *Inf. Process. Lett.*, 52(2):57–60, 1994.

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

[39] M. Veanes and N. Bjørner. Symbolic tree transducers. In *Perspectives of System Informatics (PSI'11)*, volume 7162 of *LNCS*, pages 377–393. Springer, 2011.

[40] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *POPL'12*, 2012.

[41] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, 1988.

[42] P. Walmsley. *XQuery.* O'Reilly Media, Inc., 2007.