Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation

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

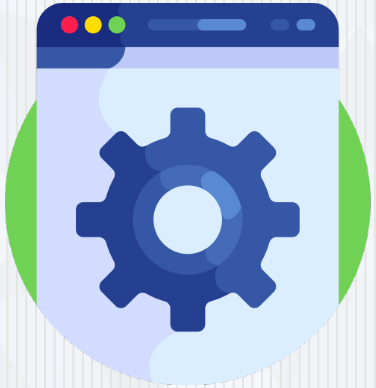
[1] University of California, Santa Barbara

[2] University of Washington

- Motivations -

Tree Traversals

Compilers



Web Browsers

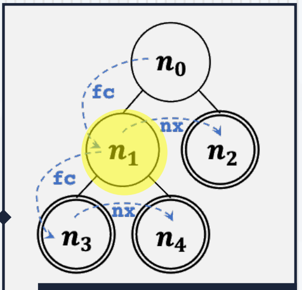
Numerical Computations

Tree traversals are widely used and play important roles.

- Motivations -

A Motivating Example

- Synthesizing A Toy Layout Engine
  - Two classes, Four Attributes
  - Attribute Grammar



example tree

```
interface Box{
  input w0,h0: int;
  output w1,w,h1,h: int;
}
class Inner: Box{
  children {
    nx : Optional[Box];
    fc : Optional[Box];
  }
  rules {
    self.w := max( self.w0, fc.w1 );
    self.w1 := max( self.w, nx.w1 );
    self.h := max( self.h0, fc.h1 );
    self.h1 := self.h + nx.h1;
  }
}
class Leaf: Box{
  children {
    nx : Optional[Box];
  }
  rules {
    self.w := self.w0;
    self.w1 := max( self.w, nx.w1 );
    self.h := self.h0;
    self.h1 := self.h + nx.h1;
  }
}
```

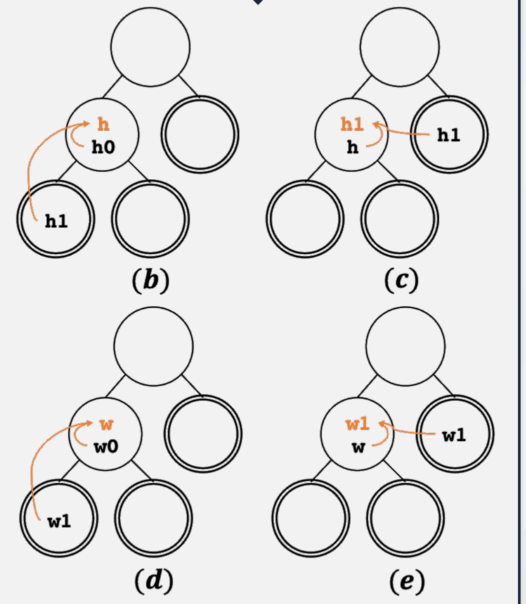
class definitions

```
traversal layout {
  case Inner{
    recur fc;
    recur nx;
    ι0;
    ι1;
    ι2;
    ι3;
  }
  case Leaf{
    recur nx;
    ι4;
    ι5;
    ι6;
    ι7
  }
}
```

symbolic traversal

```
traversal layout {
  case Inner{
    recur fc;
    recur nx;
    eval self.w;
    eval self.h;
    eval self.w1;
    eval self.h1;
  }
  case Leaf{
    recur nx;
    eval self.w;
    eval self.h;
    eval self.w1;
    eval self.h1;
  }
}
```

concrete traversal



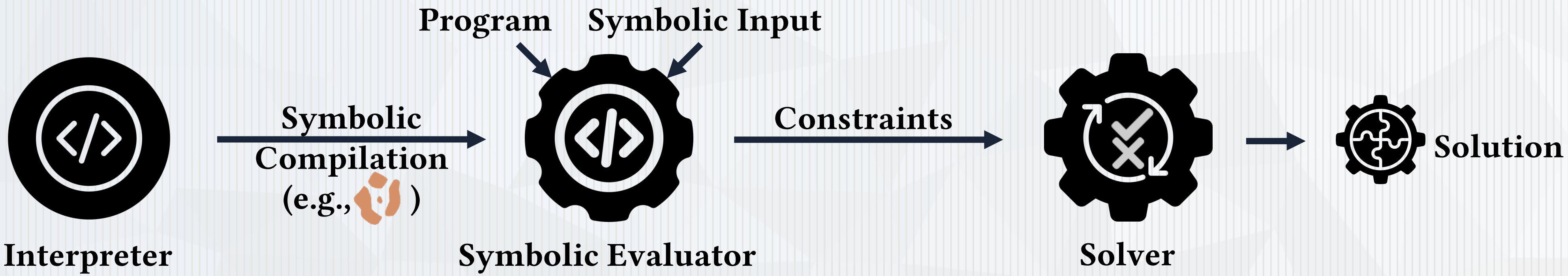
(*b*) (*c*) (*d*) (*e*)

dependencies in n_1's attributes

- Motivations -

Existing Approaches & Challenges

- Automata Based: TreeFuser[1] and Grafter[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



Program **Symbolic Input**

Interpreter → **Symbolic Compilation (e.g.,)** → **Symbolic Evaluator** → **Constraints** → **Solver** → **Solution**

---

[1] TreeFuser: a framework for analyzing and fusing general recursive tree traversals. Laith Sakka, Kirshanthan Sundararajah, Milind Kulkarni. OOPSLA 2017.

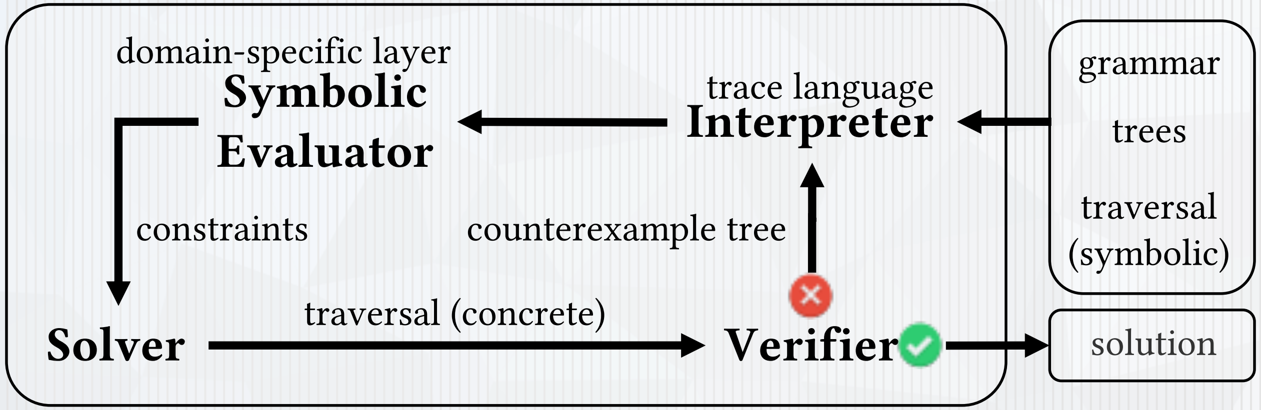
[2] Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees. Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, Milind Kulkarni. PLDI 2019.

[3] Parallel Schedule Synthesis for Attribute Grammars. Leo Meyerovich, Matthew Torok, Eric Atkinson, Rastislav Bodik. PPoPP 2013.

[4] A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. Emina Torlak, Rastislav Bodik. PLDI 2014.

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

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

$$f \in \textbf{functions} \quad \langle \mathit{const} \rangle \in \textbf{constants} \quad \langle \mathit{id} \rangle \in \textbf{identifiers}$$

Figure 6: Syntax for attribute grammar $\mathcal{L}_a$.

$$
\begin{array}{lcl}
\langle \mathit{traversal} \rangle & ::= & \text{traversal } \langle \mathit{id} \rangle \ \{ \ \langle \mathit{case} \rangle^{*} \ \} \\
\langle \mathit{case} \rangle & ::= & \text{case } \langle \mathit{id} \rangle \ \{ \ (\langle \mathit{tstmt} \rangle;)^{*} \ \} \\
\langle \mathit{recur} \rangle & ::= & \text{recur } \langle \mathit{node} \rangle \\
\langle \mathit{iterate} \rangle & ::= & \text{iterate} \ \{ \ (\langle \mathit{tstmt} \rangle;)^{*} \ \} \\
\langle \mathit{parallel} \rangle & ::= & \text{parallel} \ \{ \ (\langle \mathit{tstmt} \rangle;)^{*} \ \} \\
\langle \mathit{eval} \rangle & ::= & \text{eval } \langle \mathit{cstmt} \rangle \\
\langle \mathit{tstmt} \rangle & ::= & \iota \mid \langle \mathit{recur} \rangle \mid \langle \mathit{iterate} \rangle \mid \langle \mathit{eval} \rangle
\end{array}
$$

$$\langle \mathit{id} \rangle \in \textbf{identifiers} \quad \langle \mathit{node} \rangle \in \textbf{nodes}$$

Figure 7: Syntax for tree traversal language $\mathcal{L}_t$.

---

* Please refer to the paper for more details.

- Synthesis Using HECATE -

General-Purpose Symbolic Compilation

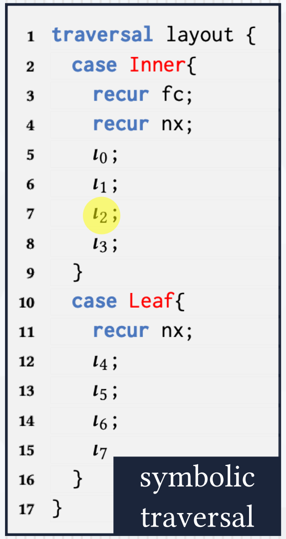
- Constraint System
  - Semantic Constraints

$$
\begin{aligned}
&(\sigma(\texttt{none},\iota_2) \implies \mathit{true})\\
\vee\ &(\sigma(\texttt{Inner.w1},\iota_2) \implies \delta(\zeta(n_1,\texttt{self.w}),t) \wedge \delta(\zeta(n_1,\texttt{nx.w1}),t)\\
&\qquad \wedge\neg\delta(\zeta(n_1,\texttt{self.w1}),t))\\
\vee\ &(\sigma(\texttt{Inner.w},\iota_2) \implies \delta(\zeta(n_1,\texttt{self.w0}),t) \wedge \delta(\zeta(n_1,\texttt{fc.w1}),t)\\
&\qquad \wedge\neg\delta(\zeta(n_1,\texttt{self.w}),t))\\
\vee\ &(\sigma(\texttt{Inner.h1},\iota_2) \implies \delta(\zeta(n_1,\texttt{self.h}),t) \wedge \delta(\zeta(n_1,\texttt{nx.h1}),t)\\
&\qquad \wedge\neg\delta(\zeta(n_1,\texttt{self.h1}),t))\\
\vee\ &(\sigma(\texttt{Inner.h},\iota_2) \implies \delta(\zeta(n_1,\texttt{self.h0}),t) \wedge \delta(\zeta(n_1,\texttt{fc.h1}),t)\\
&\qquad \wedge\neg\delta(\zeta(n_1,\texttt{self.h}),t))
\end{aligned}
$$

"choose one to schedule"

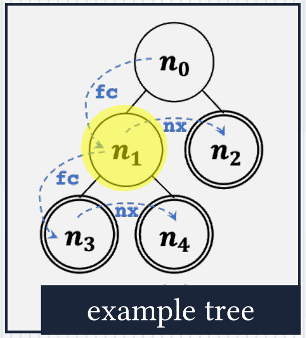
"all dependencies should have been ready"

"target attribute has not been scheduled"

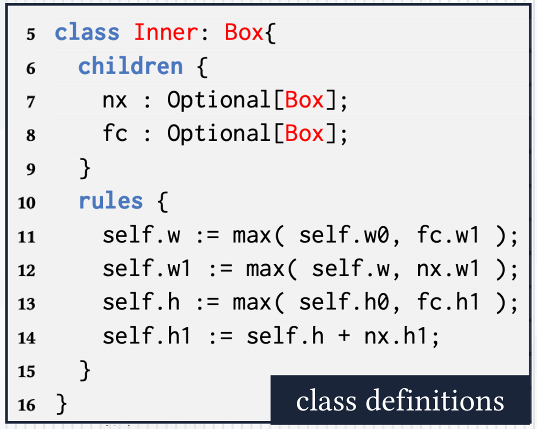


```
traversal layout {
  case Inner{
    recur fc;
    recur nx;
    ι0;
    ι1;
    ι2;
    ι3;
  }
  case Leaf{
    recur nx;
    ι4;
    ι5;
    ι6;
    ι7
  }
}
```

symbolic traversal



example tree



```
class Inner: Box{
  children {
    nx : Optional[Box];
    fc : Optional[Box];
  }
  rules {
    self.w := max( self.w0, fc.w1 );
    self.w1 := max( self.w, nx.w1 );
    self.h := max( self.h0, fc.h1 );
    self.h1 := self.h + nx.h1;
  }
}
```

class definitions

Number of timesteps grows as example trees become larger, which increases the complexity.

  - Auxiliary Constraints

$$\forall\iota.\Big(\bigvee_{a_0}\bigwedge_{a\neq a_0}\neg\sigma(a,\iota)\wedge\sigma(a_0,\iota)\Big)\vee\Big(\bigwedge_{a}\neg\sigma(a,\iota)\Big).$$

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

$$\forall a.\bigvee_{\iota_0}\bigwedge_{\iota\neq\iota_0}\neg\sigma(a,\iota)\wedge\sigma(a,\iota_0).$$

- Every rule should be used by only one slot.

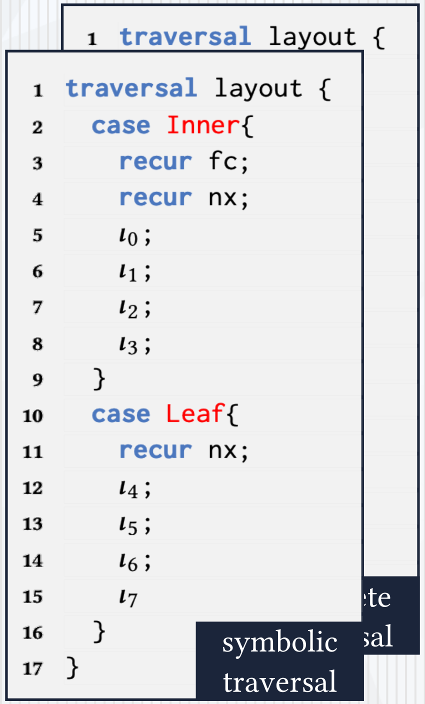
- Synthesis Using HECATE -

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:

$$n_4.\mathtt{w} \prec n_4.\mathtt{h} \prec n_4.\mathtt{w1} \prec n_4.\mathtt{h1} \prec n_3.\mathtt{w} \prec n_3.\mathtt{h} \prec n_3.\mathtt{w1} \prec n_3.\mathtt{h1}$$
$$\prec n_1.\mathtt{w} \prec n_1.\mathtt{h} \prec n_1.\mathtt{w1} \prec n_1.\mathtt{h1} \prec n_2.\mathtt{w} \prec n_2.\mathtt{h} \prec n_2.\mathtt{w1} \prec n_2.\mathtt{h1}$$
$$\prec n_0.\mathtt{w} \prec n_0.\mathtt{h} \prec n_0.\mathtt{w1} \prec n_0.\mathtt{h1}$$

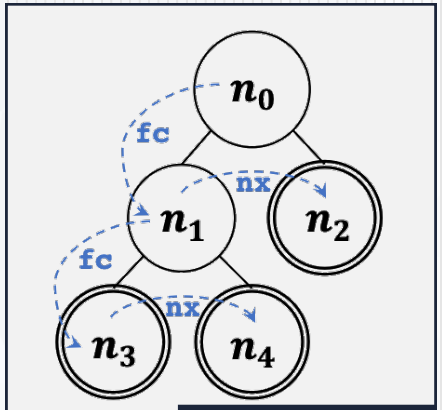


```
traversal layout {
```

```
traversal layout {
  case Inner{
    recur fc;
    recur nx;
    ι0;
    ι1;
    ι2;
    ι3;
  }
  case Leaf{
    recur nx;
    ι4;
    ι5;
    ι6;
    ι7
  }
}
```

te
sal

symbolic traversal



example tree

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

Such a traversal can be both concrete or symbolic.

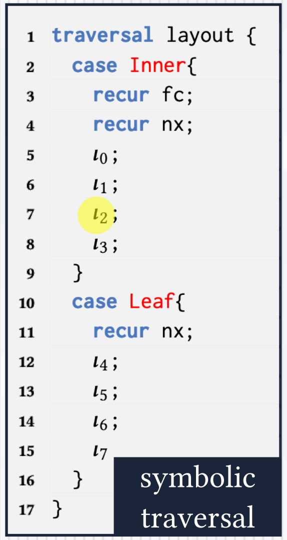
- Synthesis Using HECATE -

Domain-Specific Symbolic Compilation

- A Symbolic Trace Language

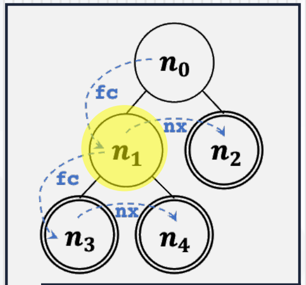
Operation	Description
`(choose [`$a_1, \ldots a_n$`])`	choose one from the attributes
`(alloc)`	returns a fresh concrete location
`(read` $n.a$`)`	logs a read from $n.a$
`(write` $n.a$`)`	logs a write to $n.a$

(assume σ(Inner.h, ι_2)
(read n_1.h0) (read n_3.h1) (write n_1.h))



```
traversal layout {
  case Inner{
    recur fc;
    recur nx;
    ι0;
    ι1;
    ι2;
    ι3;
  }
  case Leaf{
    recur nx;
    ι4;
    ι5;
    ι6;
    ι7
  }
}
```

symbolic traversal



example tree

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

$$\min \sum_i c_i x_i \quad s.t.\ \forall a_{i,j}. \sum_j a_{i,j} x_j \le b_i,$$

where all entries are integers and in particular $x_j \in \{0,1\}$.

- Synthesis Using HECATE -

Domain-Specific Symbolic Compilation

```
(assume σ(Inner.h, ι2)
        (read n1.h0) (read n3.h1) (write n1.h))
```

- Constraint System
  - Dependency Constraints

$$\sigma(\texttt{Inner.h}, \iota_2) \le \sum_{t_0<t} \kappa[n_1.\texttt{h0}, t_0] = \sigma(\texttt{Inner.h0}, \iota_0) + \sigma(\texttt{Inner.h0}, \iota_1), \qquad \text{(read for } n_1.\texttt{h0})$$

$$\sigma(\texttt{Inner.h}, \iota_2) \le \sum_{t_0<t} \kappa[n_3.\texttt{h1}, t_0] = \sigma(\texttt{Leaf.h1}, \iota_4) + \sigma(\texttt{Leaf.h1}, \iota_5) + \sigma(\texttt{Leaf.h1}, \iota_6) + \sigma(\texttt{Leaf.h1}, \iota_7), \qquad \text{(read for } n_3.\texttt{h1})$$

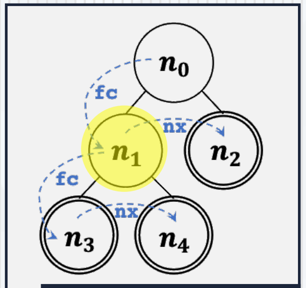
"n_1.h0 should have been scheduled somewhere before the current corresponding location"

  - Validity Constraints

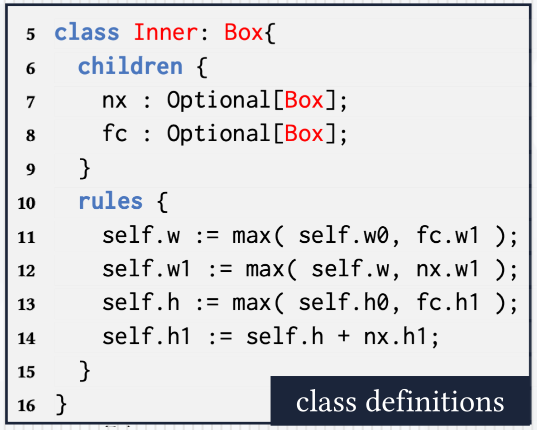
Constraints are not talking about t anymore, but about domain-specific relations now.

$$\forall \iota. \sum_a \sigma(a, \iota) \le 1,$$ - Every slot should be filled with at most one rule.

$$\forall a. \sum_\iota \sigma(a, \iota) = 1.$$ - Every rule should be used by only one slot.

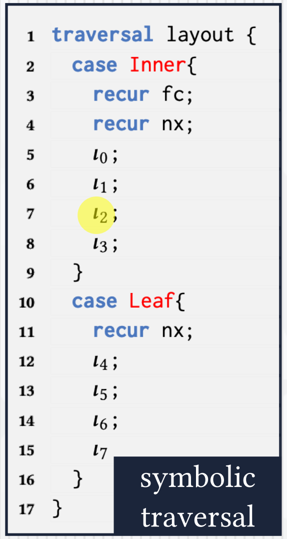


example tree



```
class Inner: Box{
  children {
    nx : Optional[Box];
    fc : Optional[Box];
  }
  rules {
    self.w := max( self.w0, fc.w1 );
    self.w1 := max( self.w, nx.w1 );
    self.h := max( self.h0, fc.h1 );
    self.h1 := self.h + nx.h1;
  }
}
```

class definitions

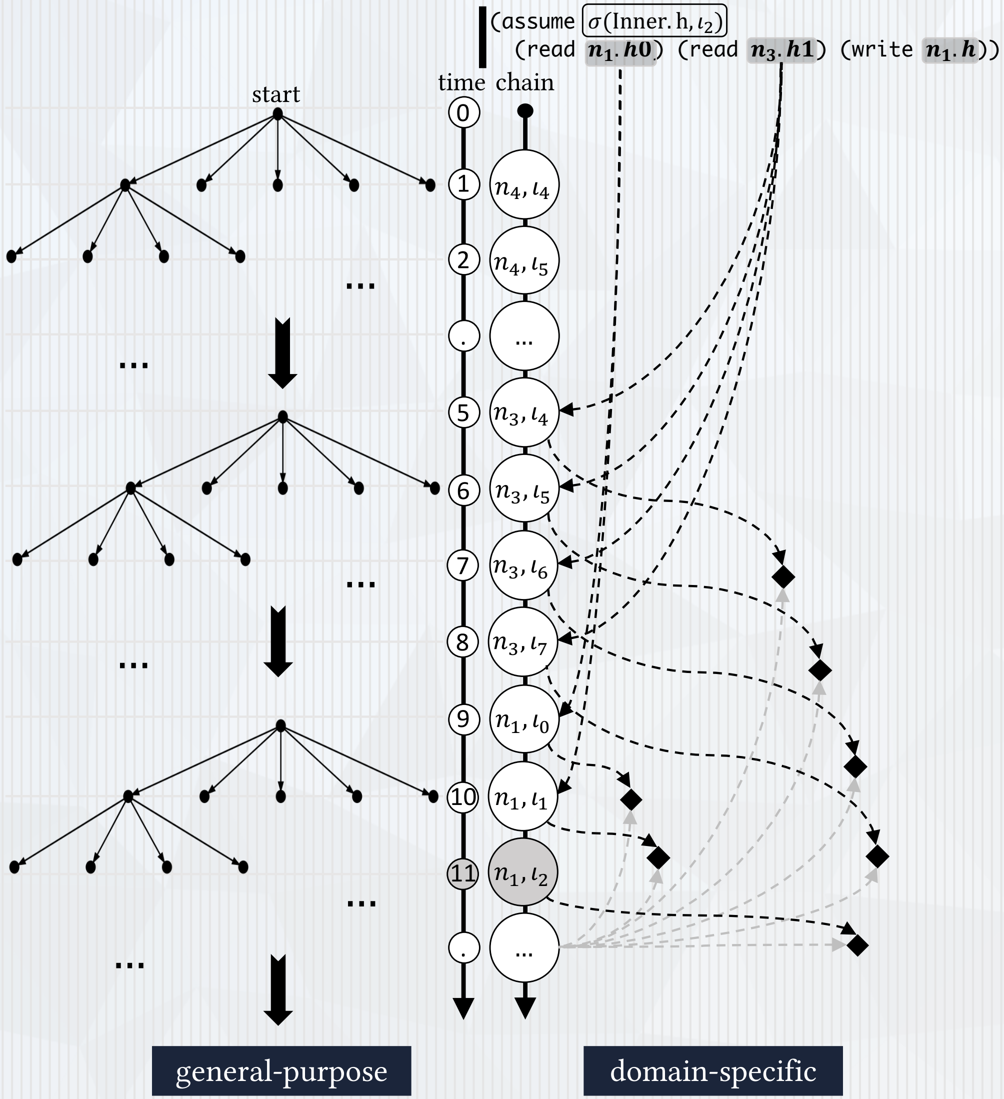


```
traversal layout {
  case Inner{
    recur fc;
    recur nx;
    ι0;
    ι1;
    ι2;
    ι3;
  }
  case Leaf{
    recur nx;
    ι4;
    ι5;
    ι6;
    ι7
  }
}
```

symbolic traversal

- Synthesis Using HECATE -

Complexity Analysis



```
(assume σ(Inner.h, ι2)
  (read n1.h0) (read n3.h1) (write n1.h))
```

* Please refer to the paper for more detailed analysis.

Evaluation

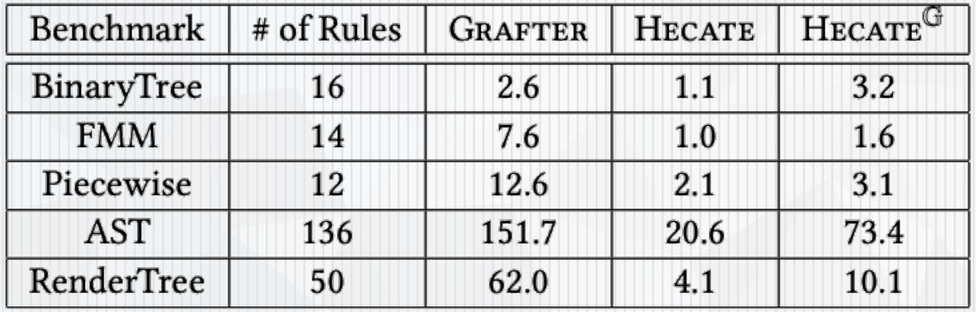
- Research Questions
    - **[Performance]** What is the performance of synthesized traversals, compared to those generated by state-of-the-art traversal synthesizers?
    - **[Expressiveness]** Is HECATE's tree language expressive enough? In particular, can it express prevailing tree traversal synthesis 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?

- Evaluation -

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$^{\mathbb{G}}$ (with general-purpose encoding). The table shows total synthesis time (synthesis + verification) in second.



Benchmark	# of Rules	GRAFTER	HECATE	HECATE$^{\mathbb{G}}$
BinaryTree	16	2.6	1.1	3.2
FMM	14	7.6	1.0	1.6
Piecewise	12	12.6	2.1	3.1
AST	136	151.7	20.6	73.4
RenderTree	50	62.0	4.1	10.1

---

[1] Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees. Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, Milind Kulkarni. PLDI 2019.

- Evaluation -

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 $^{\mathbb{L}}$: Sequential, Linked List
  - HECATE $^{\mathbb{V}}$: Sequential, Vector
  - HECATE $^{\mathbb{P}}$: Parallel, Vector

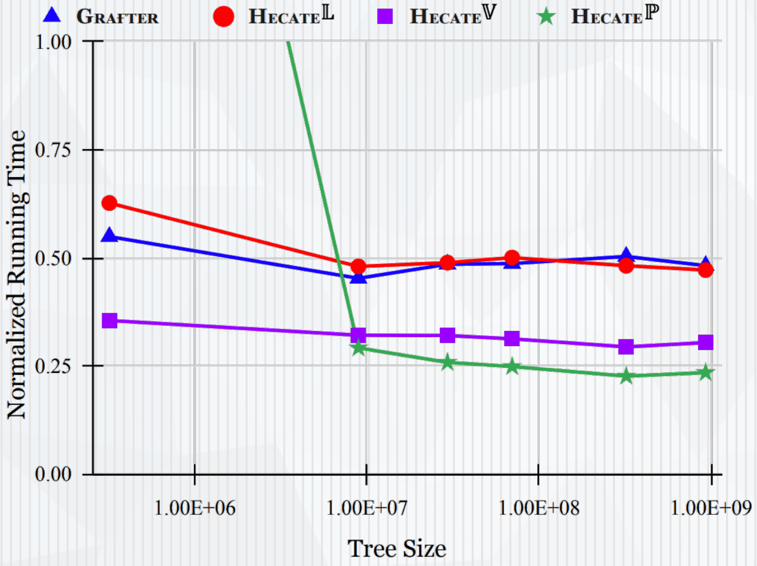


Figure 11: Running time of fused traversals compared to the unfused baseline.

With minimal efforts, Hecate can effectively explore traversals of different design choices.

- Evaluation -

Synthesizing Layout Engine in FTL[1]

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

Name	# of Rules
CSS-float	192
CSS-margin	178
CSS-full	244

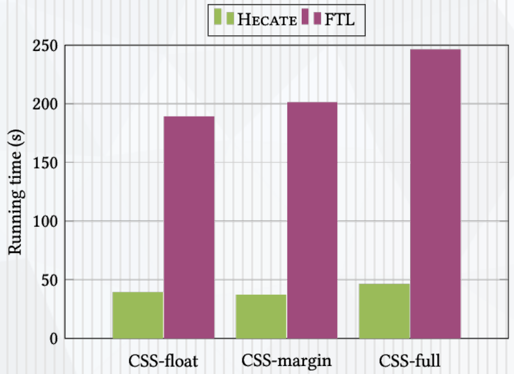


Figure 15: Comparison against FTL**: benchmark statistics (left) and results (right).**

---

[1] Parallel Schedule Synthesis for Attribute Grammars. Leo Meyerovich, Matthew Torok, Eric Atkinson, Rastislav Bodik. PPoPP 2013.

Conclusion

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

https://github.com/chyanju/Hecate

Thank you!