Abstract
We introduce a tree manipulation language, FAST, that overcomes technical limitations of previous tree manipulation languages, such as XPath and XSLT which do not support precise program analysis, or TTT and Tiburon which only support trees over finite alphabets. At the heart of FAST is a combination of SMT solvers and tree transducers, enabling it to model programs whose input and output can range over any decidable theory. The language can express multiple applications. We write an HTML "sanitizer" in FAST or C, in the theories, no changes to our algorithms are required. Unabstract approach is a SMT solver, and implement our framework. Our chosen approach is symbolic tree transducers with regular lookahead (STTRs): if the solver adds support for new theories, no changes to our algorithms are required.

1. Introduction
Tree manipulations are common, found in XML processing, compiler optimization, and natural language processing. As a result, multiple domain-specific languages exist for manipulating trees, including XPath, XSLT, TTT, and Tiburon [13, 14, 21, 22]. Unfortunately, these languages are either (1) limited in expressiveness, such as restricted to finite alphabets, or (2) too expressive for precise analysis. For example, even though used in modern browsers, XSLT is Turing-complete, and analysis of general XSLT programs must be approximate.

We thread the needle between these two issues with a domain-specific language, FAST, whose semantics is given by Symbolic Tree Transducers with Regular look-ahead (STTR). These are generalizations of classic tree transducers that take advantage of Satisfiability Modulo Theories (SMT) solvers. STTRs enjoy several closure properties and are able to compute the output corresponding to a given input tree in a single pass and without producing intermediate results. We use STTRs in place of their variant without regular look-ahead (STT) for two reasons: 1) most classes of STTRs are closed under composition, while this is not the case for STTs, and 2) STTRs can model more complex programs than STTs without sacrificing precise analysis. All the properties of STTRs are inherited by FAST. In particular, the input and output alphabets of FAST programs can range over any decidable theory. To the best of our knowledge, FAST is the first language for tree manipulations that is able to model programs over infinite alphabets while preserving decidable properties.

To support our analyses, we extend the classical results on tree transducers. We make use of Z3, a state-of-the-art SMT solver, and implement our framework. Our chosen approach is fully abstract: if the solver adds support for new theories, no changes to our algorithms are required. Unlike general-purpose programs in a language such as Java or C, in FAST certain important safety properties are decidable. For example, program equivalence is undecidable in the general case, but advanced forms of type-checking are decidable. We have implemented our analyses and found that they run quickly in practice. After completing the analyses, FAST compiles to C# and can be used in multiple practical scenarios.

Our work combines theoretical foundations with solutions to pragmatic problems. Our fundamental results enable powerful applications, both in the short and long term. We demonstrate applications to security, interference checking, and deforestation in functional language compilation. In all such cases, the use of symbolic input and output alphabets is needed to model real programs. Additionally, FAST can be used to capture string-manipulating programs, as previously described in the Bek project [8]. We also sketch how FAST captures classic techniques in machine learning and compiler optimization. Figure 1 summarizes our applications and the analyses enabling each one. We are not aware of another domain-specific language for tree manipulations able to represent programs in such different domains. Current domain-specific languages for tree manipulations are suited either to XML processing [21, 22] or to natural language processing [13, 14].

In summary we offer the following contributions:

- **STTRs**: We introduce symbolic tree transducers with regular lookahead (STTRs) as a natural extension of top-down tree transducers with regular lookahead (Section 3) and develop the theory for closure under composition (Section 4). We prove that natural extensions of the theorems to the symbolic case still hold.

- **FAST**: We introduce FAST (Section 3), a domain-specific language for tree manipulations based on STTRs. FAST has the following properties:
  - it can express a broad class of tree-manipulating programs (like HTML sanitizers);
  - it allows static checking of program properties, such as whether a given program can produce a particular output tree;
  - it allows the use of any theory supported by the SMT solver Z3. This way FAST can be extended whenever a new theory is introduced;
  - it allows complex programs to be written in a modular way without worrying about efficiency (i.e. how many times the input tree needs to be traversed); a composition of FAST programs can be efficiently transformed...
Fast programs can be compiled into efficient C# code.

**Evaluation:** We experimentally evaluate Fast in three settings (Section 5): (1) HTML sanitization: to show how complex real-world applications can be modeled in Fast; (2) AR app store: to show how Fast can efficiently check program properties statically; (3) Deforestation: to show how Fast enables efficient code generation in the presence of modular programs.

Finally, we describe related work (Section 6) and conclude (Section 7).

## 2. Motivating Example

We introduce the language Fast (Functional Abstraction of Symbolic Transducers) through a simple example and then illustrate the advantages of the language.

We start by showing a simple Fast implementation of a basic HTML sanitizer. The main purpose of an HTML sanitizer is removal of malicious active code from an HTML document. Another possible goal is modifying the input HTML to make it standards compliant. To achieve these goals, a sanitizer traverses the HTML document and removes or modifies nodes, attributes and values that can cause malicious code to be executed. 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);
2. The DOM is modified by a sequence of sanitization functions \( f_1, \ldots, f_n \);
3. The modified DOM is transformed back into an HTML document.

Some sanitizers do not compute the DOM and process the input HTML as a string. This approach usually generates an ill-formed HTML output (not standards compliant).

In the following we concentrate on describing some of the functions used during step 2. Each function \( f \) takes as input a DOM tree and transforms it into an updated DOM tree. As an example, the Fast program \( \text{sani} \) in Figure 2 traverses the input DOM and outputs a copy of it in which all subtreese whose root is labeled with "script" have been removed, and all the characters "\n" and "\" have been escaped with a "\\".

Next, we informally describe each component of the example of Figure 2. Fast is a functional language which is restricted to work on tree structures. In this example trees are defined using the type HtmlE. Each node of type HtmlE contains a tag of type string and is assigned 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. However, since DOM trees are unranked (each node can have an arbitrary number of children), we will need an encoding to represent them as trees of type HtmlE. An example of such encoding is depicted in Figure 3. The next paragraphs describe the encoding in detail.

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 \( n \) and can be either of type attr if \( n \) has at least one attribute, or type nil otherwise; 2) \(x_2\) encodes the first child of \( n \) in the DOM and can be either of type node if \( n \) has at least one child, or type nil otherwise; 3) \(x_3\) encodes the next sibling of \( n \) in the DOM and it can be either of type node if \( n \) has a sibling, or type nil otherwise.

Each HTML attribute \( a \) is encoded as an HtmlE element attr\((x_1,x_2)\) with two children \(x_1,x_2\) where: 1) \(x_1\) encodes the string value \( s \) of the attribute \( a \) and can be either of type val if \( s \) is not the empty string, or type nil otherwise; 2) \(x_2\) encodes the attribute following \( a \) in the DOM and it can be either of type attr if \( a \) is followed by an attribute, or type nil otherwise.

Each string value \( w \) is encoded as an HtmlE element val\((x)\) with one child \( x \). If the value of \( w \) is \( s_1 \ldots s_n \) for some \( n > 0 \), then \( x \) encodes the suffix \( s_2 \ldots s_n \) and \( w \) contains the string \( s_1 \). \( x \) can be either of type val if \( n > 1 \), or type nil otherwise. Each element nil has tag "" and can be seen as a termination operator for lists and trees.

The restrictions that we just informally imposed on the tree structure are formalized in Figure 2. nodeTree de-
describes the language of correct HTML encodings: 1) the tree \[ \text{node}(x_1, x_2, x_3) \] is in the language \text{node}\text{Tree} if \[ x_1 \] is in the language \text{attr}\text{Tree}, \[ x_2 \] is in the language \text{node}\text{Tree}, and \[ x_3 \] is in the language \text{node}\text{Tree}; 2) the tree \[ \text{nil} \] is in \text{node}\text{Tree} if its tag contains \text{**}. The other languages are similar.

Next, we describe the sanitization functions. The transformation \text{remScript} of type \text{HtmlE}\rightarrow\text{HtmlE} takes an input tree \[ t \] of type \text{HtmlE} and produces an output tree \[ t' \] of type \text{HtmlE}. \text{remScript} recursively processes the input tree \[ t \] if \[ \text{node}(x_1, x_2, x_3) \] with tag \"script\", the output \[ t' \] is the result of invoking \text{remScript} on \[ x_3 \], otherwise if \[ \text{node}(x_1, x_2, x_3) \] with tag different from \"script\", the output \[ t' \] is a copy of \[ t \] in which \[ x_2 \] and \[ x_3 \] are replaced by the results of invoking \text{remScript} on \[ x_2 \] and \[ x_3 \] respectively; 3) if \[ t \] is \[ \text{nil} \] the output \[ t' \] is a copy of \[ t \]. The transformation \text{esc} of type \text{HtmlE}\rightarrow\text{HtmlE} escapes the characters \"'\" and \"\". \text{esc} outputs a copy of the input tree where each node \[ cval \] with tag \"*\" or \"**\" is prepended a node \[ val \] with tag \"\". The transformations \text{remScript} and \text{esc} are then composed into a single transformation \text{rem}\_\text{esc}. \text{rem}\_\text{esc} also accepts input trees that are not in the language \text{node}\text{Tree} and therefore do not correspond to correct DOM encodings. The last line of Figure 2, defines the transformation \text{sani} that is equivalent to \text{rem}\_\text{esc}, but restricted to only accept inputs in the language \text{node}\text{Tree}.

This example showed that in \text{Fast}, simple sanitization functions can be first coded independently and then composed. This approach simplifies the implementation of the final sanitizer making it less error-prone. As we will see in Section 3, unlike in other programming languages, the composed function processes the input tree in a single pass and without producing intermediate results, making the \text{Fast} implementation efficient. Such function can be finally compiled into efficient \text{C#} code and used for practical purposes.

3. Symbolic Tree Transducers and \text{FAST}

In this section we define both the syntax of \text{Fast} and its semantics, hand in hand. The concrete syntax of \text{Fast} is shown in Figure 4. \text{Fast} is designed for describing tree transformations and is based on \textit{symbolic tree transducers with regular look-ahead or STTRs}. Symbolic tree transducers are, in essence, top-down tree transducers modulo theories. Regular look-ahead allows for checking additional conditions on subtrees before applying a transformation.

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. We assume closure under Boolean operations and equality. All other operations that are defined in the label theory may be used as well but do not affect the results, i.e., the label theory is used as a "black box". 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 [19].\footnote{Here we write \( \tau \) instead of \( \mathcal{U}^\tau \) for the subuniverse of elements of type \( \tau \).}

Example 1. An example of a decidable label theory is the Boolean combination of quantifier free formulas over linear arithmetic, uninterpreted function symbols, bit-vectors, finite enumerations, tuples, sets, and arrays. Such a label theory is supported in state-of-the-art satisfiability modulo theories (SMT) solvers such as \( \text{Z3} \) [23].

Indicators: \( ID = \{ \text{a.a}[\text{z}].2_.]/\text{a.z}[\text{z}].2_.|.|0.9\}^\ast \)

Constants: \( Const = \text{true} | \text{false} | \ldots \text{strings}, \text{numbers} \)

Basic types: \( \sigma = \text{String} | \text{Int} | \text{Real} | \text{Bool} | \ldots \)

Built-in operators: \( op = \{ < | > | * | + | \ldots \} \)

Constructors: \( c : ID \)

Natural numbers: \( k : \mathbb{N} \)

Tree types: \( \tau : ID \)

Language states: \( p : ID \)

Transformation states: \( q : ID \)

Attribute fields: \( x : ID \)

Subtree variables: \( y : ID \)

Main definitions:

\[ L ::= p | (\text{intersect } L \ L) | (\text{union } L \ L) | (\text{compose } L \ T) | (\text{restrict } L \ T) | (\text{restrict-out } L \ T) \]

\[ T ::= q | (\text{compose } T \ T) | (\text{restrict } T \ T) | (\text{restrict-out } T \ T) \]

Transformation states: \( q : ID \)

Language states: \( p : ID \)

Attribute fields: \( x : ID \)

Subtree variables: \( y : ID \)

Operations on languages and transductions:

Define \( \top \) to be \{ \( c \in \Sigma \mid \exists \sigma\) such \( c\) \( \in L \) \}. The transformation \text{sani} that is equivalent to \text{rem}\_\text{esc}, but restricted to only accept inputs in the language \text{node}\text{Tree}.

The background is enriched with labeled trees defined as follows. A \textit{ranked alphabet} \( \Sigma \) is a finite set of symbols \( \sigma \) associated with fixed nonnegative \textit{ranks} denoted \( \varrho(c) \). We let \( \Sigma(k) = \{ c \in \Sigma \mid \varrho(c) = k \} \) and \( \varrho(\Sigma) = \max(\varrho(c) \mid c \in \Sigma) \) and require that \( \varrho(\Sigma) \neq 0 \).

Definition 1. Given a ranked alphabet \( \Sigma \) and a \textit{type} \( \tau_\Sigma(\sigma) \) is the \textit{tree type} with \textit{constructors}

\[ c : \sigma \times \tau_\Sigma(\Sigma)^{2} \rightarrow \tau_\Sigma(\sigma) \quad (\text{for } c \in \Sigma), \]

and \textit{testers} \( Is_{\Sigma} : \tau_\Sigma(\Sigma)^{2} \rightarrow \text{Bool} \) that test if a tree has a given constructor \( c \). For each constructor \( c \) there is also a \textit{label accessor} \( \pi_{0}^{c} : \tau_\Sigma(\sigma) \rightarrow \sigma \) and, if \( \varrho(c) > 0 \) there are \textit{subtree accessors}

\[ \pi_{1}^{c} : \tau_\Sigma(\sigma) \rightarrow \tau_\Sigma(\sigma), \quad \text{for } 1 \leq t \leq \varrho(c). \]

Note that the \textit{arity} of each constructor \( c \in \Sigma \) is intentionally \( \varrho(c) + 1 \). The functions in Definition 1 assume standard semantics of inductive data types (modulo the theory for \( \sigma \)).\footnote{\textit{Note that the intended set} \( \tau_\Sigma(\sigma) \) \textit{is not first-order axiomatizable but is defined through the standard least fixpoint construction.}} The symbols satisfy the typed versions of the term algebra axioms [7, p. 35, (2.5)–(2.7)] (where \( L = \Sigma \)), the axioms [7, p. 61–62, (6.12)–(6.20)] (where \( \pi_{1}^{c} \) is denoted by \( c_{i} \).

In \text{Fast}, the \textit{type} keyword is used for declaring a new ranked alphabet and corresponding inductive datatype of trees. If there are multiple attribute fields then the label has the corresponding tuple type.

Example 2. The \text{Fast} program in Figure 2, declares the type \( \text{HtmlE} = \tau_{\Sigma}(\text{String}) \) over the ranked alphabet \( \Sigma = \{ \text{nil}, \text{val}, \text{attr}, \text{node} \} \) where, e.g., \( \varrho(\text{attr}) = 2 \).

We write \( c[0](t_{0}, \ldots, t_{k}) \) for \( c[0](t_{0}, t_{1}, \ldots, t_{k}) \) to distinguish the label (or attribute) term from the subtree terms. We further abbreviate \( c[k]() \) by \( c[k] \).
Example 3. A standard definition of binary trees whose labels are pairs of integers can be declared in FAS as follows: \( \text{type } bt : [\text{Int}, \text{Int}] \rightarrow [\text{Node}(0), \text{Node}(2)] \). Then \( \text{bt} = \text{Tree}_{\text{E}}(\text{Int} \times \text{Int}) \), where \( \Sigma(2) = \{ \text{Node} \} \), \( \Sigma(0) = \{ \text{Leaf} \} \). For example \( \text{Node}(3)[5,4][\text{Leaf}(4,5), \text{Leaf}(5,6)] \in \text{bt} \).

Definition 2. A Symbolic Tree Automaton (STA) \( A \) is a tuple \( (Q, \tau, \delta) \), where
- \( Q \) is a finite set of states,
- \( \tau \) is a tree type \( \text{Tree}_{\Sigma}(\sigma) \), and
- \( \delta \) is a finite set of rules \( r \) of the form \( (q, c, \varphi, \varphi') \) where \( q \in Q, c \in \Sigma, \varphi, \varphi' \in (2^Q)^{\text{e}((c)}} \) and \( \varphi \) is a \( \sigma \)-predicate. We write \( r.q, r.c, r.\varphi \) and \( r.\rho \) for the components of \( r \).

For \( q \in Q, \delta(q) \triangleq \{ r \in \delta \mid r.q = q \} \).

Definition 3. The language of \( A \) for \( q \in Q \), is the subset \( L^q_A \subseteq [L^q_A] \) of \( \tau \), where \( L^q_A \) is a fixed unary relation symbol that satisfies the following axiom, modulo the label theory, for all \( y : \tau \):

\[
L^q_A(y) \leftrightarrow \bigvee_{r \in \delta(q)} r^+ \tau^+ \neg \neg \tau^+ r^+ \neg \neg \neg \tau^+ y
\]

(We omit the subscript \( A \) in \( L^q_A \) when \( A \) is clear from the context.) The above mutually recursive definitions of \( L^q \), for \( q \in Q \), are well-defined because of the following well-founded order that decreases from left to right in the definitions: in \( r^+ \tau^+ \) all occurrences of \( L^q \) are applied to proper subterms of \( y \) and have therefore strictly smaller depth because \( \tau \) is an inductive datatype. Moreover, it follows that the axioms are satisfiable and \( L^q \) for \( q \in Q \) is unique.

Definition 4. \( A \) is in normal form or normalized if for all rules \( r \in \delta, \) and for all \( i, 1 \leq i \leq 2(r.c), \) \(| r.\rho[i] | = 1 \).

A is deterministic if it is normalized and for all rules \( (q, c, \varphi, r),(p, c, \psi, \rho) \) of \( A \), if \( \exists \varphi \land \exists \psi \neq \emptyset \) then \( q = p \).

A normalized STA without attributes\(^3\) and with a dedicated initial state corresponds to a classical tree automaton. STAs can be normalized and determined. Normalization is discussed in Section 4. Determinization is outside the scope of this paper but is based on a symbolic generalization of the classical powerset construction and provides the basis for complementing STAs. In FAST the statement

\[
\text{lang } q : \tau \{ \text{c}(y) \text{ where } \varphi(x) \text{ given } \ell(y) \} \text{ one rule } r
\]
defines all the rules in \( \delta(q) \), and thus corresponds precisely to the axiom of \( L^q \). In particular, \( r.\rho[i], \) for \( 1 \leq i \leq 2(r.c) \), is the set of all \( p \) such that \( (p, y) \) occurs in \( \ell(y) \).\(^4\)

Example 4. Consider the definition of \( \text{valTree} \) in Figure 2: \( \delta(\text{valTree}) = \{ (\text{valTree}, \text{val}, \lambda x.x \neq \text{null}, \text{valTree}), (\text{valTree}, \text{nil}, \lambda x.x = \text{null}, ()) \} \).

\[
(q, \text{Node},\lambda x.x < 4, () \emptyset), (q, 2) \rightarrow (q, 2' )
\]

Figure 5: Example of a rule \( (q, \text{Node}, \varphi, \varphi, \Lambda, \theta) \) for symbol \( \text{Node} \) of rank 2. In FAST it may look like a \( \text{Tree}_{\text{E}}(\text{Int} \times \text{Int}) \).

The guard of the rule requires the attribute to be less than 4, there is no look-ahead, and the output \( \Lambda \) and continuation \( \theta \) are depicted as a tree, where each occurrence of a leaf \((p, i) \) corresponds to a distinct output variable whose value is some continued transformation of the \( i \)th input subtree from state \( p. \) The \( \lambda \)-prefix of the output is omitted.

Definition 5. A Symbolic Tree Transducer with Regular look-ahead (STTR) \( S \) is a tuple \( (q, q^0, \tau, \tau', \Delta, R, \delta) \), where \((R, \tau, \delta) \) is an STA called the look-ahead automaton of \( S \),
- \( Q \) is a finite set of states,
- \( q^0 \in Q \) is the initial state,
- \( \tau \) is the input tree type \( \text{Tree}_{\Sigma}(\sigma) \),
- \( \tau' \) is the output tree type,
- \( \Delta \) is a finite set of rules of the form \( r = (q, c, \varphi, \varphi, \Lambda, \theta) \), where \( q \in Q, c \in \Sigma, \varphi, \varphi' \in (2^Q)^{\text{e}((c)}} \) and \( \Lambda \) is a \( \lambda \)-term \( \lambda l.x_1, \ldots, x_n. t \) called the output of \( r \),
- \( \theta \) is a term of type \( \tau \), \( x \) has type \( \sigma, n \geq 0 \), \( x_1, \ldots, x_n \) is a sequence of distinct output variables of type \( \tau' \), each \( x_i \) must occur exactly once in \( t \).

Let \( \Delta(q) \triangleq \{ r \mid r.\varphi = q \} \) and \( \Delta^+ \triangleq \{ r \mid r.\varphi = c \} \).

S is called a Symbolic Tree Transducer (STT) when \( R = \emptyset \).

Definition 6. The transduction relation of \( S \) for \( q \in \emptyset \) is the relation \( T^q_3 \triangleq [T^q_3] \) over \( \tau \times \tau' \), where \( T^q_3 \) is a fixed binary relation symbol that satisfies the following axiom, modulo the label theory, for all \( y : \tau \) and \( z : \tau' \):

\[
T^q_3(y, z) \equiv \bigvee_{(q, c, \varphi, \varphi, \Lambda, \theta) \in \Delta(q)} (q, c, \varphi, \varphi) \Lambda \text{ given } \ell(y) \text{ to } f(x, y) \ldots
\]

one rule \( r \)

where, in \( f(x, y_1, \ldots, y_k) \), for \( 1 \leq i \leq (c) \) and \( p \in Q \), each occurrence of \( (p, y_i) \) corresponds to a fresh output variable associated with \( (p, i) \) in \( \emptyset \), as illustrated in Figure 5. For all \( (p, y) \) in \( \ell(y_1, \ldots, y_k) \), \( p \) belongs to the look-ahead \( r.\rho[i] \).

Example 5. Recall the transduction statement \( q = \text{remScript} \) in Figure 2. The corresponding rules are: the “safe” case:

\[
(q, \text{node}, \lambda x.x \neq \text{"script"}, () \emptyset, \emptyset), \lambda(x, z_1, z_2, z_3).\text{node}[z](z_1, z_2, z_3), (\text{id}, (i, q), (q, 2), (q, 3))
\]

where \( id \) is the identity transformation; the “unsafe” case:

\[
(q, \text{node}, \lambda x.x = \text{"script"}, (), () \emptyset, \lambda(x, z, z_1), (q, 1))
\]

and the harmless case: \( (q, \text{nil}, [], \lambda x.\text{nil}[x], ()) \).

Definition 7. The transduction (function) of \( S \), is the function \( \pi_\tau : \tau \rightarrow 2^{\tau'} \), \( \pi_\tau(t) \triangleq \{ u \mid T^{q_0}_3 (t, u) \} \).
3.1 The role of regular look-ahead

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 delete subtrees, the domain is not preserved by the composition.

Example 6. Assume $\Sigma$ has rank 2, and $\Sigma(2) = \{b\}$ and $\Sigma(0) = \{e\}$. $\mathcal{F}_2(t) = \{t\}$ iff $t$ does not contain a false label; $\mathcal{F}_0 = \emptyset$, otherwise. $\mathcal{F}$ is the transduction that for every $t \in \text{Trees}_2(\text{Bool})$ outputs $\{e\}$. It is easy to see that both transductions are definable using STTs. Now consider the composed transduction $\mathcal{F} = \mathcal{F}_0 \circ \mathcal{F}_2$. $\mathcal{F}$ outputs $\{e\}$ iff the input tree does not contain a false label. However this function cannot be defined by any STT, when reading a node $a$, 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 that the other subtree does not contain any false labels.

A position $p$ in a tree $t$ ($p(t)$) is a sequence of ancestors that is well-defined, e.g. the position $p = [p_1, p_2]$ of $t = b[1][b[4](z, w), y]$ refers to $w$. We call $\Pi(t)$ the set of positions in $t$. Given a state $q \in Q$, and a symbol $c \in \Sigma$, two rules $r, r' \in \Delta(q)$ are equivalent with respect to a predicate $\varphi$ if $\forall A = \lambda(x, z_1, \ldots, z_n), t, r', \Lambda = \lambda(x, z_1, \ldots, z_n), t'$ and $a \in [\varphi]$, $\Pi(t(a)) = \Pi(t'(a))$ and for all $p \in \Pi(t(a))$, $1) p(t(a))$ is a variable $z_i$ if $p(t'(a))$ is a variable $z_j$ and $r.\theta(i) = r'.\theta(j)$, for some $i$ and $j$. 2) if $p(t(a))$ is not a variable then $p(t(a)) = p(t'(a))$. The following definition of deterministic STTRs is important and captures a practically useful fragment of STTRs. The intuition behind the definition is that two different transformation rules must not be enabled for the same input tree.

Definition 8. An STTR $S$ is deterministic when the following holds for all $q \in Q$, $c \in \Sigma$, and $r, r' \in \Delta(q)$, let $V = \{r.\varphi \cap r'.\varphi\}$ and let

$$L_i = (\bigcap_{p \in r.\varphi[i]} L^p) \cap (\bigcap_{p \in r'.\varphi[i]} L^p) \quad (1 \leq i \leq \sharp(c)).$$

If $V \neq \emptyset$, and for all i, $1 \leq i \leq \sharp(c)$, $L_i \neq \emptyset$, then $r.\theta = r'.\theta$ where $\psi = r.\varphi \land r'.\varphi$.

Note that the look-ahead automaton does not have to be deterministic. Whether it is or is not deterministic is orthogonal to $S$ being deterministic.

Example 7. The following ML code is intended to implement a function $\text{negIfLeftOdd}$ that negates a node value if its value in its left child is odd, and otherwise it leaves the value unchanged.

datatype tree = Leaf of nat | Node of (nat * tree * tree)
fun oddRoot (t:tree) : bool =
case n of
| Leaf(x) => (odd x)
| Node(x,t1,t2) => (odd x)

fun negIfLeftOdd (t:tree) : tree =
case n of
| Leaf(x) => n1
| Node(x,t1,t2) => case (oddRoot t1) of
| true => Node(¬x,negIfLeftOdd t1,negIfLeftOdd t2)
| false => n1

The function $\text{oddRoot}$ takes a tree as argument and returns true iff the value of the root is odd. For example $\text{negIfLeftOdd}(\text{Node}(1,\text{Leaf}(2),\text{Leaf}(3))) = \text{Node}(1,\text{Leaf}(2),\text{Leaf}(3))$

It is possible to describe $\text{negIfLeftOdd}$ by a non-deterministic STT. (nondeterminism is used to guess if the label of the left child is odd or even.) We show how the transformation can be described by a deterministic STTR. We start with an equivalent FAST program.

$$\begin{align*}
\text{type} & \quad \text{tree} [\text{x} : \text{Int}]; \text{Leaf}(0), \text{Node}(2) \\
\text{lang} & \quad \text{oddRoot}: \text{tree} \\
& \quad \text{Node}(1, 12) \text{ where } (\text{odd } x) \\
& \quad \text{Leaf}(0) \text{ where } (\text{odd } x) \\
\text{lang} & \quad \text{evenRoot}: \text{tree} \\
& \quad \text{Node}(1, 12) \text{ where } (\text{even } x) \\
& \quad \text{Leaf}(0) \text{ where } (\text{even } x) \\
\text{trans} & \quad \text{negIfLeftOdd}: \text{tree} \rightarrow \text{tree} \\
& \quad \text{Node}(1, 12) \text{ given } (\text{oddRoot t1}) \\
& \quad \text{to Node}[−x](\text{negIfLeftOdd t1, negIfLeftOdd t2}) \\
& \quad \text{Node}(1, 2) \text{ given } (\text{evenRoot t1}) \\
& \quad \text{to Node}[x](\text{negIfLeftOdd t1, negIfLeftOdd t2}) \\
& \quad \text{Leaf}() \text{ to Leaf}() \\
\end{align*}$$

A tree is in the language $\text{evenRoot}$ or $\text{oddRoot}$ if its root is even or odd respectively. We now show the corresponding deterministic STTR $S$ for the transformation $\text{negIfLeftOdd}$. The initial state is $q$, $\tau$, are the transformation rules and $\tau'$ are the look-ahead rules.

$$\begin{align*}
\delta & \cup \Delta \\
\tau_1 & : (q, \text{Node}, T, ((q_1), (q_2)), \lambda(x, z). \text{Node}[-x](\bar{z}),(\bar{q_1}, (q_2))) \\
\tau_2 & : (q, \text{Node}, T, ((q_1), (q_2)), \lambda(x, z). \text{Node}[-x](\bar{z}),(\bar{q_1}, (q_2))) \\
\tau_3 & : (q, \text{Leaf}, T, (\bar{q}, \lambda(x). \text{Leaf}[x],\bar{q})) \\
\tau_4 & : (q, \text{Node}, \text{odd}, (\bar{q}, 0)) \\
\tau_5 & : (q, \text{Node}, \text{even}, (\bar{q}, 0)) \\
\tau_6 & : (q, \text{Leaf}, \text{odd}, (\bar{q})) \\
\tau_7 & : (q, \text{Leaf}, \text{even}, (\bar{q})) \\
\end{align*}$$

We spell out the meaning of $r_1$ to clarify the semantics. The rule is enabled if the transducer is in state $q$, the input tree has constructor $\text{Node}$ whose left subtree belongs to $L^\text{even}$. The label is unrestricted. When the rule applies it produces a tree whose label is the negated input label and whose left and right subtrees are produced by transforming the corresponding input subtrees from state $q$.

3.2 Operations

The semantics of expressions of the form $\text{def} p : \tau : L$ and $\text{def} q : \tau : T$ are defined through corresponding operations on STAs and STTRs. We assume a given set of main definitions as explained above such that each basic language state $p$ denotes the tree language $[p] = L_p^n$ of the STA $A$ spanned by the rules reachable from the state $p$. Similarly, each basic transformation state $q$ denotes the transduction $[q] = \mathcal{F}_q^n$ of the STTR $S$ spanned by the rules reachable from the state $q$.

$\text{intersect}$, $\text{union}$, $\text{complement}$ correspond to Boolean operations over STAs. In those definitions an STA is assumed to have a fixed initial state.

$\text{domain}$ converts an STTR into an STA by omitting output transformations.

$\text{pre-image } q \text{p}$ computes an STA whose pre-image is $\{t \mid [q](t) \sqcap [p] \neq \emptyset\}$, i.e., the set of all trees $t$ such that some transformation of $t$ from $[q]$ ends up in $[p]$.

$\text{compose } q_1, q_2$ constructs an STTR $T$ such that if $q_1$ is deterministic or $q_2$ is linear then $[T] = [q_1] \circ [q_2]$ (see Section 4.2).

$\text{restrict } q \text{p}$ restricts the domain of $[q]$ to $[p]$, i.e., $[\text{restrict } q \text{p}](t) = [q](t)$, if $t \in [p]$; $\emptyset$, otherwise.
restrict_out \ q \ p \ restricts \ the \ output \ of \ \lfloor q \rfloor \ to \ \lfloor p \rfloor, \ i.e., \ \lfloor \text{restrict}_\text{out} \ q \ p \rfloor(t) = \{ t' \mid \exists t \in [\lfloor q \rfloor(t) \land t' \in [\lfloor p \rfloor]\}.

The above set of operations is not minimal. Several operations are special applications of composition. For example:

\text{restrict}_\text{out} \ q \ p = \ \text{compose} \ q \ (\text{restrict}_\text{I} \ p)

\text{pre-image} \ q \ p = \ \text{domain} \ (\text{restrict}_\text{out} \ q \ p)

where \text{I} is the identity STTR.

4. Composition of STTRs

The core algorithm that we are investigating here is composition of STTRs. Several other algorithms can be implemented through composition. We noted earlier that STTs, unlike SFTs [19], are not closed under composition. The two main reasons for this are to handle \text{delete} and to duplicate input subtrees. The first reason was illustrated earlier in Example 6 and, as shown here, is remedied by introducing regular look-ahead. This second reason, as illustrated in the next example, shows that duplication causes problems when combined with nondeterminism.

Example 8. Assume \Sigma has rank 2, \Sigma(2) = \{a, b\} and \Sigma(0) = \{e\}. \text{f}_1 is the transformation that given a tree \( t \in \text{Tree}_< \), nondeterministically swaps \( a \) and \( b \). For example on the input \( a(a(e, e)) \) \( f_1 \) produces \( a(e, e) \) itself and \( b(e, e) \). \text{f}_2 is the function that for every \( t \in \text{Tree}_< \) outputs the tree \( a(t, t) \). It is easy to see that \( f_1 \) and \( f_2 \) are both definable using STTs. Now consider the composed transformation \( f = f_1 \circ f_2 \). On input \( t \), the output of \( f \) is \( \{a(a(t', t')), t' \in f_1(t)\} \). However this function cannot be defined by any STT. In fact when processing the two copies of \( t \) that will be attached to the root, the STT cannot "coordinate" them and make sure they will produce the same output, due to nondeterminism.

We show that STTRs are closed under composition when one of the following conditions is fulfilled: STTR is linear (duplication of subtrees is not allowed), or the STTR is deterministic (recall Definition 8).

4.1 Composition algorithm

We first recall some basic notions over functions. Given \( f : X \to 2^Y \), and \( x \subseteq X \), \( f(x) \stackrel{\text{def}}{=} \bigcup_{y \in x} f(y) \). Given \( f : X \to 2^Y \) and \( g : Y \to 2^Z \), \( \circ \) is defined as \( (g \circ f)(x) = g(f(x)) \). This definition follows the convention in [5], i.e., \( a \) applies first \( f \), then \( g \), contrary to how \( o \) is used for standard function composition.

We first present an algorithm that given two STTRs \( S \) and \( T \) such that \( r_S \to T = r_T \) and \( r_T \to S = r_T \). We then prove that, under the assumptions mentioned above, \( S \circ T = S \circ T \).

We consider binary trees only. Generalization to arbitrary ranked alphabets is straightforward.

At the top level the algorithm is a fixpoint construction that can be described using depth first search. For uniformity of presentation and without loss of generality, look-ahead rules are represented by transduction rules as follows. We write \( \text{AND}(t_1, t_2) \) for the term \( \text{Node}[a](t_1, t_2) \) to symbolically represent a conjunction using a tree term, and we write \( \text{TRUE} \) for \( \text{Leaf}[a] \), where \( a \) is some fixed attribute value. We assume that \( R_S \subseteq \chi_S \times \chi_T \) and that each look-ahead rule such as \( p, \text{Node}, \varphi, (([p_1]), ([p_2])) \) is represented by a (domain-equivalent) transduction rule of the form

\( (p, \text{Node}, \varphi, \lambda(x, y_1, y_2). \text{AND}(\tilde{p}_1(y_1), \tilde{p}_2(y_2))) \)

\( \text{restrict}_\text{out} \ q \ p \)

\( \text{compose} \ q \ (\text{restrict}_\text{I} \ p) \)

\( \text{domain} \ (\text{restrict}_\text{out} \ q \ p) \)

\( \Theta \}

\( \text{restrict}_\text{out} \ q \ p \)

\( \text{compose} \ q \ (\text{restrict}_\text{I} \ p) \)

\( \text{domain} \ (\text{restrict}_\text{out} \ q \ p) \)

\( \Theta \}

\( S \circ T \stackrel{\text{def}}{=} \)

\( S := \text{Normalize}(S); \ T := \text{Normalize}(T); \)

\( q^0 := \{q_0^0, q_0^1\}; \ Q := \{q_0\} \cup R_S; \ \Delta := \delta_S; \ R := R_S; \)

\( \text{Stack} := [q^0]; \)

\( \text{while} \ \text{Stack} \neq [] \)

\( \text{pop} \ (p, q) \text{ from Stack; } \)

\( \text{foreach} \ r \in \text{Transduce}(p, q) \text{ add } r \text{ to } \Delta; \)

\( \text{return} \ \text{Normalize}(Q, q^1, \tau_S, \tau_T, \Delta, R, \{r \in \Delta \mid r.q \in R\}); \)

\( \text{IsSat}(\Sigma, \varphi, q) \defeq \)

\( \text{foreach} \ q, (p, \varphi, \rho, f) \in \Delta; \)

\( \text{IsSat}(\Sigma, \varphi) \text{ then yield } (f(q), \varphi \land \varphi(q), \ell); \)

\( \text{Transduce}(q, \text{Node}(a)(t_1, t_2), \ell, \xi, \ell) \defeq \)

\( \text{foreach} \ (q, \text{Node}, \varphi, \rho, f) \in \Delta; \)

\( \text{IsSat}(\Sigma, \varphi) \text{ then } \text{yield } (\rho(t_1, t_2), \xi \land \varphi(t_1), \ell); \)

\( \text{Reduce}(q, \text{Node}(a)(t_1, t_2), \ell, \xi) \defeq \)

\( \text{foreach} \ v \in \text{Reduce}(f(t_1, t_2), \xi', \ell') \text{ yield } v; \)

\( \text{Transduce}(q, \text{Leaf}(a), \ell, \xi, \ell) \defeq \)

\( \text{yield } (\ell, \xi, \ell); \)

\( \text{Reduce}(\text{Leaf}(a), \ell, \xi, \ell) \defeq \)

\( \text{yield } (\ell, \xi, \ell); \)

\( \text{Reduce}(\text{Node}(a)(t_1, t_2), \ell, \xi, \ell) \defeq \)

\( \text{foreach} \ (u_1, \xi_1, \ell_1) \in \text{Reduce}(t_1, \xi, \ell); \)

\( \text{foreach} \ (u_2, \xi_2, \ell_2) \in \text{Reduce}(t_2, \xi_1, \ell_1); \)

\( \text{yield } (\text{Node}(a)(u_1, u_2), \xi_2, \ell_2); \)

\( \text{Reduce}(\text{Leaf}(a), \ell, \xi, \ell) \defeq \text{Transduce}(q, t, \xi, \ell); \)

\( \text{where the look-ahead component is irrelevant. In transduction rules, the output } \Lambda \text{ and continuation } \theta \text{ are combined into a single term } \chi(x, \theta[1], \ldots, \theta[\theta]) \text{ where each pair } (p, i) \in \theta \text{ is considered as a conjunction point term } \tilde{p}(y). \)

In the constructed STTR \( S \circ T \) we have \( Q_{ST} \subseteq \chi_S \times Q_T \cup R_S \) and the look-ahead rules form a subset of the transduction rules. In a final phase (as described below) the merged state conditions are normalized to single state conditions.

Symbolic transduction

The main step of the algorithm, \text{Transduce}, symbolically compiles a single reduction step from \( S \) followed by multiple reduction steps from \( T \) starting in a given pair state \( (p, q) \in Q_S \times Q_T \). See Figure 6.

5 The actual implementation in C# is more verbose but quite similar to the one presented here.
In the algorithm we omit `λs` for readability, the attribute variable is assumed to be `x` and the input subtree variables for the left and right subtrees of `Node` are assumed to be `y_1` and `y_2`. The `foreach` statement corresponds to returning a finite enumeration of all possible choices. If there are no possible choices then the result is empty. We use the keyword `yield` to indicate the enumeration semantics.⁶

**Normalization**

Normalization updates the regular look-ahead component so that for all `r = (q, Node, φ, (q_1, q_2), f)`, `q` is a singleton set and, if `t_1 ∈ L^q_1` and `t_2 ∈ L^q_2`, then `f(a, t_1, t_2)` is a valid input, i.e., the look-ahead is at least as strict as the transformation function. The core idea behind the normalization is as follows.

Let `A = (R, Tree, δ)` be an STA. We are going to compute merged rules `(q, c, φ, ρ)` over merged states `q ∈ 2^R` where `ρ ∈ 2^K × 2^K` and `φ` is a set of predicates considered as a conjunction. We define the new transition relation `δ` as follows:

\[
δ(∅) \overset{def}{=} \{(0, c, φ, ρ(c))\}
\]

\[
δ^c((p)) \overset{def}{=} \{((p), c, φ, ρ) \mid (p, c, φ, ρ) ∈ δ\}
\]

\[
δ^p(p ⊔ q) \overset{def}{=} \{r \mid s \in δ^p(p), s ∈ δ^q(q)\}
\]

\[
(p, c, φ, (p_i), i=1,...,n) \wedge (q, c, ψ, (q_i), i=1,...,m) \overset{def}{=} (p ⊔ q, c, φ ∨ ψ, (p_i ⊔ q_i), i=1,...,n)
\]

Algorithmically, merged rules are computed lazily and those with unsatisfiable conditions `∧ ∈ δφ` are eliminated eagerly. New concrete states are created for reachable merged states.

**Cleanup**

After normalization, the composition is cleaned by eliminating useless states. Normalization may create states `q` in `Q ⊔ R` that are either unreachable or deadends (no tree is accepted from `q`). For example, if the only rule from a state `q ∈ R` is `(q, Node, T, (q, g))` then `q` is a deadend because `g` cannot be eliminated by applying the rule. Elimination of deadends uses the algorithm for eliminating useless symbols from a context-free grammar [9, p. 88–89]. It converts rules to context-free grammar productions where states are non-terminals and the initial state is the start symbol.

### 4.2 Properties of composition

We now show the main correctness result of the composition algorithm. An STTR is linear if for all rules `r`, and for all `i`, `1 ≤ i ≤ ||r||c`, the continuation `r.θ` contains at most one pair whose second element is `i`.

The key technical result that we need is the following quantifier-elimination lemma for proving the composition theorem of STTRs.

**Lemma 1.** If `T` is linear or if `S` is deterministic then, for all `(p, q) ∈ Q_{S ⊔ T}`, and all `t ∈ T_S`, `v ∈ T_T^q`:

\[
∃u(T^q_S((t, u) \wedge T^q_T(u, v)) \Rightarrow T^q_{S ⊔ T}(t, v))
\]

The proof of Lemma 1 is by induction over `t`. It uses laws of distributivity, in particular that `∃` distributes over disjunctions, and laws of equality. It also makes use of the dual characterization of `T^q` that uses the accessors for expressing the constraint that `z` is equal to the output of a rule to avoid the inner existential quantifiers in the definition of `T^q`, similar to the case of term algebras [7].

**Theorem 1.** If `T` is linear or if `S` is deterministic then `T_{S ⊔ T} = T_S ⊔ T_T`.

Proof. The statement `v ∈ (T_S ⊔ T_T)(t)` is (by definition) equivalent to `∃u(T^q_S((t, u) \wedge T^q_T(u, v)))`, which by Lemma 1, is equivalent to `T^q_{S ⊔ T}(t, v)`, i.e., `v ∈ T_{S ⊔ T}(t)` (since `(q, q_0) = T^q_{S ⊔ T}`).

Termination of the composition algorithm is guaranteed by the fact that the size of the search stack and thus the number of top-level iterations is bounded by `|Q_S × Q_T|`. The final normalization step is needed to eliminate look-ahead rules with merged states. Normalization is in the worst case exponential in the size of `|Q_S × Q_T|` because it uses a subset construction.

We have not investigated the precise complexity of the algorithm, or what can be said about the properties of `S ⊔ T` when `S` is nondeterministic and `T` is nonlinear.

### 5. Evaluation

Fast can be applied in multiple different applications. Section 5.1 considers HTML input sanitization for security. Section 5.2 shows how augmented reality (AR) applications can be checked for potential conflicts. Section 5.3 shows how deforestation in functional languages can be implemented using transducers. We show the lines of code required for each FAST program in Figure 7. Finally, we sketch how classic transformations from machine learning and compiler optimization 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. A common solution is encoding untrusted inputs before they can reach application outputs. For example, the `<` character might be encoded as a three character sequence `&lt;`. While finding exactly where to place calls to encoding routines is a challenging task, often encoding away potentially malicious content is too crude an approach.

Consider bulletin boards that want to allow partial markup such as `<p>` and `<b>` 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 subtler 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 [16] and the Yamanner worm [2] for the Yahoo Mail system. In fact, MySpace has repeatedly failed to properly sanitize their HTML inputs, leading to a Month of MySpace Bugs initiative [15].

Surprisingly, HTML sanitization turns out exceedingly tricky to get right; we want to permit proper formatting, while disallowing script injection. Script in HTML documents can, however, come in many forms: embedded as &lt;script&gt; blocks, attacked to HTML elements as handlers, etc. Performers can, however, come in many forms: embedded as &lt;script&gt; blocks, attacked to HTML elements as handlers, etc.

We focus on such taggers. A tagger, given a set of input elements, tags some of them with a piece of tags representing the monuments in such city. We assume that the physical world is represented as a tree of elements.

Conflict detection: We need to know if two taggers might conflict with each other. 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 (p₁, p₂):

1. **[composition]** we compute p₁ composition of p₁ and p₂;
2. **[input restriction]** we compute p₁', a restriction of p that only accepts trees where each node contains no tags;
3. **[output restriction]** we compute p₁'', a restriction of p' that only outputs trees where at least one node has been tagged twice;
4. **[check]** we check if p₁'' is the empty transducer: if this is the case p₁ and p₂ do not conflict, otherwise they conflict on every input accepted by p₁''.

Performance: Figure 9 shows the timing results for conflict analysis. To collect this data, we randomly generate 100 taggers in FAST and check whether they conflict with each other. Each tagger we generate conforms to the following properties: 1) it is non-empty; 2) it tags on average 3 nodes; 3) it tags each node at most once.

The sizes of our taggers vary from 1 to 95 states. The language we use for the input restriction has 3 states, the one for the output 5 states. We analyze 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. The x axis represents the running time in ms. Notice that the scale is logarithmic and the times are intervals (i.e. [4-8] means the running time is between 4 included and 8 excluded). The y axis represents the number of cases that completed in a given time interval. For example in Figure 9(a) we see that around 1,600 compositions took between 16 and 32 milliseconds.

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 phone AR platform applications [http://www.layar.com/layers/new] 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.

Taggers: We focus on such taggers. A tagger, given a set of input elements, tags some of them with a piece of information based on properties of the elements. As an example, consider a tagger that assigns to every city a set of input elements, tags some of them with a piece of information based on properties of the elements.
Our last case study explores the idea of deforestation. First introduced by Wadler in 1988 [20], deforestation is the idea of 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:

\[
\text{sum} = (\text{map square (upto 1 n))}
\]

Intermediate lists created as a result of evaluation are a source of inefficiency.

However, it has been observed that transducer composition can also be used to eliminate intermediate results. This can be done as long as individual functions are represented as transducers and their composition is a transducer itself. Künnemann et al. [10] consider a class of syntactic transformations (Macro Tree Transducers) that is bigger than top-down tree transducers with RLA. However their approach does not deal with infinite alphabets.

**Experiment:** Figure 10 shows two common functional programs over lists implemented in FAST: \text{filter} and \text{map}. The programs transform lists of reals into lists of reals. We assume \( p : \text{real} \rightarrow \text{bool} \) and \( f : \text{real} \rightarrow \text{real} \) to be some previously defined predicate and function respectively. The program \text{map filter} will compute the same function as applying \text{map} first and then \text{filter} but in a single pass.

We analyze how effective composition is in our setting. We consider the function \text{map} in Figure 10 with \( f \) defined to be a simple version of the Caesar cipher: \( \lambda x. (x + 5) \mod 26 \).

**Performance:** We compose the function \text{map} with itself several times and see how the performance improve when using transducers to avoid the intermediate result computation. Figure 11 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. On the other hand, the running time of the naïvely composed functions degrades linearly in the number of composed functions.

5.4 Additional Applications

**Decision Tree Manipulation.** Decision trees are a fundamental construct in machine learning, used in everything from computer vision to network anomaly detection. A decision tree is a tree where each interior node is associated with a predicate; evaluation begins at the root and proceeds down a “true” or “false” branch until reaching a leaf. The leaf then determines the final category of the input. A classic technique to improve efficiency and avoid overfitting is postpruning, in which each node is evaluated on a training set according to a performance criterion. Nodes that fail the performance criterion are removed and their children are merged into a replacement node [17].

Because the training set is finite, for each training set and each performance criterion, we can construct a transducer that encodes a node’s performance on the training set. We can then use our composition analysis to perform post-pruning in a single pass, even when multiple training sets are considered. Mansour and McAllester also show how multiple decision trees can be merged to form branching programs, in which the structure is an arbitrary directed acyclic graph instead of a tree [12]. The merge criterion they provide again depends on each node’s performance against a training set. While our transducers output trees, not graphs, they can readily mark nodes in each tree for merging.

Our formalism captures classic operations on decision trees, while opening the way to new ones. For example, our pre-image synthesis allows us to take a decision tree and recover possible “pre-pruned” trees for a given data set.

**Compiler Optimization.** Compilers employ local optimizations on IR and generated code. Some of these transformations can be represented as tree manipulations, such as common subexpression elimination. By representing these as transducers and performing composition, we obtain a single pass over IR to perform all optimizations.

6. Related Work

Symbolic finite transducers (SFTs) over lists were originally introduced in [8] with a focus on security analysis of string sanitizers. A domain-specific language for string manipulations, BEK, whose semantics is based on SFTs is also defined in [8]. The key properties that are studied in [8] from a practical point of view are idempotence, commutativity and equivalence checking of string sanitizers. The formal foundations and the theoretical analysis of the underlying SFT algorithms, in particular, an algorithm for deciding equivalence of SFTs modulo a decidable background theory is studied in [19]. The formalism of SFTs is also extended in [19] to
Symbolic Transducers (STs) that allow the use of registers. STTs are originally introduced in [18], where it is wrongly claimed that STTs are closed under composition by referring to a generalization of a proof of the classical case in [5] which is only stated for total deterministic finite tree transducers. In [6] this error is discovered and other properties of STTs are investigated. The main result of the paper [18] is an equivalence checking algorithm for single-valued linear STTs. We are currently investigating decidability of equivalence checking of single-valued STTRs.

TTT [14] and Tiburon [13] are transducers based languages specialized on tree transformations for natural language processing. TTT allows complex forms of pattern matching, but does not enable any form of analysis. Tiburon supports several transducers algorithms and allows the use of probabilistic transitions. Both the languages only support finite input and output alphabets. 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.

XPath [21] and XSLT [22] are languages for manipulating XML trees. XPath can extract, but not transform, nodes from XML trees. XSLT can transform nodes and uses XPath to access them. Both these languages are designed for programming aid and do not support analysis. However, some restricted fragments of XPath [1] have decidable emptiness. While FAST supports several decidable theories, the approach in [1] only allows equality checking of nodes in the input. Identifying a fragment of XPath expressible in FAST is an interesting research question we have not explored.

Table 12 summarizes the relations between FAST and the other domain-specific languages for tree transformations.

The connection between tree transducers and deforestation was first investigated in [20], where programs that manipulate trees over finite alphabets were considered. In [20] integers are represented as unary lists of the form \( S(\ldots (0) \ldots) \). The deforestation is done via Macro Tree Transducers (MTTs) [4]. MTTs are more expressive than Top Down Transducers with RLA, but they do not support infinite alphabets. MTTs can be composed, but the composition algorithm has high complexity. Indeed, we are not aware of an actual implementation of deforestation based on MTTs or any other transducer model.

Higher-Order Multi-Parameter Tree Transducers (HMTT) [11] are used for type-checking of functional programs. HMTTs enable sound but incomplete analysis of programs which takes multiple trees as input. HMTTs only support finite alphabets. Extending our theory to multiple input trees is an open research direction.

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 an unnormalized STA. In the classical case, the non-emptiness problem of tree automata is \( P \)-complete, while the intersection non-emptiness problem is \( \text{EXPTIME} \)-complete [3, Theorem 1.7.5].

7. Conclusions

In this paper, we have introduced FAST, a new domain-specific language for tree manipulation, gave a semantics for this language in terms of symbolic tree transducers, and we showed how multiple applications benefit from this analysis. FAST strikes a delicate balance between precise analysis and expressiveness. Our work shows this balance is sufficient across a range of applications in different fields.

References


---

<table>
<thead>
<tr>
<th>Language</th>
<th>Alphabet</th>
<th>Analyzable</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>infinite</td>
<td>yes</td>
<td>Tree-manipulating programs</td>
</tr>
<tr>
<td>TTT</td>
<td>finite</td>
<td>no</td>
<td>NL</td>
</tr>
<tr>
<td>TIBURON</td>
<td>finite</td>
<td>yes</td>
<td>NL</td>
</tr>
<tr>
<td>XPath</td>
<td>infinite</td>
<td>fragment</td>
<td>XML query (only Selection)</td>
</tr>
<tr>
<td>XSLT</td>
<td>infinite</td>
<td>no</td>
<td>XML transformations</td>
</tr>
</tbody>
</table>

Figure 12: Languages for tree-manipulating programs.