FAST: a Transducer-Based Language for Tree Manipulation

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 and obtain results comparable to leading libraries but with smaller code. Next we show how augmented reality "tagging" applications can be checked for potential overlap in milliseconds using FAST type checking. We show how transducer composition enables *deforestation* for improved performance. Overall, we strike a balance between expressiveness and precise analysis that works for a large class of important tree-manipulating programs.

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 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.

	Composition	Intersection	Pre-image
Augmented reality	✓	✓	✓
HTML sanitization	✓		✓
Deforestation	✓		
Decision trees	✓	✓	
Compiler optimization	✓		

Figure 1: Representative applications of FAST. For each application we show which analyses of FAST are needed. Section 5 discusses this in more detail.

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 of augmented reality applications submitted to an app store, 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

into a single FAST program that traverses the input tree only once without producing intermediate results; and

- 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_i takes as input a DOM tree and transforms it into an updated DOM tree. As an example, the FAST program *sani* in Figure 2 traverses the input DOM and outputs a copy of it in which all subtrees whose root is labeled with `"script"` have been removed, and all the characters `"'"` 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. *tag* contains the HTML node type of n (`div`, `br`,...).

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. *tag* contains the name of a (`id`, `style`,...).

Each string value w is encoded as an *HtmlE* element $val(x_1)$ with one child x_1. If the value of w is $s_1 \ldots s_n$ for some $n > 0$, then x_1 encodes the suffix $s_2 \ldots s_n$ and *tag* contains the string "s_1". x_1 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-

```
// Node definition
type HtmlE[tag : String]{nil(0), val(1), attr(2), node(3)}
// Language of HTML trees
lang nodeTree: HtmlE {
      node(x1, x2, x3) given
                (attrTree x1) (nodeTree x2) (nodeTree x3)
    | nil() where (= tag "")   }
lang attrTree: HtmlE {
      attr(x1, x2) given (valTree x1) (attrTree x2)
    | nil() where (= tag "")   }
lang valTree: HtmlE {
      val(x1) where (≠ tag "") given (valTree x1)
    | nil() where (= tag "")   }
// Sanitization functions
trans remScript: HtmlE->HtmlE {
      node(x1, x2, x3) where (≠ tag "script")
            to (node [tag] x1 (remScript x2) (remScript x3))
    | node(x1, x2, x3) where (= tag "script") to (remScript x3)
    | nil() to (nil [tag])   }
trans esc: HtmlE->HtmlE {
      node(x1, x2, x3) to (node [tag] (esc x1) (esc x2) (esc x3))
    | attr(x1, x2) to (attr [tag] (esc x1) (esc x2))
    | val(x1) where (or (= tag "'")(= tag """))
            to (val ["\"](val [tag] (esc x1)))
    | val(x1) where (and (≠ tag "'")(≠ tag """))
            to (val [tag] (esc x1))
    | nil() to (nil [tag])   }
// Composition and restriction to well formed input trees
def rem_esc: HtmlE->HtmlE:=(compose remScript esc)
def sani: HtmlE->HtmlE:=(restrict rem_esc nodeTree)
```

Figure 2: Implementation of a simple HTML sanitizer in FAST.

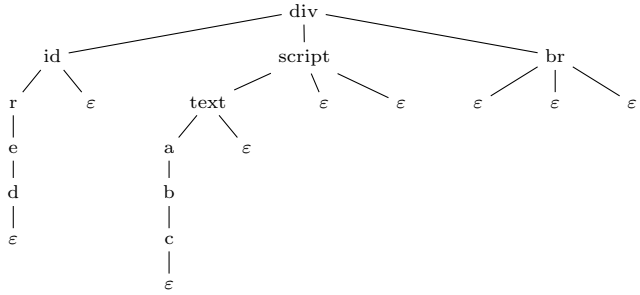

Figure 3: *HtmlE* encoding of the HTML element `<div id='red'><script>abc</script></div><br />`. `div`, `script`, and `br` are built using the constructor *node*; `id`, and `text`, are built using the constructor *attr*. All single character nodes are built using the constructor *val*, while all the ε are built using the constructor *nil*. The string appearing in the figure are the *tags*.

scribes the language of correct HTML encodings: 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 other languages are similar.

Next, we describe the sanitization functions. The transformation *remScript* of type *HtmlE->HtmlE* takes an input tree t of type *HtmlE* and produces an output tree t' of type *HtmlE*. *remScript* recursively processes the input tree t: 1) if t is $node(x_1, x_2, x_3)$ with *tag* `"script"`, the output t' is the result of invoking *remScript* on x_3, 2) if t is $node(x_1, x_2, x_3)$ 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 *remScript* on x_2 and x_3 respectively; 3) if t is *nil* the output t' is a copy of t. The transformation *esc* of type *HtmlE->HtmlE* escapes the characters `'''` and `'"'`. *esc* outputs a copy of the input tree where each node *val* with tag `"'"` or `"""` is prepended a node *val* with tag `"\"`. The transformations *remScript* and *esc* are then composed into a single transformation *rem_esc*. *rem_esc* also accepts input trees that are not in the language *nodeTree* and therefore do not correspond to correct DOM encodings. The last line of Figure 2, defines the transformation *sani* that is equivalent to *rem_esc*, but restricted to only accept inputs in the language *nodeTree*.

This example showed that in 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 FAST implementation efficient. Such function can be finally compiled into efficient C# code and used for practical purposes.

3. Symbolic Tree Transducers and FAST

In this section we define both the syntax of FAST and its semantics, hand in hand. The concrete syntax of FAST is shown in Figure 4. FAST is designed for describing tree transformations and is based on *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 *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 λ-expressions for defining anonymous functions called λ-*terms* without having to name them explicitly. In general, we use standard first-order logic and follow the notational conventions that are consistent with [19].[1]

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 Z3 [23]. ⊠

[1] Here we write τ instead of $\mathcal{U}^\tau$ for the subuniverse of elements of type τ.

```
Indentifiers          ID : (a..z|A..Z|_)(a..z|A..Z|_|.|0..9)*
Constants          Const : true | false | ... strings, numbers
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(,x:σ)* ])? { c(k)(,c(k))+ }
        | lang p : τ { Lrule (| Lrule)* }
        | trans q : τ -> τ { Trule (| Trule)* }
        | def p : τ := L
        | def q : τ -> τ := T )+
Lrule ::= c (( y(,y )*))? (where Aexp)? (given (( p y ))+)?
Trule ::= Lrule to Tout
 Tout ::= y | ( q y ) | ( c ([ Aexp(,Aexp)* ])? Tout* )

Attribute expressions:
Aexp ::= ID | Const | (op Aexp+)

Operations over languages and transductions:
   L ::= p | (intersect L L) | (union L L) |
         (complement L) | (domain T) | (pre-image T L)
   T ::= q | (compose T T) | (restrict T L) | (restrict_out T L)
```

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

The background is enriched with labeled trees defined as follows. A *ranked alphabet* Σ is a finite set of symbols c associated with fixed nonnegative *ranks* denoted $\natural(c)$. We let $\Sigma(k) \stackrel{\text{def}}{=} \{c \in \Sigma \mid \natural(c) = k\}$ and $\natural(\Sigma) \stackrel{\text{def}}{=} \max\{\natural(c) \mid c \in \Sigma\}$ and require that $\Sigma(0) \neq \emptyset$.

Definition 1. Given a ranked alphabet Σ and a *type* σ, $Tree_\Sigma\langle\sigma\rangle$ is the *tree type* with *constructors*

$$c : \sigma \times Tree_\Sigma\langle\sigma\rangle^{\natural(c)} \to Tree_\Sigma\langle\sigma\rangle \quad (\text{for } c \in \Sigma),$$

and *testers* $Is_c : Tree_\Sigma\langle\sigma\rangle \to Bool$ that test if a tree has a given constructor c. For each constructor c there is also a *label accessor* $\pi^c_0 : Tree_\Sigma\langle\sigma\rangle \to \sigma$ and, if $\natural(c) > 0$ there are *subtree accessors* $\pi^c_i : Tree_\Sigma\langle\sigma\rangle \to Tree_\Sigma\langle\sigma\rangle$, for $1 \leq i \leq \natural(c)$.

Note that the *arity* of each constructor $c \in \Sigma$ is intensionally $\natural(c) + 1$. The functions in Definition 1 assume standard semantics of inductive data types (modulo the theory for σ).[2] 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 π^c_i is denoted by c_i).

In FAST, the ***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 FAST program in Figure 2, declares the type $HtmlE = Tree_\Sigma\langle String\rangle$ over the ranked alphabet $\Sigma = \{nil, val, attr, node\}$ where, e.g., $\natural(attr) = 2$. ⊠

We write $c[t_0](t_1, \ldots, t_k)$ for $c(t_0, t_1, \ldots, t_k)$ to distinguish the label (or attribute) term from the subtree terms. We further abbreviate $c[t]()$ by $c[t]$.

[2] Note that the *intended* set $\boldsymbol{Tree_\Sigma\langle\sigma\rangle}$ is *not* first-order axiomatizable but is defined through the standard least fixpoint construction.

Example 3. A standard definition of binary trees whose labels are pairs of integers can be declared in FAST as follows: ***type*** $bt[x:Int, y:Int]\{Leaf(0), Node(2)\}$. Then $bt = Tree_\Sigma\langle Int \times Int\rangle$, where $\Sigma(2) = \{Node\}$, $\Sigma(0) = \{Leaf\}$. For example $Node[3,4](Leaf[4,5], Leaf[5,6]) \in \boldsymbol{bt}$. ⊠

Definition 2. A *Symbolic Tree Automaton (STA)* A is a tuple (Q, τ, δ), where

- Q is a finite set of *states*,
- τ is a *tree type* $Tree_\Sigma\langle\sigma\rangle$, and
- δ is a finite set of *rules* r of the form (q, c, φ, ρ) where $q \in Q$, $c \in \Sigma$, $\rho \in (2^Q)^{\natural(c)}$ and φ is a σ-predicate. We write $r.q$, $r.c$, $r.\varphi$ and $r.\rho$ for the components of r.

For $q \in Q$, $\delta(q) \stackrel{\text{def}}{=} \{r \in \delta \mid r.q = q\}$.

Definition 3. The *language of A for $q \in Q$*, is the subset $\boldsymbol{L}^q_A \stackrel{\text{def}}{=} [\![L^q_A]\!]$ of $\boldsymbol{\tau}$, where L^q_A is a fixed unary relation symbol that satisfies the following axiom, modulo the label theory, for all $y:\tau$,

$$L^q_A(y) \;\Leftrightarrow\; \bigvee_{r\in\delta(q)} \ulcorner r \urcorner(y) \quad \text{where}$$

$$\ulcorner q, c, \varphi, \rho \urcorner \;\stackrel{\text{def}}{=}\; \lambda y.\big(Is_c(y) \wedge \varphi(\pi^c_0(y)) \wedge \bigwedge_{i=1}^{\natural(c)} \bigwedge_{p\in\rho[i]} L^p_A(\pi^c_i(y))\big)$$

(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 $\ulcorner r \urcorner$, all occurrences of L^p are applied to proper subtrees of y and have therefore strictly smaller depth because τ is an inductive datatype. Moreover, it follows that the axioms are satisfiable and $\boldsymbol{L}^q$ 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, $1 \le i \le \natural(r.c)$, $|r.\rho[i]| = 1$. A is *deterministic* if it is normalized and for all rules $(q, c, \varphi, \rho), (p, c, \psi, \rho)$ of A, if $[\![\varphi]\!] \cap [\![\psi]\!] \neq \emptyset$ 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 determinized. 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

$$\boldsymbol{lang}\ q : \tau\ \{\underbrace{c(\bar{y})\ \boldsymbol{where}\ \varphi(\bar{x})\ \boldsymbol{given}\ \ell(\bar{y})}_{\text{one rule } r} \mid \ldots\}$$

defines all the rules in $\delta(q)$, and thus corresponds precisely to the axiom of L^q. In particular, $r.\rho[i]$, for $1 \le i \le \natural(r.c)$, is the set of all p such that $(p\ y_i)$ occurs in $\ell(\bar{y})$.[4]

Example 4. Consider the definition of *valTree* in Figure 2; $\delta(valTree) = \{(valTree,\ val,\ \lambda x.x \neq \texttt{""},\ (valTree)), (valTree,\ nil,\ \lambda x.x = \texttt{""},\ ())\}$. ⊠

[3] Here "without attributes" means that the label type is the *empty tuple type* whose only element is the empty tuple.

[4] FAST uses prefix notation and spaces for argument separators, e.g., a term $f(t_1, t_2)$ is written $(f\ t_1\ t_2)$. Only the top-level attribute uses the notation $[t_1, t_2]$ for constructing a tuple of attribute fields.

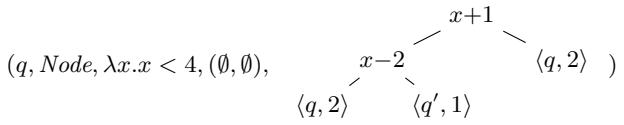

Figure 5: Example of a rule $(q, Node, \varphi, \rho, \Lambda, \theta)$ for symbol *Node* of rank 2. In FAST it may look like a *Trule* $Node(l, r)$ ***where***$(< x\ 4)$ ***to***$(Node[+\ x\ 1](Node[-\ x\ 2](q\ r)(q'\ l))(q\ r))$. The guard of the rule requires the attribute to be less than 4, there is no look-ahead, and the output Λ and continuation θ are depicted as a tree, where each occurrence of a leaf $\langle p, i\rangle$ corresponds to a distinct output variable whose value is some continued transformation of the i'th input subtree from state p. The λ-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, τ, δ) 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*,
- τ is the *input tree type* $Tree_\Sigma\langle\sigma\rangle$,
- τ' is the *output tree type*, and
- Δ is a finite set of *rules* of the form $r = (q, c, \varphi, \rho, \Lambda, \theta)$, where $q \in Q$, $c \in \Sigma$, φ is a σ-predicate, $\rho \in (2^R)^{\natural(c)}$, and
  - Λ is a λ-term $\lambda(x, z_1, \ldots, z_n).t$ called the *output* of r, where t is a term of type τ', x has type σ, $n \ge 0$, $z_1, \ldots, z_n$ is a sequence of distinct *output variables* of type τ', each z_i must occur exactly once in t;
  - $\theta \in (Q \times \{1, \ldots, \natural(c)\})^n$ is the *continuation* of r.

Let $\Delta(q) \stackrel{\text{def}}{=} \{r \in \Delta \mid r.q = q\}$ and $\Delta^c \stackrel{\text{def}}{=} \{r \in \Delta \mid r.c = c\}$. S is called a *Symbolic Tree Transducer (STT)* when $R = \emptyset$.

Definition 6. The *transduction relation of S for $q \in Q$* is the relation $\boldsymbol{T}^q_S \stackrel{\text{def}}{=} [\![T^q_S]\!]$ over $\boldsymbol{\tau} \times \boldsymbol{\tau'}$, where T^q_S is a fixed binary relation symbol that satisfies the following axiom, modulo the label theory, for all $y:\tau$ and $z:\tau'$,

$$T^q_S(y,z) \Leftrightarrow \bigvee_{(q,c,\varphi,\rho,\Lambda,\theta)\in\Delta(q)} \big(\ulcorner q, c, \varphi, \rho \urcorner(y) \;\wedge\; \exists z_1 \ldots z_n\,(z = \Lambda(\pi^c_0(y), z_1, \ldots, z_n) \;\wedge\; \bigwedge_{j=1}^{n} T^{\theta[j][1]}_S(\pi^c_{\theta[j][2]}(y), z_j))\big)$$

Again, well-definedness of Definition 6 holds for reasons similar to the case of Definition 3. In FAST, the axiom of T^q is defined by the statement

$$\boldsymbol{trans}\ q : \tau \texttt{->} \tau'\ \{\underbrace{c(\bar{y})\ \boldsymbol{where}\ \varphi(\bar{x})\ \boldsymbol{given}\ \ell(\bar{y})\ \boldsymbol{to}\ f(\bar{x}, \bar{y})}_{\text{one rule } r} \mid \ldots\}$$

where, in $f(\bar{x}, y_1, \ldots, y_{\natural(c)})$, for $1 \le i \le \natural(c)$ and $p \in Q$, each occurrence of $(p\ y_i)$ corresponds to a fresh output variable associated with $(p, i) \in \theta$, as illustrated in Figure 5. For all $(p\ y_i)$ in $\ell(y_1, \ldots, y_{\natural(c)})$, p belongs to the look-ahead $r.\rho[i]$.

Example 5. Recall the transformation state $q = remScript$ in Figure 2. The corresponding rules are: the "safe" case:

$$(q, node, \lambda x.x \neq \texttt{"script"}, (\emptyset, \emptyset, \emptyset),\ \lambda(x, z_1, z_2, z_3).node[x](z_1, z_2, z_3),\ ((id, 1), (q, 2), (q, 3)))$$

where id is the identity transformation; the "unsafe" case:

$$(q, node, \lambda x.x = \texttt{"script"}, (\emptyset, \emptyset, \emptyset),\ \lambda(x, z).z,\ ((q, 1)))$$

and the harmless case: $(q, nil, \top, (), \lambda\, x.nil[x], ())$ ⊠

Definition 7. The *transduction (function) of S*, is the function $\mathscr{T}_S : \boldsymbol{\tau} \to 2^{\boldsymbol{\tau'}}$, $\mathscr{T}_S(t) \stackrel{\text{def}}{=} \{u \mid \boldsymbol{T}^{q^0}_S(t, u)\}$.

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 Σ has rank 2, and $\Sigma(2) = \{b\}$ and $\Sigma(0) = \{e\}$. $\mathscr{T}_1$ is the transduction that given a tree $t \in \boldsymbol{Tree_\Sigma\langle Bool\rangle}$, $\mathscr{T}_1(t) = \{t\}$ iff t does not contain a false label; $\mathscr{T}_1 = \emptyset$, otherwise. $\mathscr{T}_2$ is the transduction that for every $t \in \boldsymbol{Tree_\Sigma\langle Bool\rangle}$ outputs $\{e[true]\}$. It is easy to see that both transductions are definable using STTs. Now consider the composed transduction $\mathscr{T} = \mathscr{T}_1 \circ \mathscr{T}_2$. $\mathscr{T}$ outputs $\{e[true]\}$ 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 accessors that is well-defined, e.g. the position $p = [\pi_1^b, \pi_2^b]$ 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 $r, r' \in \Delta^c(q)$ are equivalent with respect to a predicate φ, $r \overset{\varphi}{=} r'$, iff the following holds: if $r.\Lambda = \lambda(x, z_1, \ldots, z_n).t$, $r'.\Lambda = \lambda(x, z_1, \ldots, z_n).t'$ and $a \in [\![\varphi]\!]$, then $\Pi(t(a)) = \Pi(t'(a))$ and for all $p \in \Pi(t(a))$, 1) $p(t(a))$ is a variable z_i iff $p(t'(a))$ 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 $r, r' \in \Delta^c(q)$, let $V = [\![r.\varphi]\!] \cap [\![r'.\varphi]\!]$ and let

$$\boldsymbol{L}_i = \Big(\bigcap_{p \in r.\rho[i]} \boldsymbol{L}^p\Big) \cap \Big(\bigcap_{p \in r'.\rho[i]} \boldsymbol{L}^p\Big) \quad (\text{for } 1 \le i \le \natural(c)).$$

If $V \neq \emptyset$ and, for all i, $1 \le i \le \natural(c)$, $\boldsymbol{L}_i \neq \emptyset$, then $r.\theta \overset{\psi}{=} r'.\theta$ where $\psi = r.\varphi \wedge r'.\varphi$.

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 `negIfLeftOdd` that negates a node value if the 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 (t1:tree) : bool =
  case n1 of
     Leaf(x) => (odd x)
   | Node(x,t1,t2) => (odd x)
fun negIfLeftOdd (t1:tree) : tree =
  case n1 of
     Leaf(x) => n1
   | Node(x,t1,t2) => case (oddRoot t1) of
        | true => Node(-x,negIfLeftOdd t1,negIfLeftOdd t2)
        | false => n1
```

The function `oddRoot` takes a tree as argument and returns true iff the value of the root is odd. For example

```
negIfLeftOdd(Node(1,Leaf(2),Leaf(3)) = Node(1,Leaf(2),Leaf(3))
negIfLeftOdd(Node(1,Leaf(5),Leaf(4)) = Node(-1,Leaf(5),Leaf(4))
```

It is possible to describe `negIfLeftOdd` by a nondeterministic 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.

```
type tree[x : Int]{Leaf(0), Node(2)}
lang oddRoot:tree {
      Node(t1, t2) where (odd x)
    | Leaf() where (odd x)   }
lang evenRoot:tree {
      Node(t1, t2) where (even x)
    | Leaf() where (even x)   }
trans negIfLeftOdd:tree->tree {
      Node(t1, t2) given (oddRoot t1)
            to Node[−x](negIfLeftOdd t1, negIfLeftOdd t2)
    | Node(t1, t2) given (evenRoot t1)
            to Node[x](negIfLeftOdd t1, negIfLeftOdd t2)
    | Leaf() to Leaf[x]   }
```

A tree is in the language *evenRoot* or *oddRoot* if its root is even or odd respectively. We now show the corresponding deterministic STTR S for the transformation *negIfLeftOdd*. The initial state is q, r_i are the transduction rules and r'_i are the look-ahead rules.

	$\delta \cup \Delta$
r_1 :	$(q, Node, \top, (\{q_o\}, \emptyset), \lambda(x, \bar{z}).Node[-x](\bar{z}), ((q,1),(q,2)))$
r_2 :	$(q, Node, \top, (\{q_e\}, \emptyset), \lambda(x, \bar{z}).Node[x](\bar{z}), ((q,1),(q,2)))$
r_3 :	$(q, Leaf, \top, (), \lambda x.Leaf[x], ())$
r'_1 :	$(q_o, Node, odd, (\emptyset, \emptyset))$
r'_2 :	$(q_e, Node, even, (\emptyset, \emptyset))$
r'_3 :	$(q_o, Leaf, odd, ())$
r'_4 :	$(q_e, Leaf, even, ())$

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 *Node* whose left subtree belongs to $\boldsymbol{L}^{q_o}$. 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 ***def*** p : τ `:=` L and ***def*** q : τ `->` τ' `:=` T are defined through corresponding operations over 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 $[\![p]\!] = \boldsymbol{L}^p_A$ of the STA A spanned by the rules reachable from the state p. Similarly, each basic transformation state q denotes the transduction $[\![q]\!] = \mathscr{T}^q_S$ of the STTR S spanned by the rules reachable from the state q.

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

domain converts an STTR into an STA by omitting output transformations.

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

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).

restrict q p restricts the domain of $[\![q]\!]$ to $[\![p]\!]$, i.e., $[\![\textbf{\textit{restrict}}\ q\ p]\!](t)$ is $[\![q]\!](t)$, if $t \in [\![p]\!]$; $\emptyset$, otherwise.

restrict_out q p restricts the output of $[\![q]\!]$ to $[\![p]\!]$, i.e., $[\![$***restrict_out*** q $p]\!](t)$ is $\{t' \mid t' \in [\![q]\!](t) \wedge t' \in [\![p]\!]\}$;

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

$$\begin{array}{rcl} \textbf{\textit{restrict\_out}}\ q\ p & = & \textbf{\textit{compose}}\ q\ (\textbf{\textit{restrict}}\ I\ p) \\ \textbf{\textit{pre-image}}\ q\ p & = & \textbf{\textit{domain}}\ (\textbf{\textit{restrict\_out}}\ q\ p) \end{array}$$

where 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 their ability to *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. The second reason, as illustrated in the next example, shows that duplication causes problems when combined with nondeterminism.

Example 8. Assume Σ has rank 2, $\Sigma(2) = \{\mathtt{a}, \mathtt{b}\}$ and $\Sigma(0) = \{\epsilon\}$. f_1 is the transformation that given a tree $t \in \textbf{\textit{Tree}}_{\boldsymbol{\Sigma}}$ nondeterministically swaps as, and bs. For example on the input $\mathtt{a}(\epsilon, \epsilon)$ f_1 produces $\mathtt{a}(\epsilon, \epsilon)$ itself and $\mathtt{b}(\epsilon, \epsilon)$. f_2 is the function that for every $t \in \textbf{\textit{Tree}}_{\boldsymbol{\Sigma}}$ outputs the tree $\mathtt{a}(t, t)$. 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 $\{\mathtt{a}(t', t') | t' \in f_1(t)\}$. 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 $\boldsymbol{f} : X \to 2^Y$ and $\boldsymbol{x} \subseteq X$, $\boldsymbol{f}(\boldsymbol{x}) \stackrel{\text{def}}{=} \bigcup_{x \in \boldsymbol{x}} \boldsymbol{f}(x)$. Given $\boldsymbol{f} : X \to 2^Y$ and $\boldsymbol{g} : Y \to 2^Z$, $\boldsymbol{f} \circ \boldsymbol{g}(x) \stackrel{\text{def}}{=} \boldsymbol{g}(\boldsymbol{f}(x))$. This definition follows the convention in [5], i.e., $\circ$ applies first $\boldsymbol{f}$, then $\boldsymbol{g}$, contrary to how $\circ$ is used for standard function composition.

We first present an algorithm that given two STTRs S and T such that $\tau'_S = \tau_T$ constructs a composed STTR $S \circ T$ such that $\tau_{S \circ T} = \tau_S$ and $\tau'_{S \circ T} = \tau'_T$. We then prove that, under the assumptions mentioned above, $\mathscr{T}_{S \circ T} = \mathscr{T}_S \circ \mathscr{T}_T$. 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 $AND(t_1, t_2)$ for the term $Node[a](t_1, t_2)$ to symbolically represent a conjunction using a tree term, and we write $TRUE$ for $Leaf[a]$, where a is some fixed attribute value. We assume that $R_S \subseteq Q_S$ and $R_T \subseteq Q_T$ and that each look-ahead rule such as $(p, Node, \varphi, (\{p_1\}, \{p_2\}))$ is represented by a (domain-equivalent) transduction rule of the form

$$(p, Node, \varphi, \_, \lambda(x, y_1, y_2).AND(\widehat{p_1}(y_1), \widehat{p_2}(y_2)))$$

```
1  S ∘ T ≝
2    S := Normalize(S); T := Normalize(T);
3    q⁰ := (q_S⁰, q_T⁰); Q := {q₀} ∪ R_S; Δ := δ_S; R := R_S;
4    Stack := [q⁰];
5    while Stack ≠ []
6      pop (p, q) from Stack;
7      foreach r ∈ Transduce(p, q) add r to Δ;
8    return Normalize(Q, q⁰, τ_S, τ'_T, Δ, R, {r ∈ Δ | r.q ∈ R});
9
10 Transduce(p, q) ≝
11   foreach (p, c, φ, ρ, f) ∈ Δ_S
12     foreach (g, ψ, ℓ) ∈ TransduceT(q, f, φ, ρ)
13       yield ((p, q), c, ψ, ℓ, g);
14
15 TransduceT(q, Leaf[t₀], ξ, ℓ) ≝
16   foreach (q, Leaf, φ, ρ, f) ∈ Δ_T
17     if IsSat(ξ ∧ φ(t₀)) then yield (f(t₀), ξ ∧ φ(t₀), ℓ);
18 TransduceT(q, Node[t₀](t₁, t₂), ξ, ℓ) ≝
19   foreach (q, Node, φ, f, ρ) ∈ Δ_T;
20     if IsSat(ξ ∧ φ(t₀)) then
21       foreach (_, ξ', ℓ') ∈ Reduce(ρ(t₁, t₂), ξ ∧ φ(t₀), ℓ)
22         foreach v ∈ Reduce(f(t₀, t₁, t₂), ξ', ℓ') yield v;
23 TransduceT(q, p̂(y_i), ξ, ℓ) ≝
24   o := (p, q);
25   if o ∉ Q then { add o to Q; push o to Stack; }
26   if q ∈ R_T then { add o to R; ℓ := AND(ô(y_i), ℓ); }
27   yield (ô(y_i), ξ, ℓ);
28
29 Reduce(Leaf[t₀], ξ, ℓ) ≝
30   yield (Leaf[t₀], ξ, ℓ);
31 Reduce(Node[t₀](t₁, t₂), ξ, ℓ) ≝
32   foreach (u₁, ξ₁, ℓ₁) ∈ Reduce(t₁, ξ, ℓ)
33     foreach (u₂, ξ₂, ℓ₂) ∈ Reduce(t₂, ξ₁, ℓ₁)
34       yield (Node[t₀](u₁, u₂), ξ₂, ℓ₂);
35 Reduce(q̂(t), ξ, ℓ) ≝ TransduceT(q, t, ξ, ℓ);
```

Figure 6: Composition algorithm of STTRs.

where the look-ahead component is irrelevant. In transduction rules, the output Λ and continuation θ are combined into a single term $\Lambda(x, \theta[1], \ldots, \theta[|\theta|])$ where each pair $(p, i) \in \theta$ is considered as a continuation point term $\widehat{p}(y_i)$.

In the constructed STTR $S \circ T$ we have $Q_{S \circ T} \subseteq Q_S \times Q_T \cup R_S$ 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, **Transduce**, symbolically composes 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]

[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 the 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 $r = (q, \mathit{Node}, \varphi, (\boldsymbol{q}_1, \boldsymbol{q}_2), f)$, $\boldsymbol{q}_i$ is a singleton set and, if $t_1 \in \boldsymbol{L}^{\boldsymbol{q}_1}$ and $t_2 \in \boldsymbol{L}^{\boldsymbol{q}_2}$, 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 $A = (R, \mathit{Tree}\langle\sigma\rangle, \delta)$ be an STA. We are going to compute *merged rules* $(\boldsymbol{q}, c, \boldsymbol{\varphi}, \boldsymbol{\rho})$ over *merged states* $\boldsymbol{q} \in 2^R$ where $\boldsymbol{\rho} \in 2^R \times 2^R$ and $\boldsymbol{\varphi}$ is a set of predicates considered as a conjunction. We define the new transition relation $\boldsymbol{\delta}^c$ as follows:

$$\begin{aligned}\boldsymbol{\delta}^c(\emptyset) &\stackrel{\text{def}}{=} \{(\emptyset, c, \emptyset, \emptyset^{\natural(c)})\}\\ \boldsymbol{\delta}^c(\{p\}) &\stackrel{\text{def}}{=} \{(\{p\}, c, \{\varphi\}, \rho) \mid (p, c, \varphi, \rho) \in \delta\}\\ \boldsymbol{\delta}^c(\boldsymbol{p} \cup \boldsymbol{q}) &\stackrel{\text{def}}{=} \{r \mathbin{/\!\!\backslash\!\!/} s \mid r \in \boldsymbol{\delta}^c(\boldsymbol{p}), s \in \boldsymbol{\delta}^c(\boldsymbol{q})\}\end{aligned}$$

$$(\boldsymbol{p}, c, \boldsymbol{\varphi}, (\boldsymbol{p}_i)_{i=1}^{\natural(c)}) \mathbin{/\!\!\backslash\!\!/} (\boldsymbol{q}, c, \boldsymbol{\psi}, (\boldsymbol{q}_i)_{i=1}^{\natural(c)}) \stackrel{\text{def}}{=} (\boldsymbol{p} \cup \boldsymbol{q}, c, \boldsymbol{\varphi} \cup \boldsymbol{\psi}, (\boldsymbol{p}_i \cup \boldsymbol{q}_i)_{i=1}^{\natural(c)})$$

Algorithmically, merged rules are computed lazily and those with unsatisfiable conditions $\bigwedge_{\varphi \in \boldsymbol{\varphi}} \varphi$ 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 $Q \cup R$ that are either unreachable or *deadends* (no tree is accepted from q). For example, if the only rule from a state $q \in R$ is $(q, \mathit{Node}, \top, (q, q))$ then q is a deadend because q 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 nonterminals 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, $1 \le i \le \natural(r.c)$, the continuation $r.\theta$ 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 $(p, q) \in Q_{S\circ T}$, and all $t \in \boldsymbol{\tau_S}$, $v \in \boldsymbol{\tau'_T}$:*

$$\exists u(\boldsymbol{T}_S^p(t, u) \wedge \boldsymbol{T}_T^q(u, v)) \Leftrightarrow \boldsymbol{T}_{S\circ T}^{(p,q)}(t, v)$$

The proof of Lemma 1 is by induction over t. It uses laws of distributivity, in particular that $\exists$ 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 $\mathscr{T}_{S\circ T} = \mathscr{T}_S \circ \mathscr{T}_T$.*

Proof. The statement $v \in (\mathscr{T}_S \circ \mathscr{T}_T)(t)$ is (by definition) equivalent to $\exists u(\boldsymbol{T}_S^{q_S^0}(t, u) \wedge \boldsymbol{T}_T^{q_T^0}(u, v))$, which by Lemma 1, is equivalent to $\boldsymbol{T}_{S\circ T}^{(q_S^0, q_T^0)}(t, v)$, i.e., $v \in \mathscr{T}_{S\circ T}(t)$ (since $(q_S^0, q_T^0) = q_{S\circ T}^0$).

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 $|Q_S \times Q_T|$. 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 $|Q_S \times Q_T|$ 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 $S \circ T$ 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.

Program	LOC
nondet.fast	7
domain.fast	11
apply.fast	13
lists.fast	14
functions.fast	15
intersect.fast	15
union.fast	15
restrict.fast	21
constants.fast	23
trees.fast	32
sample.fast	35
deforestation.fast	52
compose.fast	65
htmlSanitizer.fast	101
bodytree.fast	173

Figure 7: FAST programs.

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 `<`. 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 `<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 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

[6] The corresponding statement in C# is `yield return`.

Site	Size	Time		Size After	
		Fast	Purifier	Fast	Purifier
Amazon.com	219	119	185	127	6
Bing.com	20	8	29	8	9
Cnn.com	98	88	247	86	78
Economist.com	75	42	184	49	49
Facebook.com	409	75	306	218	9
Gmail.com	104	5	21	3	3
Google.com	25	16	35	15	10
Wikipedia.com	52	61	126	59	45
Yahoo.com	293	81	265	120	103
Yelp.com	48	49	143	40	39
Average	134	54	154	71	35

Figure 8: Applying HTML sanitization to popular sites.

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 `<script>` blocks, attacked to HTML elements as handlers, and even hidden in style sheets and included elements.

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. Our version on an HTML sanitizer written in Fast and automatically translated by the Fast compiler into C# is partially shown in Figure 2 and is described in Section 2.

Performance: To compare different sanitization strategies in terms of performance, we chose 10 popular large-scale web sites and obtained their HTML content, ranging from 20 KB (Bing) to 409 KB in size (Facebook). Figure 8 shows a comparison of both the obtained sizes post-sanitization as well as sanitizer running times.

Overall, the Fast-based sanitizer is both faster (3.1x on average), amenable to reasoning and analysis, and considerably more maintainable than HTML Purify (101 lines vs. over 10,000 lines). Clearly, HTML Purify is considerably more aggressive at stripping out relevant content, often resulting in much smaller HTML document outputs, but this is not necessarily a good feature. Indeed, we want a sanitizer that produces safe output, while keeping as much of the original document intact as possible.

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

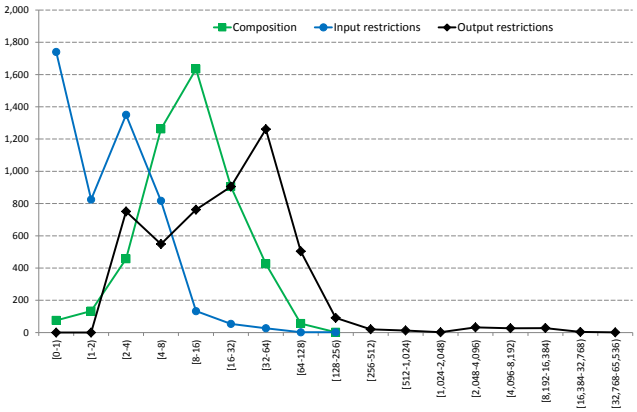

Figure 9: Running times for operations on transducers; the x-axis represent time intervals in ms. The y-axis shows how many cases run in a time belonging to an interval. For example in plot (a) about 1,600 compositions took between 16 and 32 milliseconds.

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 $\langle p_1, p_2 \rangle$:

1. [**composition**] we compute p, composition of p_1 and p_2;
2. [**input restriction**] we compute p', a restriction of p that only accepts trees where each node contains no tags;
3. [**output restriction**] we compute p'', a restriction of p' that only outputs trees where at least one node has been tagged twice;
4. [**check**] we check if p'' is the empty transducer: if this is the case p_1 and p_2 do not conflict, otherwise they conflict on every input accepted by p''.

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 terminated in a time between 8 and 16 ms. All the compositions are computed in less than 250 ms. 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:

```
type RList[i : Int]{nil(0), cons(1)}
trans map : RList->RList {
      nil() to (nil[0])
    | cons(y) to (cons [(f i)] (map y))
}
trans filter : RList->RList {
      nil() to (nil[0])
    | cons(y) where (p i) to (cons [i] (filter y))
    | cons(y) where (not(p i)) to (filter y)
}
def map_filter : RList -> RList := (compose map filter)
```

Figure 10: Two common functional programs over lists implemented in FAST: *filter* and *map*.

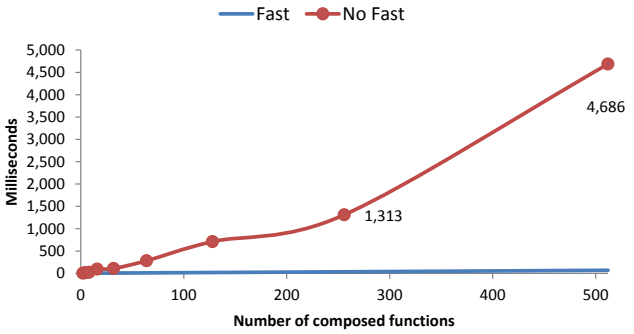

Figure 11: Deforestation advantage for a list of 4,096 integers.

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 version to have up to 5,000 states and 100,000 rules;
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 constraints over reals of degree 3.

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

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:

```
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 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. Kühnemann *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: *filter* and *map*. The programs transform lists of reals into lists of reals. We assume $p : real \mapsto bool$ and $f : real \mapsto real$ to be some previously defined predicate and function respectively. The program *map_filter* will compute the same function as applying *map* first and then *filter* but in a single pass.

We analyze how effective composition is in our setting. We consider the function *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 *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

Language	Alphabet	Analyzable	Domain
FAST	infinite	yes	Tree-manipulating programs
TTT	finite	no	NLP
TIBURON	finite	yes	NLP
XPath	infinite	fragment	XML query (only Selection)
XSLT	infinite	no	XML transformations

Figure 12: Languages for tree-manipulating programs.

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 $S(S(\ldots(0)\ldots))$. 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 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

[1] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and xml reasoning. In *Proceedings of the twenty-fifth symposium on Principles of database systems*, PODS '06, pages 10–19, 2006.

[2] E. Chien. Malicious yahooligans. `http://www.symantec.com/avcenter/reference/malicious.yahooligans.pdf`, 2006.

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

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

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

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

[7] W. Hodges. *Model theory*. Cambridge Univ. Press, 1995.

[8] 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.

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 1979.

[10] A. Kühnemann and J. Voigtländer. Tree transducer composition as deforestation method for functional programs, 2001.

[11] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 495–508, 2010.

[12] Y. Mansour and D. Mcallester. Boosting using branching programs. *Journal of Computer and System Sciences*, 64:2002, 2000.

[13] J. May and K. Knight. A primer on tree automata software fornatural language processing, 2008.

[14] A. Purtee and L. Schubert. TTT: A tree transduction language for syntactic and semantic processing. In *Proceedings of the Workshop on Applications of Tree Automata Techniques in Natural Language Processing*, pages 21–30, 2012.

[15] RSnake. Month of MySpace bugs. `http://ha.ckers.org/blog/20070322/month-of-myspace-bugs/`, Mar. 2007.

[16] Samy. The Samy worm. `http://namb.la/popular/`, 2005.

[17] O. Sima. Post-pruning in C4.5. `http://octaviansima.wordpress.com/2011/03/25/decision-trees-c4-5/`.

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

[19] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finate state transducers: Algorithms and applications. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'12)*, 2012.

[20] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP*, pages 344–358, 1988.

[21] World Wide Web Consortium. XML path language, 1999.

[22] World Wide Web Consortium. XSL transformation, 1999.

[23] Z3. `http://research.microsoft.com/projects/z3`.