Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation

Yanju Chen ¹, Junrui Liu ¹, Yu Feng ¹, Rastislav Bodik ²

¹University of California, Santa Barbara
²University of Washington
Tree Traversals

- Motivations -

Tree traversals are widely used and play important roles.

- Compilers
- Web Browsers
- Numerical Computations
- Motivations -

A Motivating Example

- Synthesizing A Toy Layout Engine
- Two classes, Four Attributes
- Attribute Grammar
**Existing Approaches & Challenges**

- **Automata Based:** TreeFuser\(^1\) and GRAFT\(^2\)
  - Deterministic Rewrite Rules (Complex to Maintain)

- **Synthesis Based:** FTL\(^3\)
  - Constraints Generated by Domain Experts (Manual and Error-Prone)

- **General-Purpose Symbolic Compilation**
  - Solver-Aided Programming Languages, e.g., Rosette\(^4\)
  - Path Explosions & Complex Constraint System

---

Overview: HECATE

- A CEGIS Framework for Tree Traversal Synthesis
- A Domain-Specific Trace Language
  - For Disentangling Complex Dependencies in Trees
  - For Generating Easy-to-Solve Constraints for Tree Traversal Synthesis
- A Tool Called HECATE
  - For Expressive, Efficient and Flexible Tree Traversal Synthesis
- Synthesis Using HECATE -

Attribute Grammar & Traversal Language

\[
\begin{align*}
\langle \text{interface} \rangle & \ ::= \text{interface } \langle \text{id} \rangle \ \{ \langle \text{tup} \rangle ; \}^* \\
\langle \text{class} \rangle & \ ::= \text{class } \langle \text{tup} \rangle \ \{ \langle \text{children} \rangle \ \langle \text{rules} \rangle \} \\
\langle \text{children} \rangle & \ ::= \text{children } \{ \langle \text{tup} \rangle ; \}^* \\
\langle \text{rules} \rangle & \ ::= \text{rules } \{ \langle \text{cstmt} \rangle ; \}^* \\
\langle \text{tup} \rangle & \ ::= \langle \text{id} \rangle ; \langle \text{id} \rangle , (\langle \text{id} \rangle )^* \\
\langle \text{sel} \rangle & \ ::= \langle \text{id} \rangle . \langle \text{id} \rangle ? . \langle \text{id} \rangle \\
\langle \text{expr} \rangle & \ ::= \langle \text{const} \rangle \mid \langle \text{sel} \rangle \mid f(\langle \text{expr} \rangle )^* \mid \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mid \text{fold} (\langle \text{expr} \rangle + ) \mid \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle \\
\langle \text{cstmt} \rangle & \ ::= \langle \text{sel} \rangle := \langle \text{expr} \rangle \\
\langle \text{op} \rangle & \ ::= + \mid - \mid \times \mid \div \mid \ldots
\end{align*}
\]

\[f \in \text{functions} \quad \langle \text{const} \rangle \in \text{constants} \quad \langle \text{id} \rangle \in \text{identifiers}\]

Figure 6: Syntax for attribute grammar $L_a$.

\[
\begin{align*}
\langle \text{traversal} \rangle & \ ::= \text{traversal } \langle \text{id} \rangle \ \{ \langle \text{case} \rangle ; \}^* \\
\langle \text{case} \rangle & \ ::= \text{case } \langle \text{id} \rangle \ \{ \langle \text{tstmt} \rangle ; \}^* \\
\langle \text{recur} \rangle & \ ::= \text{recur } \langle \text{node} \rangle \\
\langle \text{iterate} \rangle & \ ::= \text{iterate } \{ \langle \text{tstmt} \rangle ; \}^* \\
\langle \text{parallel} \rangle & \ ::= \text{parallel } \{ \langle \text{tstmt} \rangle ; \}^* \\
\langle \text{eval} \rangle & \ ::= \text{eval } \langle \text{cstmt} \rangle \\
\langle \text{tstmt} \rangle & \ ::= \text{t} \mid \langle \text{recur} \rangle \mid \langle \text{iterate} \rangle \mid \langle \text{eval} \rangle
\end{align*}
\]

\[\langle \text{id} \rangle \in \text{identifiers} \quad \langle \text{node} \rangle \in \text{nodes}\]

Figure 7: Syntax for tree traversal language $L_t$.

* Please refer to the paper for more details.
**General-Purpose Symbolic Compilation**

- **Constraint System**
  - **Semantic Constraints**
    
    \[(\sigma(\text{none}, t_2) \Rightarrow \text{true})\]
    
    \[\forall (\sigma(\text{Inner.w1}, t_2) \Rightarrow \delta(\zeta(n_1, \text{self.w}), t) \land \delta(\zeta(n_1, \text{nx.w1}), t) \land \neg\delta(\zeta(n_1, \text{self.w1}), t))\]
    
    \[\forall (\sigma(\text{Inner.w}, t_2) \Rightarrow \delta(\zeta(n_1, \text{self.w0}), t) \land \delta(\zeta(n_1, \text{fc.w1}), t) \land \neg\delta(\zeta(n_1, \text{self.w1}), t))\]
    
    \[\forall (\sigma(\text{Inner.h1}, t_2) \Rightarrow \delta(\zeta(n_1, \text{self.h}), t) \land \delta(\zeta(n_1, \text{nx.h1}), t) \land \neg\delta(\zeta(n_1, \text{self.h1}), t))\]
    
    \[\forall (\sigma(\text{Inner.h}, t_2) \Rightarrow \delta(\zeta(n_1, \text{self.h0}), t) \land \delta(\zeta(n_1, \text{fc.h1}), t) \land \neg\delta(\zeta(n_1, \text{self.h1}), t))\]

  - **Auxiliary Constraints**
    
    \[\forall t. (\bigvee_{a \neq a_0} \neg \sigma(a, i) \land \sigma(a_0, i)) \lor (\bigvee_a \neg \sigma(a, i)).\]
    
    \[\forall a. \bigvee_{t_0, t_0 \in t_0} \neg \sigma(a, i) \land \sigma(a, i_0).\]

- **Synthesis Using HECATE**

- Every slot should be filled with at most one rule.
- Every rule should be used by only one slot.

- All dependencies should have been ready.
- Target attribute has not been scheduled.

Number of timesteps grows as example trees become larger, which increases the complexity.
- Synthesis Using HECATE -

Domain-Specific Symbolic Compilation

- **[Traversal]** Given a tree, a traversal defines a total order relation $\prec$ over the set of all locations of the tree.

- **[Example]** A concrete post-order traversal on the example tree yields the following total order of locations:

  \[
  n_4.w < n_4.h < n_4.w_1 < n_4.h_1 < n_3.w < n_3.h < n_3.w_1 < n_3.h_1 < n_1.w < n_1.h < n_1.w_1 < n_1.h_1 < n_2.w < n_2.h < n_2.w_1 < n_2.h_1 < n_0.w < n_0.h < n_0.w_1 < n_0.h_1
  \]

We can map a traversal from time domain to relational domain.

Such a traversal can be both concrete or symbolic.
Domain-Specific Symbolic Compilation

- **A Symbolic Trace Language**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(choose [a₁,...,aₙ])</td>
<td>choose one from the attributes</td>
</tr>
<tr>
<td>(alloc)</td>
<td>returns a fresh concrete location</td>
</tr>
<tr>
<td>(read n.a)</td>
<td>logs a read from n.a</td>
</tr>
<tr>
<td>(write n.a)</td>
<td>logs a write to n.a</td>
</tr>
</tbody>
</table>

\[
(\text{assume } \sigma(\text{Inner}.h, \nu_2) \\
(\text{read } n_1.h\theta) (\text{read } n_3.h1) (\text{write } n_1.h))
\]

- **[0-1 Integer Linear Programming]** Given coefficients \(a, b\) and \(c\), the 0-1 ILP problem is to solve for \(x\) as follows:

\[
\min \sum_i c_i x_i \quad s.t. \quad \forall i,j. \sum_j a_{i,j} x_j \leq b_i,
\]

where all entries are integers and in particular \(x_j \in \{0,1\}\).
Domain-Specific Symbolic Compilation

(assume $\sigma(\text{Inner.h}, t_2)$
(read $n_1.h0$) (read $n_3.h1$) (write $n_1.h$))

- Constraint System
  - Dependency Constraints
    \[
    \sigma(\text{Inner.h}, t_2) \leq \sum_{t_0 < t} \kappa[n_1.h0, t_0]
    = \sigma(\text{Inner.h0}, t_0) + \sigma(\text{Inner.h0}, t_1), \quad \text{(read for $n_1.h0$)}
    \]
    \[
    \sigma(\text{Inner.h}, t_2) \leq \sum_{t_0 < t} \kappa[n_3.h1, t_0]
    = \sigma(\text{Leaf.h1}, t_4) + \sigma(\text{Leaf.h1}, t_5)
    + \sigma(\text{Leaf.h1}, t_6) + \sigma(\text{Leaf.h1}, t_7), \quad \text{(read for $n_3.h1$)}
    \]
  - Validity Constraints
    \[
    \forall i. \sum_a \sigma(a, i) \leq 1. \quad \text{- Every slot should be filled with at most one rule.}
    \]
    \[
    \forall a. \sum_i \sigma(a, i) = 1. \quad \text{- Every rule should be used by only one slot.}
    \]

Constraints are not talking about $t$ anymore, but about domain-specific relations now.

"$n_1.h0$ should have been scheduled somewhere before the current corresponding location"

Every slot should be filled with at most one rule.

Every rule should be used by only one slot.
Complexity Analysis

- Synthesis Using HECATE -

* Please refer to the paper for more detailed analysis.
Evaluation

• Research Questions
  • [Performance] What is the performance of synthesized traversals, compared to those generated by state-of-the-art traversal synthesizers?
  • [Expressiveness] Is HECATE’s tree language expressive enough? In particular, can it express prevailing tree traversal synthesis problems and solve them?
  • [Flexibility] Can HECATE be extended to explore traversals of different design choices?
  • [Efficiency] What is the benefit of the domain-specific encoding compared to general-purpose encoding?
Comparison against GRAFTER\(^1\)

- GRAFTER
  - Static Dependence Analysis
  - Access Automata
- Benchmarks (Adapted from GRAFTER)
  - Five Real-World Representative Problem
    - Binary Search Tree
    - Fast Multipole Method
    - Piecewise Functions
    - Abstract Syntax Tree
    - Layout Rendering Tree

---

Table 2: Comparison between GRAFTER, HECATE and HECATE\(^G\) (with general-purpose encoding). The table shows total synthesis time (synthesis + verification) in second.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># of Rules</th>
<th>GRAFTER</th>
<th>HECATE</th>
<th>HECATE(^G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryTree</td>
<td>16</td>
<td>2.6</td>
<td>1.1</td>
<td>3.2</td>
</tr>
<tr>
<td>FMM</td>
<td>14</td>
<td>7.6</td>
<td>1.0</td>
<td>1.6</td>
</tr>
<tr>
<td>Piecewise</td>
<td>12</td>
<td>12.6</td>
<td>2.1</td>
<td>3.1</td>
</tr>
<tr>
<td>AST</td>
<td>136</td>
<td>151.7</td>
<td>20.6</td>
<td>73.4</td>
</tr>
<tr>
<td>RenderTree</td>
<td>50</td>
<td>62.0</td>
<td>4.1</td>
<td>10.1</td>
</tr>
</tbody>
</table>

A Case Study: RenderTree

- A Total of Five Rendering Passes
  1. Resolving Flexible Widths
  2. Resolving Relative Widths
  3. Computing Heights
  4. Propagating Font Styles
  5. Finalizing Element Positions

- Variants of Different Synthesizers
  - GRAFTER
  - HECATE\textsubscript{L}: Sequential, Linked List
  - HECATE\textsubscript{V}: Sequential, Vector
  - HECATE\textsubscript{P}: Parallel, Vector

With minimal efforts, Hecate can effectively explore traversals of different design choices.
Synthesizing Layout Engine in FTL\textsuperscript{[1]}

- FTL
  - Specialized for Layout Engine
  - Prolog Style Declarative Language for Partial Schedules
- Benchmarks (Adapted from FTL)
  - CSS-float
  - CSS-margin
  - CSS-full

<table>
<thead>
<tr>
<th>Name</th>
<th># of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS-float</td>
<td>192</td>
</tr>
<tr>
<td>CSS-margin</td>
<td>178</td>
</tr>
<tr>
<td>CSS-full</td>
<td>244</td>
</tr>
</tbody>
</table>

Figure 15: Comparison against FTL: benchmark statistics (left) and results (right).

Conclusion

- **HECATE**: A Novel Framework for Tree Traversal Synthesis
- Domain-Specific Symbolic Compilation
- Performance, Expressiveness, Flexibility and Efficiency

Thank you!

https://github.com/chyanju/Hecate