FAST: a Transducer Based Language for Manipulating Trees

Presented By:
Loris D’Antoni

Joint work with:
Margus Veanes, Ben Livshits, David Molnar
Motivation

**Trees** are common input/output data structures

- **XML** query, type-checking, etc...
- **Compilers**/optimizers (from parse tree to parse tree)
- Tree manipulating programs: **data structures** algorithms, **ontologies**, etc...
HTML Sanitization

Removing **malicious** active code from HTML documents is a **tree transformation**.

```
<body>
  <script>
    malicious code
  </script>
  <div>
    <p>“Today I’m happy”</p>
  </div>
</body>

SANITIZE

```

```
<body>
  <div>
    <p>“Today I’m happy”</p>
  </div>
</body>
```
What do we Need?

- **Remove** bad elements (scripts...)
- **Remove** malicious URLs
- **Replace** deprecated tags

We want to **write** these single transformations **separately** to avoid errors
### Interesting Properties

<table>
<thead>
<tr>
<th>Composition: $T(x) = T_2(T_1(x))$</th>
<th>To achieve <strong>speed</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type-checking:</strong> given two languages $I, O$ $T(I)$ is always in $O$</td>
<td>Check if the sanitizer ever produces a malicious output</td>
</tr>
<tr>
<td><strong>Pre-image:</strong> compute the input that produces a particular output</td>
<td>Produce <strong>counterexamples</strong> if type-checking fails</td>
</tr>
</tbody>
</table>

**DEMO:** [http://rise4fun.com/Fast/jN](http://rise4fun.com/Fast/jN)
FAST Compiler

- FAST code
- SMT solver
  - Transducers
  - Analysis and optimization
- C#
Stages by Example

```
fun f (i:int) : int := (mod (+ i 5) 26)
alphabet IList [i:int] {nil(0),cons(1)}
public trans mapC : IList -> IList {
    nil() to (nil [0])
    | cons(x) to (cons [(f i)] (mapC| x))
}
```

```
public IEnumerable<TreeIList> mapC2()
{
    if (this.symbol == IList.nil)
    {
        if (true)
        {
            yield return TreeIList.MakeTree(IList.nil, 0, new TreeIList[0] { });
        }
    }
    if (this.symbol == IList.cons)
    {
        if (true)
        {
            IEnumerable<TreeIList> _mapC2_x1 = children[0].mapC2();
            foreach (var mapC2x1 in _mapC2_x1)
            {
                yield return TreeIList.MakeTree(IList.cons, ((5 + ((5 + (i % 26)) % 26)) % 26), new TreeIList[1] { mapC2x1 });
            }
            if (false) yield return null;
        }
    }
```

Transducers
mapC  mapC2
CHOOSING THE RIGHT FORMALISM
Semantics as Transducers

Goal:

find a \textbf{decidable} class

of \textbf{tree transducers}

that can \textbf{express} the previous examples
Top Down Tree Transducers
[Engelfriet75]

\[ q(a(x_1,x_2)) \rightarrow b(c,q_1(x_1)) \]

Decidable properties: type-checking, etc...

Domain expressiveness: only finite alphabets
Symbolic Tree Transducers [PSI11]

Decidable properties:
- type-checking, etc...

Domain expressiveness:
- infinite alphabets using predicates and functions

Structural expressiveness:
- can’t delete a node without reading it first

Alphabet theory has to be DECIDABLE
We’ll use Z3 to check predicate satisfiability
Improving structural expressiveness

Transformation: delete the left child if its root greater than 5

If we delete the node we can’t check that the left child was actually greater than 5

Regular Look-Ahead (RLA)
Regular Look Ahead (TOPR)

**Transformation:** delete a node if its left child is greater than 5

1. $q \leftarrow 1$
2. $p_1 \leftarrow 7$
3. $p_2 \leftarrow 3$
4. $\rightarrow$
5. $1 \rightarrow q$
6. $3 \rightarrow$

**Rules can ask** whether the children are in particular languages
- $p_1$: the language of trees whose root is greater than 5
- $p_2$: the language of all trees

**Decidable properties:**
- type-checking, etc...

**Domain expressiveness:**
- infinite alphabets

**Structural expressiveness:**
- good enough to express our examples
<table>
<thead>
<tr>
<th>Top Down Tree Transducers [Engelfriet75]</th>
<th>Decidability</th>
<th>Complexity</th>
<th>Structural Expressiveness</th>
<th>Infinite alphabets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top Down Tree Transducers with Regular Look-ahead [Engelfriet76]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
<td>x</td>
</tr>
<tr>
<td>Streaming Tree Transducers [AlurDantoni12]</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Data Automata [Bojanczyk98]</td>
<td>~</td>
<td>x</td>
<td>x</td>
<td>v</td>
</tr>
<tr>
<td>Symbolic Tree Transducers [VeanesBjoerner11]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Symbolic Tree Transducers RLA</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
<td>v</td>
</tr>
</tbody>
</table>
COMPOSITION OF SYMBOLIC TRANSDUCERS WITH REGULAR LOOKAHEAD
Composition of $\text{STT}^R$

This is not always possible!!
Find the biggest class for which it is possible
Classes of STTR

**DETERMINISTIC:** at most one transducer rule applies for each input tree

**LINEAR:** each child appear at most once in the right hand side of each rule
When can we Compose?

Theorem: \[ T(x) = T_2(T_1(x)) \]
definable by a Symbolic Tree Transducers with RLA if

- \( T_1 \) is deterministic, OR
- \( T_2 \) is linear

All our examples fall in this category

Alphabet theory has to be **DECIDABLE**
We’ll use Z3 to check predicate satisfiability
Pre-image as Composition

Domain\((T \circ O)\)
FAST: Decidable by Design

Composition
Type-checking
Pre-image

Symbolic Tree Transducers with RLA

SMT Solver for Alphabet Theory
CASE STUDIES AND EXPERIMENTS
Case Studies and Experiments

Program Optimization:
Deforestation of functional programs

Verification:
HTML sanitization
Analysis of functional programs
Augmented reality app store

Infinite Alphabets:
Integer
Data types
Deforestation

Removing **intermediate** data structures from programs

```plaintext
alphabet IList [i : int] { nil(0), cons(1) }

trans mapC: IList → IList {
    nil() → nil [0]
    | cons(x) → cons [(i+5)%26] (mapC x)
}

def mapC²: IList → IList := compose mapC mapC
```

**ADVANTAGE:** the program is a single transducer reads the input list **only once**, thanks to transducers **composition**
Deforestation: Speedup

- **Fast**
- **No Fast**

Mathematical notation:
\[ f(f(f(\ldots f(x)\ldots)) \]

Graph showing the speedup in milliseconds vs. the number of composed map functions.
Analysis of Functional Programs

// Increments all the elements of the list by 1
Public Trans map_inc : IntList -> IntList {
    nil() to (nil [0])
    | cons(x) to (cons [(inc i)] (map_inc x))
}

// Removes all the odd elements from the list
Public Trans filter_ev : IntList -> IntList {
    nil() to (nil [0])
    | cons(x) where (odd i) to (filter_ev x)
    | cons(x) where (even i) to (cons [i] (filter_ev x))
}

// Compose the four functions into a single one
Def map_filt_2 : IntList -> IntList := (compose (compose map_inc filter_ev) (compose map_inc filter_ev))

// Empty list language
Public Lang not_emp_list : IntList {
    nil()
}

// Non-empty list language
Public Lang not_emp_list : IntList {
    cons(x)
}

// Check whether map_filt_2 ever outputs a non-empty list
Def map_filt_2_rest : IntList -> IntList := (restrict_out map_filt_2 not_emp_list)
AssertTrue (is_empty_trans map_filt_2_rest)
Recognizers output data that can be seen as a tree structure.
Apps as Tree Transformations

Applications that use recognizers can be modeled as **FAST** programs

```plaintext
trans addHat: STree -> STree
   Spine(x,y) to Spine(addHat(x), y)
   | Neck(h,l,r) to Neck(addHat(h), l, r)
   | Head(a) to Head(Hat(a))
```
Composition of Programs

Two **FAST** programs can be composed into a single **FAST** program.

\[ p_1 ; p_2 \]
Interference analysis

Apps can be **malicious**: try to overwrite outputs of other apps

Apps **interfere** when they annotate the **same node** of a recognizer’s output

<table>
<thead>
<tr>
<th>Interfering apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add cat ears</td>
</tr>
<tr>
<td>Add hat</td>
</tr>
<tr>
<td>Add pin to a city</td>
</tr>
<tr>
<td>Blur a city</td>
</tr>
<tr>
<td><strong>Amazon Buy Now button</strong></td>
</tr>
<tr>
<td><strong>Malicious Buy Now button</strong></td>
</tr>
</tbody>
</table>

We can **compose** them and check if they interfere **statically**!!

– Put checker in the **AppStore** and analyze Apps before approval
Interference Analysis in Practice

100 generated FAST programs, up to **85 functions** each

Check statically if they **conflict** pairwise for **ANY** possible input

Checked 99% of program pair in less than 0.5 sec!

For an App store these are perfectly fine
Conclusion

**FAST:** a versatile language for tree manipulating programs with decidable analysis

Symbolic tree transducers with RLA

**FAST is online:** [http://rise4fun.com/Fast/](http://rise4fun.com/Fast/)