Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation

Yanju Chen
yanju@cs.ucsb.edu
University of California, Santa Barbara
USA

Yu Feng
yufeng@cs.ucsb.edu
University of California, Santa Barbara
USA

Junrui Liu
junrui@cs.ucsb.edu
University of California, Santa Barbara
USA

Rastislav Bodik
bodik@cs.washington.edu
University of Washington
USA

ABSTRACT

Efficient computation on tree data structures is important in compilers, numeric computations, and web browser layout engines. Efficiency is achieved by statically scheduling the computation into a small number of tree traversals and by performing the traversals in parallel when possible. Manual design of such traversals leads to bugs, as observed in web browsers. Automatic schedulers avoid these bugs but they currently cannot explore a space of legal traversals, which prevents exploring the trade-offs between parallelism and minimizing the number of traversals.

We describe Hecate, a synthesizer of tree traversals that can produce both serial and parallel traversals. A key feature is that the synthesizer is extensible by the programmer who can define a template for new kinds of traversals. HECATE is constructed as a solver-aided domain-specific language, meaning that the synthesizer is generated automatically by translating the tree traversal DSL to an SMT solver that synthesizes the traversals. We improve on the general-purpose solver-aided architecture with a scheduling-specific symbolic evaluation that maintains the engineering advantages solver-aided design but generates efficient ILP encoding that is much more efficient to solve than SMT constraints.

On the set of Grafter problems, HECATE synthesizes traversals that trade off traversal fusion to exploit parallelism. Additionally, HECATE allows defining a tree data structure with an arbitrary number of children. Together, parallelism and data structure improvements accelerate the computation 2× on a tree rendering problem. Finally, HECATE’s domain-specific symbolic compilation accelerates synthesis 3× compared to the general-purpose compilation to an SMT solver; when scheduling a CSS engine traversal, this ILP-based synthesis executes orders of magnitude faster.

CCS CONCEPTS

- Software and its engineering → Automatic programming.

KEYWORDS

symbolic compilation, program synthesis, tree traversal

ACM Reference Format:


1 INTRODUCTION

Traversal of tree structures is the foundation behind many applications: compilers leverage traversals of abstract syntax trees (ASTs) to analyze and optimize source codes. Layout engines in web browsers rely on traversals of render trees to determine the locations and appearances of HTML elements on web pages. Implementing tree traversals is a daunting task as it needs to strike a good balance between modularity and performance. On one hand, due to the complexity of modern layout engines, browser developers have to manually design scheduling strategies for rendering tree traversals in exchange for better performance. On the other hand, tree traversals in compilers are designed in a modular way, where mutually dependent traversals read and update attributes of ASTs [41]. This provides a great opportunity for automated scheduling of tree traversals. In particular, traversals that operate on the same node can be merged to reduce the overhead of node visiting and improve locality.

Even though manual scheduling of tree traversals offers fine-grained control for maximizing performance of a layout engine, the complexity of layout semantics (e.g., from W3C CSS standards) make it difficult to maintain the infrastructure and fix the notorious bugs. For instance, the Servo layout engine contains several bugs that have been open for over five years [14, 25] due to a mismatch between the intended semantics and the architecture chosen for its implementation1.

Automated scheduling of tree traversals aims to merge modular passes (or visitors) that operate on the same node of a tree. However, existing approaches are far from satisfactory. For instance, there are approaches that are specialized to certain types of tree traversals,

1A Servo developer remarked that “it took three weeks before I realize[d] the actual complexity of the problem”, which refers to the bug that is still open by the time of this submission; the Servo developers have resolved to delay fixing it until a complete rewrite of the layout engine is done [36].
such as TreeFuser [49] and Grafter [41]. But they leverage deterministic rewrite rules as well as automata-based representations that are complex to maintain. Synthesis-based tools like FTL [32] express tree computations using attribute grammars and leverage constraint solvers (i.e., Prolog) to find candidate solutions that satisfy the dependencies among tree computations. However, FTL requires domain experts for translating the layout semantics into constraints in Prolog, which is error-prone.

Motivated by these observations, we introduce HECATE, a synthesizer of tree traversals that can produce both serial and parallel traversals. In particular, HECATE provides a high-level tree language for defining templates for new kinds of tree traversals. The core synthesis engine of HECATE is built on top of a solver-aided framework [45], which lifts the execution of an interpreter for tree language programs into constraints that can be solved by off-the-shelf solvers. As a result, HECATE eliminates the enormous engineering efforts in FTL while preserving the efficiency and flexibility of exploring different design choices. To use HECATE to synthesize a concrete traversal, the developer only needs to specify a simple traversal template with holes yet to be filled with computation rules defined by the tree language. After that, HECATE completes the traversals using a counterexample-guided inductive synthesis (CEGIS) loop [43]: the synthesizer searches for a candidate traversal that works for the initial examples. The verifier then looks for a counterexample that fails for the traversal and invokes the synthesizer again to find a new candidate that is consistent with the counterexample. The process continues until the verifier cannot find additional counterexamples.

As we show later in the evaluation, direct interpretation of full semantics of a tree traversal will lead to difficult-to-solve constraints due to path explosions. To address this, HECATE employs a domain-specific symbolic compilation strategy, which maintains the usability of symbolic compilation, yet scales to problems orders of magnitude larger. The key insight is a semantic projection layer between the interpreter and the symbolic compilation engine that tailors the constraint generation procedure. Specifically, we introduce a trace language that disentangles complex dependencies from time domain to relational domain, where constraints can be equivalently expressed independent of time, thus clearing out path explosions while still ensuring the correctness of constraints. Under domain-specific symbolic compilation, the trace language generates integer linear programming (ILP) constraints that can be solved efficiently.

We implement HECATE and compare it against GRAFTER and FTL, showing that our tool is expressive, efficient, and flexible. On the set of GRAFTER benchmarks, HECATE synthesizes traversals that trade-off traversal fusion to exploit parallelism. Additionally, HECATE allows defining a tree data structure with an arbitrary number of children. Together, parallelism and data structure improvements accelerate the computation 2× on a tree rendering problem. Finally, HECATE’s domain-specific symbolic compilation accelerates synthesis 3× compared to the general-purpose compilation to an SMT solver; when scheduling a CSS engine traversal, this ILP-based synthesis executes orders of magnitude faster.

To summarize, we make the following contributions:

- We propose a CEGIS framework for tree traversals.

---

**Figure 1**: Pseudo-code class definitions (unfused and fused versions) for rendering tree example.

- We propose a domain-specific trace language that disentangles complex dependencies from time domain to relational domain, which results in easy-to-solve constraints.
- We implement the proposed ideas in a tool called HECATE and demonstrate that it achieves 3× speed-up on GRAFTER benchmarks compared to general-purpose symbolic compilation.

## 2 OVERVIEW

In this section, we illustrate how HECATE works using a running example.

**A Rendering Tree Example.** Layout engines in modern web browsers utilize the box model in rendering procedures. This example demonstrates simplified behaviors of two types of boxes: Inner boxes and Leaf boxes where the former can hold child boxes and the latter can’t. Figure 1(a) shows the realization of the boxes. In the example:

```cpp
1 class Box {  // same as unfused
2 public:
3     int w0, h0; // input (default)  
4     int w1, h1; // helper
5     int w, h;   // output (final)
6 }
7
t class Inner: public Box {  // same as unfused
8 public:
9     Box* fc; // first child
10     Box* nx; // next sibling
11 }
12
13 void Inner::calcWidth() {
14     fc->calcWidth();
15     nx->calcWidth();
16     w = max( w, fc->w );
17     w = max( w, nx->w );
18     w = max( w, fc->w1 );
19     void Inner::calcHeight() {
20         h1 = h + nx->h1;
21         h = max( h, fc->h );
22         h = max( h, fc->h1 );
23         h = h + nx->h1;
24     }
25 }
26
t class Leaf: public Box {  // same as unfused
27 public:
28     Box* mx; // next sibling
29 }
30
31 void Leaf::calcWidth() {
32     w = w0;
33     w = max( w, mx->w );
34     void Leaf::calcHeight() {
35         h = h0;
36         h = h + mx->h1;
37     }
38 }
39
(a) unfused version  (b) fused version
```

---

1031
Figure 2: A motivating example tree with different properties shown (a): access paths; (b)-(e): dependencies between attributes of nodes.

- Line 16: The final width of an Inner box (denoted by \( w \)) is decided by the larger one between its default width (denote by \( w_0 \)) and the maximum width of its children (denoted by \( fc->w_1 \)).
- Line 22: The final height of an Inner box (denoted by \( h \)) is decided by the larger one between its default height (denoted by \( h_0 \)) and the summed height of its children (denoted by \( fc->h_1 \)).
- Line 33, 38: For a Leaf node, its final width and height are decided by its provided default values (denoted by \( w_0 \) and \( h_0 \), respectively).
- Specifically, the helper variable \( w_1 \) records the maximum final width among a node and all its siblings accessible by \( nx \), and the helper variable \( h_1 \) records the summed final height of a node and all its siblings accessible by \( nx \). In other words, a node can refer to \( w_1 \) of its first child as the maximum final width among all its children, and refer to \( h_1 \) of its first child as the summed final height of all its children.

The two methods calcWidth and calcHeight demonstrate the computations for final width and height, respectively.

Typically, rendering a box requires a proper tree traversal to compute all the attribute values. Besides, in real-world use case scenarios, a finer-grained scheduling of computations is usually required as an optimization. As shown in Figure 1(b), a more efficient method fusedCalc is synthesized to perform width and height computations at the same time. Because attribute values may depend upon one another, solving for a correct order of attribute evaluations becomes challenging.

How Hecate Works. Now with Hecate, the user starts by providing: 1) a symbolic traversal as shown in Figure 4(a) with slots \( i_l \) in which we can schedule at most one computation rule from the grammar, 2) an example tree shown in Figure 2(a), and 3) the corresponding semantics written in Hecate’s visitor language (shown in Figure 3). Hecate then automatically synthesizes a concrete traversal by filling the slots with computation rules while respecting all the read-write dependencies, as shown in Figure 4(b).

More specifically, executing the traversal over the example tree on an Inner node (e.g., \( n_1 \)) first recursively computes the attributes of its child nodes (line 3 in Figure 4(b), i.e., \( n_3 \) and \( n_4 \) in Figure 2(a)) and its next sibling (line 4, i.e., \( n_2 \)), and then attributes of itself (line 5-8). For a Leaf node, the computation is similar but without the recursion on the children.

Figure 5 illustrates the overview of Hecate, which instantiates a CEGIS loop with two phases: 1) Given the specification that contains class definitions as attribute grammar, initial trees, and a symbolic traversal, the synthesizer searches for a concrete traversal, and sends it to the verifier; 2) The verifier checks the correctness of the concrete traversal over all possible example trees (up to depth \( k \)).
and returns a counterexample tree that fails the current traversal. Then the synthesizer finds a new candidate that is consistent with the newly added counterexample tree. This process continues until the verifier cannot find additional counterexamples to refute the current traversal, which will be returned as the correct solution.

To synthesize the desired traversal, Hecate leverages Rosette [45] to lift an interpreter for tree traversal into a synthesizer through symbolic compilation. In particular, given a symbolic traversal with slots, symbolic compilation expands each slot into all possible choices of computation rules, and executes the traversal to generate dependency constraints under different choices. A concrete traversal is then obtained by solving the constraints using off-the-shelf SMT solvers [18, 24].

While a faithful interpretation of the semantics of the traversal usually causes path explosions, as choices of rules depend on choices made on preceding execution steps, it generates complex SMT constraints that are hard to solve, thus damaging the performance. Our solution is to insert a domain-specific layer below the interpreter and above the symbolic engine (i.e., Rosette), which exposes a domain-specific tree visitor grammar that still allows us to write rules based on attribute grammar for nodes of types Inner and Leaf, which share the same set of attributes declared by the interface box. Specifically:

- each Inner node has two children \( nx \) and \( fc \) that point to its next sibling and first child respectively;
- a Leaf node does not have children.

Rules for computing the attributes vary across different classes. e.g.,

- attributes \( w \) and \( b \) of a Leaf node only depend on attributes from itself;
- attribute \( w \) from an Inner node depends on the default width of itself (i.e., \( self.w0 \)) and the maximum width of its children (i.e., \( fc.w1 \)),
- attribute \( h \) from an Inner node depends on the default height of itself (i.e., \( self.h0 \)) and the summed height of its children (i.e., \( fc.h1 \)).

3.2 Language for Tree Traversals

To formally define a tree traversal, we first introduce domains for different notations.

**Syntax.** Figure 7 summarizes the language \( \mathcal{L}_t \) for expressing tree traversals. In particular, A traversal (traversal) is declared with a list of (case) blocks. Each (case) block matches a node type and contains statements of the following forms:

\[
\begin{align*}
\text{(interface)} & ::= \text{interface } (id) \{ (\text{tstmt})^* \} \\
\text{(class)} & ::= \text{class } (\text{tup}) \{ (\text{children}) \{ (\text{rules}) \} \} \\
\text{(children)} & ::= \text{children } (\{ (\text{tup})^* \} \} \\
\text{(rules)} & ::= \text{rules } \{ (\text{cstmt})^* \} \\
\text{(tup)} & ::= \{ \text{(id)}: (\text{tup})^* \} \\
\text{(sel)} & ::= \text{sel} (\text{id}) ? (\text{id})^* \\
\text{(expr)} & ::= \{ \text{const} \} \{ \text{sel} \} \{ f (\text{expr})^* \} \\
\text{(eval)} & ::= \text{eval } (\text{cstmt}) \\
\text{(tstmt)} & ::= \{ \text{recur} \} \{ \text{iterate} \} \{ \text{eval} \} \\
\{ \text{id} \} & \in \text{identifiers } (\text{node}) \in \text{nodes} \\
\end{align*}
\]

As a result, \( \mathcal{L}_a \) is specialized for modeling the behaviors of reading and writing of attributes of the current node and its children, which essentially describes the dependency relations between attributes of nodes.

**Example 3.1.** The code snippet from Figure 3 declares the attribute grammar for nodes of types Inner and Leaf, which share the same set of attributes declared by the interface box. Specifically:

- each Inner node has two children \( nx \) and \( fc \) that point to its next sibling and first child respectively;
- a Leaf node does not have children.

Rules for computing the attributes vary across different classes. e.g.,

- attributes \( w \) and \( b \) of a Leaf node only depend on attributes from itself;
- attribute \( w \) from an Inner node depends on the default width of itself (i.e., \( self.w0 \)) and the maximum width of its children (i.e., \( fc.w1 \)),
- attribute \( h \) from an Inner node depends on the default height of itself (i.e., \( self.h0 \)) and the summed height of its children (i.e., \( fc.h1 \)).
yields the following total order of locations:

Figure 2

Figure 4

Figure 3

for a

Figure 4

15

Figure 2

Figure 4

Figure 8

36

and the

where in every time step \( t \in T \) enforces the order of different rules. An attribute grammar provides a list of attributes \( A \) that uniquely determine their corresponding computation rules. Each traversal provides a list of slots (i.e., \( S \)) for holding computation rules from the attribute grammar. Each node \( n \) has a set of locations (i.e., \( L \)) that refer to its corresponding attributes in runtime.

Definition 3.2. Traversal. Given a tree, a traversal defines a total order relation \( < \) over the set of all locations of the tree.

Example 3.3. A concrete post-order traversal (i.e., Figure 4(b)) on the tree in Figure 2 yields the following total order of locations:

\[
\begin{align*}
&n_4.w < n_4.h < n_4.w_1 < n_4.h_1 < n_3.w < n_3.h < n_3.w_1 < n_3.h_1
\end{align*}
\]

where in every time step \( t \in T \) one location is visited. Note that different traversals may induce different orders.

A traversal is symbolic if it contains at least one slot \( i \) and is concrete otherwise.

Example 3.4. Figure 4(a) declares a symbolic and post-order traversal over nodes of types Inner and Leaf. Figure 4(b) is an instantiation of the previous symbolic traversal. In particular, the traversal first computes the attributes for the leaves of type Leaf, and then the attributes of the nodes of type Inner.

Note that the concrete post-order traversal preserves the read-write dependencies induced by the attribute grammar in Figure 3. On the contrary, a pre-order traversal would not be valid since it violates the read-write dependencies imposed by attributes including \( w \) and \( h \) of a Inner node. We then define the tree traversal synthesis problem as follows:

Definition 3.5. Tree Traversal Synthesis. Given an attribute grammar \( L_a \) and a symbolic traversal \( P_i \) with holes, a tree traversal synthesis problem is to induce a concrete traversal \( P_t \) by completing the holes \( i \) in \( P_i \) with computation rules in \( L_a \), such that for arbitrary instantiated trees from \( L_a \): 1) all attributes are computed, and 2) all read-write dependencies are preserved.

Example 3.6. Given the attribute grammar \( L_a \) in Figure 3 and the symbolic traversal in Figure 4(a), Hecate synthesizes the concrete traversal in Figure 4(b). Given arbitrary tree derived from Figure 2, the synthesized traversal computes all attributes of the tree exactly once and respects all read-write dependencies.

As we show later, to bypass the challenge of complex SMT constraints that are generated by a faithful interpretation of a traversal’s semantics, a scalable approach to solve the placement and resource allocation problems is to use ILP to map the computation rules to the available slots in the traversal [15, 36].

Definition 3.7. 0-1 Integer Linear Programming. Given coefficients \( a, b \) and \( c \), the 0-1 ILP problem is to solve for \( x \) as follows:

\[
\min \sum_i c_i x_i \quad s.t. \quad a_i x_j + b_i \leq 1,
\]

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

We obtain a set of ILP constraints that is easy to solve by ILP solvers from domain-specific symbolic compilation via program written in trace language, which is deferred to Section 5 for a detailed discussion.

4 TRAVERSE SYNTHESIS

In this section, we formally introduce our synthesis framework for tree traversals that is based on counterexample-guided inductive synthesis (CEGIS). In what follows, we first give a high-level overview of the synthesis framework, then we show how to reduce the synthesis problem to a general-purpose symbolic compilation problem based on Rosette. Finally, we briefly discuss its limitation.

4.1 System Overview

As shown in Figure 5, Hecate takes as inputs an attribute grammar \( L_a \), a symbolic traversal with unknown slots in \( L_t \), and an initial tree for validating the correctness of the traversal. The output of Hecate is a concrete traversal that respects all the read-write dependencies imposed by the attribute grammar.

Synthesis. Figure 8(a) sketches the core synthesis engine that is built on top of Rosette [45], a hybrid symbolic compiler that combines symbolic execution and bounded model checking to compute compact constraints. In particular, the general-purpose interpreter interpret for tree traversals takes as inputs a grammar grammar, a traversal traversal, and a concrete tree tree. Following the total order induced by traversal, the outermost loop of the interpreter recursively visits each node in tree and its corresponding locations loc(line 2). When evaluating a symbolic choice for a slot, symbolic evaluation considers each alternative concrete rule (line 3-4), generates the constraints stating that the dependencies are ready and the target has not been computed (line 6-7), sets the target attribute as ready, and updates the program state (line 8). The interpreter behaves as a regular emulator when it runs with concrete traversals and trees. For instance, running it with the post-order traversal in Figure 4(b) and the example tree in Figure 2 will pass all assertions and terminate normally. What is more interesting is that given a symbolic traversal traversal with slots yet to be filled, Rosette runs the interpreter with traversal and a concrete tree tree under symbolic evaluation; this encodes all possible concrete traversals that preserve the dependencies in tree, effectively lifting the interpreter to be a synthesizer.
Figure 8: Code snippets of general-purpose interpreter and domain-specific interpreter.

Verification. To ensure that the synthesized traversal traversal is not only correct on the initial example tree but also on all possible trees, we again leverage Rosette to build our verifier. In particular, the core of the verifier is another interpreter that is almost identical to the one in Figure 8(a). Now the inputs of the interpreter include a concrete traversal traversal that needs to be verified, as well as a symbolic tree traversal that encodes the space of all possible concrete examples up to depth k. In that case, symbolically evaluating traversal traversal yields a formula stating that traversal is correct on all instantiations of traversal. If the formula is satisfiable, the verifier then returns a counterexample to the synthesizer that will look for another candidate. Similar to prior work in Neo [20] and Bonsai [15], our symbolic tree traversal is encoded as a bounded m-ary tree derived from the attribute grammar. We omit the details since it is not the main contribution.

We call the interpreter in Figure 8(a) general-purpose symbolic compilation. By nature, its encoding (general-purpose encoding) establishes read-write dependencies across different execution time steps. While general-purpose encoding is fairly intuitive and straightforward to implement, it may lead to complex constraints that are difficult to solve.

4.2 General-Purpose Symbolic Compilation

In this section, we elaborate on the details behind general-purpose symbolic compilation. Using the domains introduced in Section 3.2, we first define a set of relational operators to formalize the general-purpose symbolic compilation:

- **Assignment.** The assignment operator maps an attribute a ∈ A and a slot i ∈ I to a boolean variable in B, i.e., σ : A × I → B. Since each attribute a is uniquely computed by a computation rule from the attribute grammar, predicate σ(a, i) evaluates to true iff the rule for computing attribute a is scheduled at slot i.

- **Dereference.** Recall that in Figure 6, each computation rule for an attribute is composed of multiple access paths sel appearing in a statement. During tree traversal, the dereference operator ζ(n, sel) returns the concrete location I (i.e., N × A) that access path sel of node n points to.

- **Ready Bit.** The ready bit operator maps a node n ∈ N, an attribute a ∈ A, and a time step t ∈ T to a boolean variable in B, i.e., δ : N × A × T → B. Here, predicate δ(n, a, t) returns true if the attribute a of node n is already computed (i.e., ready for being read by other computation rules) before time step t.

- **Symbolic Choice.** Recall that a symbolic traversal contains at least one slot i that represents at most one attribute computation yet to be scheduled. To handle this case, we introduce a symbolic choice operator choose for non-deterministically choosing an attribute (to compute) from a list of available attributes A. For instance, (choose [Inner.h, Inner.w, none]) returns one of the attributes from the list.

Now, during the general-purpose symbolic compilation in Figure 8(a), the interpreter executes statements in traversal traversal and dynamically inserts assertions (line 6-7) to state the correctness of every single computation rule. In particular, the correctness enforces read-write dependency using the ready bit operator δ. Therefore, for a statement with chosen attribute, e.g., eval _self.h at slot i2 of node n1, Rosette compiles it into the following constraints:

δ(ζ(n1, _self.h0), t) ∧ δ(ζ(n1, fc.h1), t) ∧ ¬δ(ζ(n1, _self.h), t)

where t is the current time step and rule _self.h := max(_self.h0, fc.h1) (line 13 in Figure 3) is used to compute attribute _self.h for node n1 of type Inner. Here, the above constraints state two properties about the read-write dependencies: 1) attributes of nodes (i.e., _self.h0 for n1 and fc.h1 for n3) should be ready before they are read, and 2) the attribute of a node (i.e., _self.h for n1) should not be ready until it is written.

The interpreter starts by executing a slot statement i in the symbolic traversal. In that case, each i is dynamically replaced by a statement that non-deterministically chooses an available attribute a_i to schedule 5: eval (choose [a_1, ..., a_n]). After that, Rosette symbolically evaluates the above statement and compiles it into a formula stating all possible cases where each case is guarded by the conjunction of assignment operators σ that represent the cumulative choices so far.

**Example 4.1.** For instance, at time step t, when the interpreter executes slot i2, i.e. line 7 in Figure 4(a), on Inner node n1 in Figure 10(a), it can choose one of the five options from none, Inner.w1, Inner.w, Inner.h1 and Inner.h according to its attribute grammar in Figure 10(b):

\[
\text{eval (choose S)}
\]

where

S : [none, Inner.w1, Inner.w, Inner.h1, Inner.h]
which is further transformed into the following formula:

\[
\begin{align*}
\sigma & (\text{none, } t_2) \implies \text{true} \\
\vee \ (\sigma(\text{Inner}, w, t_2) & \implies \delta(z(n_1, \text{self.w}, t) \land \delta(z(n_1, n.x.w), t) \\
& \land \neg \delta(z(n_1, \text{self.w}), t) ) \\
\vee \ (\sigma(\text{Inner}, w, t_2) & \implies \delta(z(n_1, \text{self.w}, t) \land \delta(z(n_1, n.x.w), t) \\
& \land \neg \delta(z(n_1, \text{self.w}), t) ) \\
\vee \ (\sigma(\text{Inner}, h, t_2) & \implies \delta(z(n_1, \text{self.h}, t) \land \delta(z(n_1, n.x.h), t) \\
& \land \neg \delta(z(n_1, \text{self.h}), t) ) \\
\vee \ (\sigma(\text{Inner}, h, t_2) & \implies \delta(z(n_1, \text{self.h}, t) \land \delta(z(n_1, n.x.h), t) \\
& \land \neg \delta(z(n_1, \text{self.h}), t) )
\end{align*}
\]

where \( \sigma(\text{Inner}, h, t_2) \) evaluates to \text{true} if we decide to compute attribute \text{Inner} \cdot h at slot \( t_2 \).

- \text{Inner} \cdot h \text{ and } \text{fc} \cdot h \text{ should be ready before time step } t, \text{ and}
- \text{self} \cdot h \text{ should not be scheduled before time step } t.

In addition to correctness constraints, we also enforce auxiliary constraints to induce valid traversals. For instance, the following constraint requires every slot be filled with at most one rule:

\[
\forall i \left( \bigvee_{a_0 \in a_0} \neg \sigma(a, i) \land \sigma(a_0, i) \right) \land \bigvee_{a \in a} \neg \sigma(a, i)
\]

And the following requires every rule be used by only one slot:

\[
\forall a_0 \left( \bigvee_{i \in a_0} \neg \sigma(a, i) \land \sigma(a_0, i) \right)
\]

Performance Analysis. While it is intuitive and straightforward to build a tree traversal synthesizer using general-purpose encoding, it suffers from path explosion by faithfully following the execution of a traversal, even with the effective state-merging and pruning strategy from Rosette. Figure 9 shows how the number of symbolic state grows as time goes by. Consider a tree of \( n \) nodes with an average of \( k \) slots per node, the general-purpose symbolic compilation will generate constraints based on a chain of length \( n \cdot k \) with dependencies between choices made in a recursive way, i.e. nested choose operations. Assuming that every slot has a candidate set of \( q \) rules to fill in on average, the total number of symbolic states after compilation can be \( u \) to \( q^{n \cdot k} \). As shown in our evaluations, the general-purpose symbolic compilation creates constraints that take a long time to solve.

5 DOMAIN-SPECIFIC SYMBOLIC COMPILATION

As discussed in Section 4, a general-purpose symbolic compilation faithfully follows the execution of a traversal across different execution time steps, which leads to constraints that are hard to solve. To mitigate this problem, we propose a domain-specific trace language, which projects the complex dependencies from time domain to relational domain and yields easy-to-solve constraints. In what follows, we first introduce the trace language \( L_r \), and then show how to obtain ILP constraints via domain-specific symbolic compilation.

5.1 A Trace Language for Tree Traversals

As shown in Figure 8(b), thanks to Rosette, the skeleton of domain-specific interpreter for synthesizing tree traversals can be obtained with a minor modification over the general-purpose version: upon executing a statement in traversal traversal, instead of directly adding its corresponding assertions, we first translate the statement into another program in trace language \( L_r \), and then leverage Rosette to lift the execution of the new tree program to constraints that can be modeled as an integer linear programming (ILP) problem.
The syntax and semantics of \( L_r \) are summarized in Table 1. Intuitively, \( L_r \) understands dependency relations carried through attributes on nodes with fully abstract contents. In particular:

- \((\text{read } n.a)\) logs the read action of attribute \( a \) on node \( n \);
- \((\text{write } n.a)\) logs a write to \( n.a \);
- \((\text{choose } [a_1, ..., a_n])\) non-deterministically selects an attribute \( a_i \) to compute.

For the sake of simplicity, we use the built-in assume function in Rosette to explicitly enumerate each option of symbolic choices under different assumptions.

During the execution of the domain-specific interpreter (Figure 8(b)) at line 6, Hecate invokes a syntax-directed transpilation procedure to generate the corresponding trace program, which captures the dependency relations to ensure the correctness of tree traversals, and provides succinct statements that eventually lead to efficient constraints (Section 5.2).

### Example 5.1
Following Example 4.1, suppose we are in the symbolic traversal (i.e., Figure 4(a)) at slot \( t_3 \) and the current node is \( n_3 \) in Figure 10(a). And the synthesizer decides to select a rule \( \text{self}, h := \text{max}( \text{self}, h_0, f_c, h_1) \) from the visitor program in Figure 10(b) to compute the \( \text{Inner}.h \) attribute in slot \( t_2 \), then a syntax-directed transpilation procedure is invoked to generate the following trace program:

\[
(\text{assume } \sigma(\text{Inner}.h, t_2) \quad (\text{read } n_3.h_0) \quad (\text{read } n_3.h_1) \quad (\text{write } n_3.h)).
\]

Semantically the above trace program states that in order to compute \( \text{Inner}.h \) at slot \( t_3 \), two attributes (i.e., \( n_3.h_0 \) and \( n_3.h_1 \)) should first be read and another attribute (i.e., \( n_3.h \)) should then be written. The trace program records read-write dependencies in a more compact way without introducing time steps.

### 5.2 Symbolic Compilation of Trace Program

Even if we obtain a trace program \( P_r \) using the procedure discussed in Section 5.1, the trace program itself does not mitigate the path explosion problem because similar to the general-purpose encoding, the symbolic choice statements in the trace program still encode path conditions at each time step. To address this challenge, we discuss how our domain-specific compilation further projects the executions of the trace program into compact constraints that can be solved by efficient ILP solvers [24].

### Dependency Constraints
We first introduce a dependency operator \( \kappa \) that takes as inputs a location \( l \in \mathbb{N} \times A \), a time step \( t \in T \), and returns a boolean variable \( \sigma \) that specifies all possible slots in which the attribute \( a \) of location \( l \) was computed. In other words, \( \kappa \) encodes the dependency between locations (i.e., \text{read}) and their corresponding attributes (i.e., \text{write}).

We then illustrate the relationship between the trace program and the dependency operator \( \kappa \). In particular, for a write instruction \((\text{write } n.a)\) at time step \( t \) guarded by \( \sigma(a,i) \), the dependency operator \( \kappa \) gets updated by:

\[
\kappa(n.a,t) \leftarrow \sigma(a,i),
\]

where attribute \( a \) is written at time step \( t \) if \( \sigma(a,i) \) evaluates to 1 (i.e., true).

For a read instruction \((\text{read } n.a)\) at time step \( t \) guarded by \( \sigma(a,i) \), if \( \sigma(a,i) \) evaluates to true, then it implies that attribute \( a \) must be written somewhere before time step \( t \). Formally speaking, we have constraint:

\[
\sigma(a,i) \iff (\exists t_0, (t_0 < t) \land \kappa(n.a,t_0)),
\]

which can be easily translated into its equivalent ILP constraint \(^6\):

\[
\sigma(a,i) \leq \sum_{t_0 < t} \kappa(n.a,t_0). \quad \text{(read constraint)}
\]

### Example 5.2
Following Example 4.1 but in domain-specific symbolic compilation, as shown in Figure 9, suppose we want to schedule \( \text{self}, h := \text{max}( \text{self}, h_0, f_c, h_1) \) at slot \( t_2 \) of node \( n_1 \), which corresponds to the following trace program:

\[
(\text{assume } \sigma(\text{Inner}.h, t_2) \quad (\text{read } n_1.h_0) \quad (\text{read } n_1.h_1) \quad (\text{write } n_1.h)).
\]

The domain-specific encoding compiles the above trace program into the following ILP constraints:

\[
\begin{align*}
\sigma(\text{Inner}.h, t_2) &\leq \sum_{t_0 < t} \kappa(n_1.h_0, t_0) \\
&= \sigma(\text{Inner}.h_0, t_0) + \sigma(\text{Inner}.h_0, t_1), \quad \text{(read for } n_1.h_0) \\
\sigma(\text{Inner}.h, t_2) &\leq \sum_{t_0 < t} \kappa(n_1.h_1, t_0) \\
&= \sigma(\text{Leaf}.h_1, t_0) + \sigma(\text{Leaf}.h_1, t_1) + \sigma(\text{Leaf}.h_1, t_2), \quad \text{(read for } n_1.h_1)
\end{align*}
\]

where \( t \) corresponds to the time step when visiting \( t_2 \) of node \( n_1 \). According to Definition 3.2, since a traversal defines a total order relation over all locations of a tree, we can map the location that is currently being evaluated to a certain time step \( t \) and generate constraints that require all the dependencies of this location are ready before time step \( t \). This is done by \( \kappa \) in the example. Then, we again utilize the mapping to cancel the time step variables in the constraints by mapping them back to potential locations, thus resulting in a more compact constraint system.

### Validity Constraints
Similar to the general-purpose encoding in Section 4.2, we also impose extra constraints to ensure the validity of the traversals.

- For every slot \( i \), at most one rule can be filled in:

\[
\forall i. \sum_a \sigma(a,i) \leq 1, \quad \text{(slot constraint)}
\]

- Every rule \( a \) is used by exactly one slot \( i \):

\[
\forall a. \sum_i \sigma(a,i) = 1. \quad \text{(rule constraint)}
\]

\(^6\)Interchangeably, we use the same domain notation B to denote the boolean domain and \( 0,1 \) integer domain for general-purpose and domain-specific symbolic encodings, respectively.
Performance Analysis. To understand why the domain-specific compilation generates better constraints than the general-purpose version, we use Figure 9 to show a comparison between two strategies. Here we use the total number of symbolic states (i.e., the input space of all relational operators that introduce symbolic states) to approximate the complexity of constraints. Both the assignment operator $\sigma : A \times I \rightarrow B$ and the readiness operator $\delta : N \times A \times T \rightarrow B$ introduce symbolic states. Even though the size of the symbolic states generated by the readiness operator can grow as the size of the tree $n$, the number of attributes $q$, and the number of slots $k$ increase, it’s still bounded by a polynomial growth. In particular, the domain-specific encoding generates a maximum of $(1+n^3) \cdot q \cdot k$ symbolic states, which is more compact and less complex than the exponential number generated by general-purpose encoding.

6 EVALUATION

In this section, we describe the results of the experimental evaluation, which is designed to answer the following key research questions:

1. (Expressiveness) Is Hecate’s tree (visitor/traversal/trace) language expressive enough? In particular, can it express prevailing tree traversal synthesis problems and solve them?
2. (Performance) What is the performance of synthesized traversals, compared to those generated by state-of-the-art traversal synthesizers?
3. (Flexibility) Can Hecate be extended to explore traversals of different design choices?
4. (Efficiency) What is the benefit of the domain-specific encoding compared to general-purpose encoding?

For all experiments, Hecate requires user-provided attribute grammar, a symbolic traversal and an initial example tree as input, and outputs a concrete traversal in tree traversal language $L_t$.

6.1 Comparison against Grafter

We first compare Hecate against Grafter [41], the state-of-the-art tree traversal synthesizer based on static dependence analysis. In particular, Grafter builds access automata that summarises dependency relations for tree visitors, and synthesizes tree traversals using a deterministic algorithm. We adapt the original benchmark set from Grafter, which contains five representative tree traversal synthesis problems from real-world applications. Since Grafter benchmarks are written in C++, we also implement a code generator for converting concrete traversals synthesized by Hecate into corresponding C++ versions through syntax-directed translation. To study the benefit of domain-specific encoding discussed in Section 5, we also implement general-purpose encoding discussed in Section 4.2, which we denote as Hecate$^{G}$.

Efficiency and Expressiveness. Table 2 shows the results of the comparison. In particular, Hecate supports all 5 benchmarks from Grafter and successfully synthesizes the correct solutions (i.e., traversals that are semantically equivalent to the ones generated by Grafter) within 5.9 seconds on average. Specifically, Hecate yields an averaged speed-up of 3.1x compared to Hecate$^{G}$ and 8.0x compared to Grafter. The evaluation shows that Hecate’s tree language is expressive to support a variety of tree traversal applications. Furthermore, the comparison between Hecate and Hecate$^{G}$ also demonstrates the benefits of domain-specific encoding.

Performance. To evaluate the performance of the synthesized traversals, we directly adopt the workload from Grafter. Since our symbolic traversals are written in a way to “fuse” tree visitors whenever possible, like Grafter, the performance of our synthesized traversals are almost identical to the ones generated by Grafter. However, unlike Grafter that uses a deterministic algorithm for generating one unique solution for each benchmark, the tree language enables Hecate to flexibly explore various traversals of different design choices, some of which lead to dramatic performance speed-up. In what follows, we elaborate on the details using a case study from one of Grafter’s benchmarks: RenderTree.

Usability. To further minimize user effort, we implement a variant Hecate$^{A}$ that incorporates an auto-tuner that can automatically search for useful symbolic traversals during synthesis. In particular, the user only has to provide attribute grammar, and Hecate$^{A}$ will construct the example trees and initiate an outer loop that searches for a symbolic traversal that ensures correctness of its corresponding synthesized concrete traversal. Our experimental results indicate Hecate$^{A}$ can solve four Grafter benchmarks as fast as Hecate; for the AST benchmark with complex symbolic traversals, it takes Hecate$^{A}$ more than 30 mins to find a solution. We show that it is possible to get rid of more manual inputs for Hecate using a simple auto-tuner.

6.2 Case Study: RenderTree

In the RenderTree benchmark, a document tree consists of a list of pages containing nested horizontal and vertical containers with concrete elements as leaf nodes (e.g., text boxes, images, and itemized lists). A total of five rendering passes compute various visual attributes: (1) resolving flexible widths, (2) resolving relative widths, (3) computing heights, (4) propagating font styles, and (5) finalizing positions of elements. Each pass potentially depends on the attributes computed by previous passes.

Unlike Grafter, which only generates one unique traversal that fuses tree visitors whenever possible, Hecate offers a number of design choices. For instance, tree nodes frequently visit their children, which can be modeled using either linked lists or vectors. Moreover, when the children have no dependencies between themselves, the user may parallelize the computations using the

Table 2: Comparison between Grafter, Hecate and Hecate$^{G}$

<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>156</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>

7See Appendix A for a detailed case study for another benchmark: AST.
Figure 12: Class definitions in Hecate for rendering tree example, optimized with vector data structure. Only key refactorings are listed.

```cpp
1 class Inner: Box{
2     children {
3         cs: [Box]; // vector of 0 or more elements
4     }
5     rules {
6         self.w := fold( max, self.w0, cs.w );
7         self.h1 := fold( +, 0, cs.h );
8         self.h := max( self.h0, self.h1 );
9     }
10 }
```

(parallel) construct in the symbolic traversal. Then, Hecate can verify the absence of inter-dependency between children and generate a parallel scheduling over the list of children. Here, we evaluate the performance of the following variants: 1). HecateL: sequential linked-list-based traversal 2). HecateV: sequential vector-based traversal 3). HecateP: parallel vector-based traversal. Figure 11 summarizes the comparison with Grafter. Here, each line corresponds to one of the variants. The x-axis shows the tree size and the workload is directly adopted from the Grafter paper. The y-axis shows normalized running time over the unfused baseline, averaged over 10 trials.

Linked-List-Based Traversal. Due to limitation of Grafter’s static analysis, it only supports linked list for modeling variable-length arrays of children. The HecateL variant uses the same linked list data structure, and is able to synthesize a schedule that is semantically equivalent to Grafter’s fused traversal. Specifically, HecateL achieves competitive performance against Grafter, where both candidates get more than 50% running time reduction over the unfused traversal.

Sequential Vector-Based Traversal. The traversals in real-world compilers like Clang [37] leverage vectors for iterating children. Because the vector-based layout typically leads to better cache locality and reduces the number of dynamic dispatching due to virtual functions, it is crucial for a traversal synthesizer to explore different design choices. However, Grafter does not support vector-based representation due to limitation in its static analysis. On the other hand, as shown in Figure 12 and Figure 13, it only takes HecateV a few lines of changes to refactor a linked-list-based traversal to its vector-based version. In particular, HecateV achieves around 70% running time reduction and almost 40% speed-up over Grafter’s fused traversal.

Parallel Vector-Based Traversal. As Grafter tacitly assumes that fusion opportunities should be exploited whenever possible, it’s designed to reduce the number of tree node visits. This heuristic, despite being effective in some scenarios, may prevent further optimizations and lead to sub-optimal traversals in terms of overall running time.

Consider the fused example shown in Figure 14(b): the fused loop iterates over the children to call a traversal function c->fusedCalc() before updating the running maximum for certain values. Assuming that each c->fusedCalc() is independent from each other, we can "de-fuse" the for loop into two: as shown in Figure 14(c) the first loop is decomposed into parallel traversals, and the second loop updates the running maximum in a sequential fashion. Although the "de-fused" traversal yields a higher number of node visits, it can benefit from parallel execution if the cost of children traversal calls far outweighs the cost of the sequential second visit. This example shows how unconditionally fusing computations might prevent fine-grained optimizations.

As shown in Figure 11, as the tree size grows, the speed-up brought by the parallel variant HecateP gradually overcomes its overhead, bringing an additional 23% improvement over the sequential vector-based variant HecateV.

The evaluation shows that, with minimal effort, Hecate can effectively explore traversals of different design choices.

6.3 Synthesizing Layout Engine in FTL

To show the advantages of our domain-specific encoding, we compare Hecate against FTL [32], a synthesizer specialized for layout engines. In particular, FTL introduces a Prolog-style declarative language for expressing partial schedules with holes. After that, FTL devises a sophisticated synthesis algorithm that leverages Prolog’s unification algorithm for effectively generating the schedule as a composition of parallel tree traversals.

Benchmarks. Since FTL is not actively maintained anymore, we can only run it on three variants of attribute grammars (i.e., CSS
class Inner: public Box {
    /* class def same as unfused */
    public:
    #
    vector<Box*> cs;
    #
    void Inner::calcWidth() {
        w = w0;
    }
    for (auto c : cs) {
      c->calcWidth();
      w = max( w, c->w );
    }
    h1 += c->h;
    void Inner::calcHeight() {
        h1 = 0;
    }
    for (auto c : cs) {
      c->calcHeight();
      h1 += c->h;
    }
    void Inner::calcHeight() {
        h1 = 0;
    }
    for (auto c : cs) {
      c->calcHeight();
      h1 += c->h;
    }
    void Inner::fusedCalc() {
      h1 = 0;
    }
    for (auto c : cs) {
      c->fusedCalc();
      h1 += c->h;
    }
    void Inner::fusedCalc() {
      h1 = 0;
    }
    for (auto c : cs) {
      c->fusedCalc();
      h1 += c->h;
    }
    h1 += c->h;
    h = max( h0, h1 );
  }

  (a) unfused version

  (b) fused version

  (c) "de-fused" version

Figure 14: Pseudo-code class definitions (unfused, fused and "de-fused" versions) for rendering tree example, optimized with vector data structure. Only key refactorings are listed.

<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).

rules) that are not supported by Grafter: 1) CSS-float represents the basic CSS rules together with float rules [6, 8, 10], 2) CSS-margin denotes the basic CSS rules together with rules for margin collapse [5, 7, 9], and 3) CSS-full is the superset of the previous two and it incorporates the most challenging CSS features such as absolute position, margin collapse, float, and others. Figure 15( left) summaries the statistics of the attribute grammars in terms of number of rules.

Performance. Figure 15(right) shows the results of the comparison and it takes Hecate only a fraction of time in FTL. Specifically, for the CSS-float grammar, it takes FTL 189 seconds to synthesize the traversal while it only takes Hecate 39 seconds to finish. As the number of rules grows in CSS-full, both tools take a bit longer time, but Hecate is still 5X faster than FTL. To confirm the effectiveness of our domain-specific encoding, we run the general-purpose encoding Hecate on all three benchmarks. Hecate can not terminate within 30 mins.

This evaluation shows that Hecate can be extended to compute complex CSS semantics supported by real-world layout engines and the domain specific encoding plays a crucial on scaling the tool on those complex benchmarks.

7 RELATED WORK

The closest analog to Hecate in the existing literature is FTL [32]. Like Hecate, FTL synthesizes schedules for browser layout engines; unlike Hecate, FTL translates the layout semantics to a Prolog program, and uses the Prolog kernel to search for schedules. Also unlike Hecate, FTL is specialized to a particular solver, constraint encoding, attribute grammar language, and schedule language; Hecate is considerably more flexible, and its trace language allows it to scale to larger and more complex attribute grammars.

Grafter is another synthesizer for tree traversals. Unlike Hecate, Grafter is based on static analysis, where it generates automata that captures the dependencies indicated between statements and invokes a deterministic algorithm to rewrite and fuse traversals into more compact ones, thus synthesizing new traversals. While Grafter is fast, extending it to new specifications may require extra expert knowledge to devise new tree fusion theories.

Several authors have produced formalizations of browser layout like those used by Hecate to define the layout semantics. Besides those introduced by FTL, Cassius [35] formalizes a subset of browser layout in linear real arithmetic in order to synthesize CSS from examples using an SMT solver, and VizAssert [34] extends that formalization with finitization reductions to support a large subset of the CSS standard, including floating layout, which is widely used in modern web pages but is tricky even for experts to reason about. The Cornipickle [22] project, meanwhile, used first-order modal logic to define visual proprities of specific web pages. VizAssert later adapted Cornipickle’s logic to SMT reasoning. Besides web page layout in particular, there is a rich history of work on constraint-based systems for specifying and synthesizing layouts [1, 4, 23, 44, 48, 52] and on domain-specific languages for describing structured graphics [51] and visual manipulations [17].

Tools for layout problems in web pages form a rich and dynamic topic in the software engineering literature [3, 28–30, 49, 50]. Tools to detect parts of a web page that render differently in different browsers [16, 31, 39] are a large and important subclass of these tools. While these tools are aimed for web page developers (unlike Hecate, which may be used by browser developers), their number demonstrates the challenges that layout bugs impose on practitioners and the importance of the problems Hecate addresses. In fact, practitioners commonly test their web pages against specific instances of browsers and operating systems by loading pages in virtual machine instances [11–13]. The manual inspection that this easy-to-use and widely adopted testing approach requires could be reduced if better tooling reduces the frequency or severity of layout bugs.

Many attribute grammar formalisms [26] assume dynamic scheduling, in contrast to the fully static scheduling presented here. For a large class of attribute grammars, the problem of scheduling an
attribute grammar onto a sequence of traversals is known to be NP-hard [19], though polynomial-time scheduling algorithms for restricted classes of grammars exist [32]. However, these restricted classes have not been classified or well-studied.

Constraint solving based on satisfiability modulo theories [33] has become a powerful tool for program analysis as practical, high-performance solvers have become available [2, 18, 21]. Solver-based verification and synthesis tools have a long and rich history in programming languages community [27, 42, 43]. Traditional solver-aided tools use a custom constraint solver or manually translate problems into constraints for a specific existing solver. Solver-aided domain-specific languages [45, 47] instead automatically generate solver constraints based on symbolic execution and custom language extensions. For example, Rosette [46] uses Racket’s meta-programming features to provide a high-level interface to several solvers. Hecate is build atop Rosette, but uses its trace language to abstract over the low-level features presented in generic Rosette constraints and significantly improves runtime.

8 CONCLUSION

We propose Hecate, a novel framework for synthesizing tree traversals. The core of Hecate is a domain-specific symbolic compilation strategy for tree traversal synthesis that maintains the engineering advantages of solver-aided language, yet achieves better performance. The evaluation shows that Hecate’s tree language is expressive as it supports traversals from all Grafter benchmarks and complex features in layout engines. Hecate’s domain-specific symbolic compilation is efficient as it achieves 3x speed-up compared to general-purpose symbolic compilation. Finally, our case analysis shows that Hecate can explore traversals of different design choices with simple modifications.

ACKNOWLEDGMENTS

This work has been supported in part by the ACI OACś1535191, FMitF CCF-1918027, OIA-1936731, SaTC-1908494, the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352), the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01, grants from DARPA FA8750-14-C-0011 and DARPA FA8750-16-2-0032, the Google faculty award, as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm.

A CASE STUDY: AST

Compilers routinely traverse abstract syntax trees (ASTs) to perform program transformation and validation. The AST benchmark models a simple imperative language with variable assignments, arithmetic expressions, decrement and increment statements, conditional statements, and functions. The benchmark further implements a total of six de-sugaring and optimization passes: 1) de-sugaring decrement statements, 2) de-sugaring increment statements, 3) constant propagation, 4) replacement of variable references to constants, 5) constant folding, and 6) elimination of unreachable branches.

Similar to the RenderTree benchmark, we evaluate the performance of three variants of Hecate: 1) HecateL: sequential linked-list-based traversal, 2) HecateV: sequential vector-based traversal, 3) HecateP: parallel vector-based traversal. Figure 16 summarizes the comparison with Grafter. Each line corresponds to one of the variants. The x-axis shows the tree size, while the y-axis shows normalized running time over the unfused baseline, averaged over 10 trials.

Overall, the linked-list based traversal HecateL achieves around 50% running time reduction compared to unfused baseline, which is similar to Grafter fused traversal. However, the choice of linked lists for representing lists of statements is not so much a necessity as a limitation from Grafter’s static analysis. HecateV in contrast, lets us replace the underlying data structure with vectors with minimal code modification, leading to a further 10% reduction in running time. Furthermore, HecateP is able to take advantage of the data-independency between optimization passes on different AST functions. Although there is inevitable overhead when the parallel schedules synthesized by Hecate are evaluated on smaller trees, the performance gains gradually overcome the overhead, and result in over 75% running time reduction over the unfused baseline.

REFERENCES


