Synthesis of Parallel Tree Programs with Domain-Specific Symbolic Compilation

by

Nate Yazdani

Supervised by Ras Bodik

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2017

Presentation of work given on _____

Thesis and presentation approved by _____

Date _____

ABSTRACT

This thesis proposes a new method of domain-specific symbolic compilation with which to construct efficient and programmable synthesis systems for parallel tree programs. This thesis presents an implementation of such a synthesis systems for scheduling the parallel evaluation of attribute grammars. This synthesis system is evaluated on the goals of efficiency and programmability against the current state of the art for programmable synthesis systems. The synthesis system fares well in this evaluation.

Contents

1 INTRODUCTION

1.1 Motivation

Many program optimizations boil down to scheduling computation at some level of abstraction. Modern compilers incorporate many such optimizations. Notable examples are instruction scheduling [19], polyhedral loop transformation [4, 5], and automatic vectorization [1, 12]. Instruction scheduling orders a set of instructions so as to expose maximal instruction-level parallelism while respecting data dependences. Polyhedral loop transformation rearranges loop nests to improve data locality and to expose opportunities for further optimization of the loop body. Automatic vectorization transforms scalar computation into equivalent vector computation (*e.g.*, inside a loop body following polyhedral loop transformation). In each instance, a portion of the source program is transformed into a semantically equivalent version that scores higher on some optimization objective, such as the degree of potential instruction-level parallelism.

Recent work has applied scheduling techniques for powerful optimization and even *synthesis* of programs at a high level of abstraction. Prominent examples are automatic parallelization and vectorization of recursive procedures [9, 15] and synthesis of image-processing pipelines [11, 13, 14]. In both cases, automated reasoning on an appropriate high-level abstraction of program behavior is key to tractability. The first work is a source-level program optimization but infers such an abstraction of relevant program behavior using a complex analysis [18]. The second work, a system named Halide, instead starts from a carefully designed high-level abstraction of the intended program behavior — provided by the user — and then synthesizes an optimal implementation in a *scheduling language*. The Halide scheduling language expresses the implementation decisions of interest, from which one can generate an implementation in a language like C. These systems are indicative of a new paradigm in program construction, wherein users specify intended program behavior with a high-level *domain-specific language* and rely on the computer to derive an efficient implementation through scheduling into a lower-level language.

Even without synthesis, explicit representation of relevant implementation decisions with a scheduling language has significant benefits for programmers. In fact, industry adoption of the Halide system began on the basis of manual scheduling. When schedules are represented explicitly with a domain-specific language, software engineers can quickly experiment with different low-level implementations and search for one with desirable performance characteristics. When available, synthesis techniques augment the user's search with automation.

1.2 Problem

The goal of this thesis is synthesis of parallel tree programs with both *efficiency* and *programmability* of the underlying synthesis system. In particular, the synthesis problem is posed as scheduling the parallel evaluation of an attribute grammar, provided by the user as the specification. Efficiency enables synthesis to scale to larger and more complex problem instances, also enabling use in interactive or dynamic applications. Programmability facilitates adaptation of the synthesis system to evolving user needs, ideally as easily as changing a classical interpreter of the schedule language.

No prior synthesis system for parallel tree programs has achieved *both* efficiency and programmability. Efficient synthesis is typically accomplished with hand-written constraint generators, but the complexity of an efficient constraint encoding sacrifices programmability. Programmable synthesis has more recently been accomplished with *solver-aided domain-specific languages* [16], which translate a classical schedule interpreter to constraints; however, the general-purpose constraint generation often precludes a particularly efficient encoding, with workarounds introducing the same drawbacks as direct constraint generators. Figures 1 and 2 illustrate and describe the high-level architecture of these two approaches. Both omit details of the process to extract a final program

$$G \in L_{AG} \xrightarrow{\text{generation}} \phi \in L_{Constraint}$$

Fig. 1. High-level architecture of a synthesis system based on a *constraint generator*. The system analyzes the attribute grammar G and syntactically translates it to a constraint ϕ in some constraint language.

$$G \in L_{AG} \xrightarrow{\text{interpretation}} \mathsf{int}(\mathsf{G}, \mathsf{t}, \mathsf{sch}[\vec{x}]) \in L_{Racket} \xrightarrow{\text{partial evaluation}}$$
$$\mathsf{P}[\vec{x}] \in L_{Core} \xrightarrow{\text{symbolic evaluation}} \phi[\vec{x}] \in L_{FOL}$$

Fig. 2. High-level architecture of a synthesis system based on a *solver-aided domain-specific language*. The system runs the schedule interpreter int on an attribute grammar G with an example tree t and a symbolic schedule $\mathsf{sch}[\vec{x}]$, constructed in terms of primitive symbolic variables $\vec{x}$. Partial evaluation reduces this to a residual program $\mathsf{P}[\vec{x}]$ in some core symbolic language, for which symbolic evaluation generates a constraint system $\phi[\vec{x}]$ in first-order logic.

$$G \in L_{AG} \xrightarrow{\text{interpretation}} \mathsf{int}_{\mathsf{trace}}(\mathsf{G}, \mathsf{t}, \mathsf{sch}[\vec{x}]) \in L_{Racket} \xrightarrow{\text{partial evaluation}}$$
$$\mathsf{P}[\vec{x}] \in L_{Trace} \xrightarrow{\text{symbolic evaluation}} \phi[\vec{x}] \in L_{ILP}$$

Fig. 3. High-level architecture of a synthesis system based on a *symbolic abstraction* for parallel program traces. The system runs the schedule interpreter $\mathsf{int}_{\mathsf{trace}}$ (implemented with the symbolic trace) on an attribute grammar G with an example tree t and a symbolic schedule $\mathsf{sch}[\vec{x}]$, constructed in terms of primitive symbolic variables $\vec{x}$. Partial evaluation vec this to a residual program $\mathsf{P}[\vec{x}]$ in the symbolic trace language, for which symbolic evaluation generates a constraint system $\phi[\vec{x}]$ as an integer linear program.

from a constraint solution, which is the same for all systems discussed and not the focus of this work.

1.3 Solution

We propose a method of *domain-specific symbolic compilation* for synthesis of parallelism schedules that maintains the engineering advantages of solver-aided domain-specific languages, by means of a *symbolic abstraction* for program traces, yet achieves greater scalability in constraint solving by orders of magnitude. Figure 3 illustrates and describes the high-level architecture of the synthesis system. In particular, this thesis presents three contributions:

(1) A symbolic abstraction for parallel program traces — a *symbolic trace* — that naturally expresses parallel scheduling problems.
(2) A domain-specific extension for symbolic compilation that generates efficiently solvable constraints for operations on a symbolic trace.
(3) An efficient and programmable synthesizer for parallel evaluation schedules of an attribute grammar, built with the prior two contributions.

1.4 Overview

Here, we provide a brief overview of the rest of the thesis. Section 2 describes the formalism of attribute grammars; the syntax of L_{AG}; their definition of an input domain of trees; their computational semantics on such trees; and an example in the form of the *HVBox* attribute grammar. Section 3 describes a language of tree traversal schedules; the syntax of L_S; the computational semantics of such schedules; a simple interpreter of L_S; and two example schedules for *HVBox*. Section 4 describes the abstraction offered by the symbolic trace; the high-level semantics of its operations; and an interpreter for L_S implemented in its language, L_T. Section 5 describes general-purpose symbolic compilation, as implemented by the Rosette programming language, and extension of domain-specific symbolic compilation for the symbolic trace language L_T. Section 6 describes a nontrivial extension of the scheduling problem, affecting both L_{AG} and L_T, that requires significant modifications to both versions of the scheduling interpreter, illustrating the programmability of the symbolic trace language. Section 7 analyzes the efficiency and programmability in greater detail on several benchmarks. Section 8 overviews key related works. Finally, Section 9 summarizes the main points and ideas of this thesis.

$$
\begin{array}{lcll}
\mathit{Iface} & ::= & \texttt{interface}\ \mathit{Id}\ \{\ \mathit{Attr}^*\ \} & \\
\mathit{Class} & ::= & \texttt{class}\ \mathit{Id}_1\ :\ \mathit{Id}_2\ \{\ \mathit{Child}^*\mathit{Attr}^*\mathit{Rule}^*\ \} & \\
\mathit{Child} & ::= & \texttt{child}\ \mathit{Id}_1\ :\ \mathit{Id}_2; & \\
\mathit{Attr} & ::= & \texttt{input}\ \mathit{Id}\ :\ \mathit{Type}; \mid \texttt{output}\ \mathit{Id}\ :\ \mathit{Type}; & \\
\mathit{Rule} & ::= & \mathit{Id}_1.\mathit{Id}_2\ :=\ \mathit{Expr}; & \\
\mathit{Expr} & ::= & \mathit{Id}_1.\mathit{Id}_2 \mid \mathit{Id}(\mathit{Expr}^*) \mid \mathit{Expr}_1 \circ \mathit{Expr}_2 \mid \ldots & \circ \in \{+, -, *, /, \&\&, ||\} \\
\mathit{Type} & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \ldots &
\end{array}
$$

Fig. 4. Syntax for the language of attribute grammars, L_{AG}.

2 ATTRIBUTE GRAMMARS

An *attribute grammar* [8] is a declarative formalism for a large class of tree computations. An attribute grammar consists of two components:

(1) A *context-free grammar* describing the shape of the tree domain.
(2) A set of *attributes* with *evaluation rules* for computing attributes from each other.

The evaluation rules for an attribute grammar are unordered with respect to each other. To evaluate an attribute grammar on some tree is to perform every evaluation rule on every applicable node of the tree, while respecting dependencies. Evaluation must therefore determine an order of computation that satisfies all dependencies between attributes. The *scheduling* problem for attribute grammars is to find an order that satisfies the dependencies for any possible tree. With such a schedule, one may statically compile the evaluation of an attribute grammar into an efficient, deterministic program.

2.1 Syntax

Figure 4 presents the syntax for a language of attribute grammars, L_{AG}. A user specifies the desired parallel tree program with an attribute grammar in this syntax.

An attribute grammar is a natural form of specification for users, as its constructs correspond to those in object-oriented programming. The syntax of L_{AG} emphasizes this correspondence. An interface serve a role comparable to an abstract class or, more literally, an interface in some object-oriented languages; an interface defines the common set of attributes for all implementing classes. A class serves a role similar to classes in mainstream object-oriented languages, though the purpose corresponds most closely to “plain-old data" or “struct" types in some object-oriented languages. A class in L_{AG} inherits from a single interface ($\texttt{class}\ \mathit{Id}_1\ :\ \mathit{Id}_2\ \{\ \ldots\ \}$) and specifies the following features of its node instances: the full set of attributes, the set of child nodes with expected interfaces ($\texttt{child}\ \mathit{Id}_1\ :\ \mathit{Id}_2$), and the set of evaluation rules defining local or child attributes.

2.2 Semantics

The computational meaning of an attribute grammar is summarized by the semantics given in Figure 6 [1], which assumes that the input tree is well-formed according to the rules given in Figure 5. Initially, every attribute starts out uninitialized, indicated with a $\perp$ value, except input attributes. Evaluating the attribute grammars means repeatedly performing evaluation rules until none are

[1]The intent of the semantics is to elucidate the meaning of attribute-grammar evaluation and draw attention to the inherent nondeterminism. Low-level details are therefore omitted, as they would hinder rather than aid clarity of these features.

$$\textsc{Attr}_\text{I}\ \frac{\text{self} \in \text{tree} \qquad \text{attr} \in \text{self.class.inputs}}{\text{self.fields[attr]} \neq \bot} \qquad \textsc{Attr}_\text{O}\ \frac{\text{self} \in \text{tree} \qquad \text{attr} \in \text{self.class.outputs}}{\text{self.fields[attr]} = \bot}$$

Fig. 5. Selected rules for a well-formed input tree of an attribute grammar.

$$\textsc{Eval}\ \frac{\begin{array}{c}\text{self} \in \text{tree} \qquad \text{rule} \in \text{self.class.rules} \\ \text{self.fields[rule.lhs]} = \bot \qquad \forall \text{attr} \in \text{rule.deps},\ \text{self.fields[attr]} \neq \bot\end{array}}{\text{self.fields[rule.lhs]} \leftarrow \text{eval(self, rule.rhs)}}$$

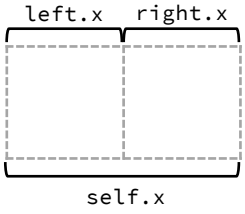
Fig. 6. Nondeterministic semantics for evaluation of an attribute grammar.

left uninitialized. To do so, one must nondeterministically select (1) a node and (2) an evaluation rule such that (a) the left-hand-side attribute of the rule is uninitialized and (b) all dependencies (*i.e.*, attributes appearing in the right-hand-side expression) are initialized. A schedule for evaluation of an attribute grammar codifies a deterministic procedure to perform these nondeterministic choices.

2.3 Example

Attribute grammars naturally express many problems of practical interest, such as document layout and data visualization. Figure 7 (on the following page) shows a large fragment of an attribute grammar for *HVBox*, a data visualization that recursively partitions a space horizontally and vertically. The portions omitted simply set up the root node, *i.e.*, initialize `x` and `y` attributes to `0`.

```
interface HVBox {
    output w, h, x, y : int;
}
// HVBox ::= HVBox_left HBox HVBox_right
class HBox : HVBox {
    child left, right : HVBox;
    left.x := self.x;
    right.x := self.x + left.w;
    left.y := self.y;
    right.y := self.y;
    self.w := left.w + right.w;
    self.h := max(left.h, right.h);
}
// HVBox ::= HVBox_lower VBox HVBox_upper
class VBox : HVBox {
    child upper, lower : HVBox;
    upper.x := self.x;
    lower.x := self.x;
    upper.y := self.y + lower.h;
    lower.y := self.y;
    self.w := max(upper.w, lower.w);
    self.h := upper.h + lower.h;
}
// HVBox ::= Leaf
class Leaf : HVBox {
    input w_0, h_0 : int;
    self.w := self.w_0;
    self.h := self.h_0;
}
```



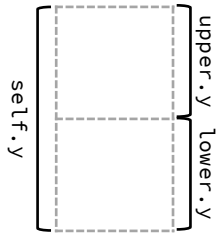


Fig. 7. The *HVBox* attribute grammar and illustration of HBox and VBox cells.

$$
\begin{aligned}
\mathit{Sched} &::= \mathit{Sched}_1 \texttt{ ; ; } \mathit{Sched}_2 \mid \mathit{Sched}_1 \texttt{ || } \mathit{Sched}_2 \mid \mathit{Trav} \\
\mathit{Trav} &::= \mathit{Order} \texttt{ \{ } \mathit{Visit}^* \texttt{ \}} \\
\mathit{Order} &::= \texttt{pre} \mid \texttt{post} \\
\mathit{Visit} &::= \mathit{Id} \mapsto \mathit{Slot}^* \\
\mathit{Slot} &::= \mathit{Id}_1 . \mathit{Id}_2
\end{aligned}
$$

Fig. 8. Syntax of the language of tree traversal schedules, L_S.

3 TREE TRAVERSAL SCHEDULES

Synthesis requires an abstraction with which to reason about the implementation space. For evaluation of an attribute grammar, *tree traversal schedules* are a natural choice, as an efficient evaluation strategy must traverse the tree in some manner to apply evaluation rules. While not every possible traversal is expressible in this language of tree traversal schedules, programmability of the synthesis system will allow users to augment the language as needed, unlike past work.

3.1 Syntax

Figure 8 shows the syntax for the language of tree traversal schedules, L_S. A schedule takes one of three forms: a parallel composition of two subschedules, a sequential composition of two subschedules, or a traversal pass. A traversal pass is either pre-order (top-down) or post-order (bottom-up). A traversal pass also specifies the sequence of evaluation rules — each identified by its left-hand-side attribute — to perform at each node visit by class.

3.2 Semantics

Figure 9 gives the deterministic semantics for the language of tree traversal schedules [2]. A sequential composition runs the subschedules left to right. A parallel composition may run the subschedules either left to right or right to left. (While it may seem inaccurate to specify the parallel semantics in this way, it is perfectly sound for evaluation of an attribute grammar; in this context, there is no temporary, intermediate with which concurrent threads could interfere.) A traversal pass runs the denoted visitor procedure on the tree in the named traversal order.

An important feature of this semantics is that the crucial decisions affecting performance — traversal order and schedule composition — are abstracted away from the lower-level details of individual evaluation rules. This separation makes synthesis tractable, as an outer loop can enumerate many “skeleton" schedules (that omit placement of individual evaluation rules) given an efficient mechanism to complete (or reject) each skeleton schedule.

3.3 Examples

Figure 10a shows a sequential tree traversal schedule for *HVBox*. Box dimensions are computed in a preorder, or bottom-up, traversal, so that attribute values may flow upward through the tree. Subsequently, box coordinates are computed in a postorder, or top-down, traversal so that values may flow downward through the tree. The sequential ordering of the two traversal passes ensures that box dimensions are computed before the box coordinates.

[2]The intent of the semantics is to elucidate the meaning of attribute-grammar evaluation and draw attention to the newly enforced determinism. Low-level details are therefore omitted, as they would hinder rather than aid clarity of these features.

$$\textsc{Seq}\ \frac{\langle \text{sched}_1, \text{tree}\rangle \Downarrow \text{tree'} \qquad \langle \text{sched}_2, \text{tree'}\rangle \Downarrow \text{tree''}}{\langle \text{sched}_1 \,;\, ; \text{sched}_2, \text{tree}\rangle \Downarrow \text{tree''}}$$

$$\textsc{Par}_\text{L}\ \frac{\langle \text{sched}_1 \,;\, ; \text{sched}_2, \text{tree}\rangle \Downarrow \text{tree'}}{\langle \text{sched}_1 \,||\, \text{sched}_2, \text{tree}\rangle \Downarrow \text{tree'}} \qquad \textsc{Par}_\text{R}\ \frac{\langle \text{sched}_2 \,;\, ; \text{sched}_1, \text{tree}\rangle \Downarrow \text{tree'}}{\langle \text{sched}_1 \,||\, \text{sched}_2, \text{tree}\rangle \Downarrow \text{tree'}}$$

$$\textsc{Pre}\ \frac{\langle \text{preorder(tree, visitor)}, \text{tree}\rangle \Downarrow \text{tree'} \qquad \text{visitor} = [\![\text{visits}]\!]}{\langle \texttt{pre\{visits\}}, \text{tree}\rangle \Downarrow \text{tree'}}$$

$$\textsc{Post}\ \frac{\langle \text{postorder(tree, visitor)}, \text{tree}\rangle \Downarrow \text{tree'} \qquad \text{visitor} = [\![\text{visits}]\!]}{\langle \texttt{post\{visits\}}, \text{tree}\rangle \Downarrow \text{tree'}}$$

Fig. 9. Deterministic semantics for the language of tree traversal schedules, L_S

```
post {
  HBox ↦ self.w, self.h
  VBox ↦ self.w, self.h
  Leaf ↦ self.w, self.h
} ;; pre {
  HBox ↦ left.x, right.x,
         left.y, right.y
  VBox ↦ lower.x, upper.x,
         lower.y, upper.y
  Leaf ↦
}
```

(a) A sequential tree traversal schedule.

```
(post {
  HBox, VBox, Leaf ↦ self.w
} ;; pre {
  HBox ↦ left.x,  right.x
  VBox ↦ lower.x, upper.x
  Leaf ↦
}) || (post {
  HBox, VBox, Leaf ↦ self.h
} ;; pre {
  HBox ↦ left.y, right.y
  VBox ↦ lower.y, upper.y
  Leaf ↦
})
```

(b) A parallel tree traversal schedule.

Fig. 10. Two tree traversal schedules for the *HVBox* attribute grammar.

More complex schedules also exist, with potentially interesting performance characteristics. Figure 10b shows a tree traversal schedule for *HVBox* that makes explicit use of parallelism. This schedule specifies a parallel composition of two subschedules, each of which is a sequence of two traversal passes as in the previous schedule. The first branch computes box widths and x-coordinates, while the second branch computes box heights and y-coordinates. This division of work onto two threads is correct because each subschedule is data-independent of the other; each thread reads and writes a set of attributes disjoint from the other.

Even for a simple attribute grammar like *HVBox*, dividing the work onto two cores can be highly advantageous. Parallel computation may be faster, but moreover, the data independence of the two threads promotes effective caching, especially with a structure-of-arrays tree representation. For a tree several times larger than available cache memory, performance may increase significantly due to this factor alone.

```
(define (interp_check grammar sched tree)
  (match sched
    [(seq s1 s2) ; s1 ;; s2
     (interp_check grammar s1 tree)
     (interp_check grammar s2 tree)]
    [(par s1 s2) ; s1 || s2
     ; Check subschedules for data independence
     (interp_check grammar (seq s2 s1) (copy tree))
     (interp_check grammar (seq s1 s2) tree)]
    [(pre visits) ; pre { visits }
     (preorder (visitor grammar visits) tree)]
    [(post visits) ; post { visits }
     (postorder (visitor grammar visits) tree)]))

(define ((visitor grammar visits) node)
  (let ([class (lookup grammar (classname node))]
        [slots (lookup visits (classname node))])
    (for ([slot slots])
      (for/all ([slot_i slot]) ; for each concrete value...
        (check node (get-rule class slot_i))))))

(define (check self rule)
  (let ([attr (lhs rule)]
        [expr (rhs rule)])
    (assert (not (ready? self attr)))
    (for ([dependency (dependencies expr)])
      (assert (ready? self dependency)))
    (set-ready! self attr)))
```

Fig. 11. An interpreter to check an L_S schedule on an L_{AG} attribute grammar and tree.

3.4 Interpreter

Figure 11 shows a simple interpreter for this scheduling language, implemented in the Rosette programming language and intended for use as a solver-aided domain-specific language (*i.e.*, amenable to general-purpose symbolic compilation). The only construct used in the interpreter specific to Rosette (*i.e.*, not found in its host language Racket) is the `for/all` operation, which applies a body to every concrete alternate of a symbolic value; Section 5 describes this operation in greater detail. The interpreter assumes that both the attribute grammar and the tree traversal schedule passed some sort of basic type checking and are free of any undefined references and ill-typed expressions. Since the interpreter is intended for schedule synthesis, it abstracts away the actual attribute values and only tracks whether or not an attribute is ready (*i.e.*, initialized). Similarly, it computes a parallel composition in both sequential directions, so as to ensure the absence of data-dependence in either direction. Low-level details and helper procedures are omitted for clarity.

Operation	Description
`(choose` $v_1 \ldots v_n$`)`	returns a symbolic choice from the given concrete alternate values, to construct a *program hole*, written ??
`(alloc` ℓ`)`	returns a concrete location, ℓ
`(read` ℓ`)`	logs a read from the given concrete location ℓ
`(write` ℓ`)`	logs a write to the given concrete location, ℓ
`(step)`	advances the program to the next statement
`(fork ([`x c`]) `e`)`	evaluates the body e with x bound to each concrete alternate of the symbolic choice c
`(parallel` e_1 e_2`)`	evaluates e_1 then e_2 while checking for data independence across the two threads

Table 1. Operations of the symbolic trace language L_T, each returning `(void)` unless noted otherwise..

4 SYMBOLIC ABSTRACTION FOR TRACES

The symbolic abstraction for parallel program traces — or symbolic trace — is an interface upon which one may implement a synthesis system dealing with parallelism. This interface forms the *symbolic trace language* L_T, a domain-specific language embedded in the Racket programming language.

4.1 Syntax

Table 1 summarizes the individual operations of the symbolic trace. These operations are embedded in the host language, Racket. Therefore, these syntactic forms permit free combination and composition with syntax and definitions from that programming language, including its extensive standard library. The example interpreter in Figure 12 demonstrates this syntactic interoperability.

4.2 Semantics

The semantics of the symbolic trace language are, effectively, what one might guess: each operation is appended to a global trace of program evaluation. However, a final call to a constraint generation operation expresses three correctness conditions on the accumulated symbolic trace:

(1) *Single assignment.* Every location is written at most once.
(2) *Dependency satisfaction.* Every read to a location is preceded by the write to that location.
(3) *Data independence.* Every read to a location is not dependent on a write by a concurrent thread.

With these operations, the user implements a tracing interpreter to define the validity of a parallel program according to the semantics of their scheduling language, such as L_S, as shown in Figure 12. Such a tracing interpreter is *abstract* in its reading and writing of memory. Rather than actually producing and consuming the resources through which dependencies are carried, the tracing interpreter simply reports when reads and writes happen to each memory location. While seemingly an inconvenience, this situation is actually preferable for schedule synthesis, because a valid schedule ought to work correctly for any possible values in the tree or other input data. The following section discusses symbolic compilation of this language to an efficiently solvable constraint system in the form of an integer linear program.

The symbolic trace provides direct support for scheduling program statements over sequential and parallel control flow. Partial evaluation reduces away other control-flow constructs, such as

```
(define (interp_trace grammar sched tree)
  (match sched
    [(seq sched_1 sched_2) ; scheds_1 ;; sched_2
     (interp_trace grammar sched_1 tree)
     (interp_trace grammar sched_2 tree)]
    [(par sched_1 sched_2) ; sched_1 || sched_2
     ; Check subschedules for data independence
     (parallel
       (interp_trace grammar sched_1 tree)
       (interp_trace grammar sched_2 tree))]
    [(pre visits) ; pre { visits }
     (preorder (visitor grammar visits) tree)]
    [(post visits) ; post { visits }
     (postorder (visitor grammar visits) tree)]))

(define ((visitor grammar visits) node)
  (let ([class (lookup grammar (classname node))]
        [slots (lookup visits (classname node))])
    (for ([slot slots])
      (fork ([slot_i slot]) ; for each concrete value...
        (trace class node (get-rule class slot_i))))))

(define (trace self rule)
  (let ([attr (lhs rule)]
        [expr (rhs rule)])
    (for ([dependency (dependencies expr)])
      (read (location-of self dependency)))
    (write (location-of self attr))
    (step)))
```

Fig. 12. An interpreter to check an L_S schedule, implemented in the symbolic trace language L_T.

recursion. Enumerative search over these higher-level constructs gives the synthesis system control over optimization and is reasonably efficient [3].

4.3 Interpreter

Figure 12 shows an interpreter for L_S implemented in L_T. Note the significant similarity to the earlier interpreter for L_S given in Figure 11.

```
post {
    HBox ↦ ??²(HBox.rules)
    VBox ↦ ??²(VBox.rules)
    Leaf ↦ ??²(Leaf.rules)
} ;; pre {
    HBox ↦ ??⁴(HBox.rules)
    VBox ↦ ??⁴(VBox.rules)
    Leaf ↦ ??⁰(Leaf.rules)
}
```

Fig. 13. A sketch of a tree traversal schedule for *HVBox*

5 SYMBOLIC COMPILATION

Building a synthesis systems for tree traversal schedules is ideally a straightforward process from here. The rest of this section considers synthesis from a *sketch* of a tree traversal schedule with symbolic syntax — or program holes — only for attribute slots. In other words, the synthesis problem is simplified to assigning attributes to available slots in the sketch of a tree traversal schedules. The sketch is built using a construct for symbolic choice, available both in general-purpose and domain-specific symbolic compilation. A schedule sketch is therefore a symbolic value for a schedule syntax. Figure 13 presents a schedule sketch for the *HVBox* attribute grammar (*cf.*, Figure 7). Despite initial appearances, this simplification does not actually sacrifice anything; most practical synthesis systems based on symbolic compilation rely on enumerative search in a space of program sketches for which completion of an individual sketch (*i.e.*, assigning concrete syntax to program holes) is tractable by symbolic compilation and constraint solving [3]. Section 5.1 discusses this further.

5.1 Synthesis and Verification

The techniques for synthesis and verification by symbolic compilation are independent of whether symbolic compilation is general-purpose or domain-specific.

For both methods of symbolic compilation, the standard technique of counterexample-guided inductive synthesis (CEGIS) [2] is applicable. In CEGIS, synthesis is performed by a search-verify loop for some mechanically checkable specification. The search finds a candidate program correct on a set of concrete example inputs and the verification checks whether the candidate program's correctness generalizes to the entire domain, perhaps up to a bound. If not, then the verification generates a counterexample input on which the candidate program was incorrect, and the CEGIS loop returns to the search phase with an example set augmented by this counterexample.

For fast and optimal synthesis, an outer loop should be used to enumerate sketches specifying the schedule's control flow, *e.g.*, traversal types and compositions [3]. While not strictly necessary, enumerating sketches expressing the control flow allows finer control for optimization and may result in more efficiently solvable queries to the solver. With this separation, optimizing to a new cost model is as simple as writing a new sketch enumerator. To avoid trying trivially similar sketches that differ only in the allocation of holes to traversals, the symbolic trace allows extra holes in the schedule that it fills with a user-provided no-op.

For schedule verification, the interpreter is run on a concrete schedule and a symbolic input. Since state-of-the-art techniques for symbolic compilation require a bound on the size of symbolic values, this direct approach can only provide bounded verification. For scheduling problems with highly regular input domains, one can sometimes choose example inputs carefully so as to generalize to the entire, possibly infinite domain, as is the case for attribute grammars.

5.2 Symbolic Evaluation

In either style of symbolic compilation, there is a notion of *symbolic branching*, denoted `for/all` in Rosette and `fork` in the symbolic trace language. Symbolic branching explores the evaluation of a body for each *symbolic alternate* (*i.e.*, possible concrete value) of a given symbolic value, under a *path condition* that now includes an assertion that the active symbolic alternate is the true one. An example illustrates the idea:

$$\texttt{(for/all ([x (choose } v_0\ v_1\texttt{)]) (assert (P x)))} \Downarrow \big(v = v_0 \Rightarrow P(v_0)\big) \wedge \big(v = v_1 \Rightarrow P(v_1)\big)$$

This operation is the core of symbolic evaluation. Symbolic compilation is the generalized notion of translating a program's semantics to constraints, based on this technique and others like it.

5.3 General-Purpose

In order to handle nearly arbitrary programs, general-purpose symbolic compilation tracks the evolution of the symbolic program state in a generic way, and it consequently derives generic constraints. Unfortunately, such generic constraints are often inefficiently solvable. At a small scale, this is inconsequential, but realistic synthesis problems require more efficiently solvable constraints. In the worst case, general-purpose symbolic compilation builds a tree of symbolic possibilities, such as $\mu(\phi_1, \mu(\phi_2, w, x), \mu(\phi_3, y, z))$, where w, x, y, z are concrete values guarded by the logical predicates $\phi_1 \wedge \phi_2,\ \phi_1 \wedge \neg\phi_2,\ \neg\phi \wedge \phi_3,\ \neg\phi_1 \wedge \neg\phi_3$, respectively. For an assertion such as $P(\mu(\phi_1, \mu(\phi_2, w, x), \mu(\phi_3, y, z)))$, general-purpose symbolic compilation generates the following constraint:

$$\Big((\phi_1 \wedge \phi_2) \implies P(w)\Big) \wedge \Big((\phi_1 \wedge \neg\phi_2) \implies P(x)\Big) \wedge \Big((\neg\phi_1 \wedge \phi_2) \implies P(y)\Big) \wedge \Big((\neg\phi_1 \wedge \neg\phi_2) \implies P(z)\Big)$$

As the depth of this tree of symbolic values and the complexity of the predicate P increases, this becomes increasingly problematic.

To get an intuition for how one can outperform general-purpose symbolic compilation, suppose, in the above example, that the predicate P could not possibly hold for some concrete value unless that value's guard also held. For instance, $P(w)$ could not hold unless $\phi_1 \wedge \phi_2$ also held (at the very least). In that case, the following constraint is equivalent to the one above, given the added domain context:

$$P(w) \vee P(x) \vee P(y) \vee P(z)$$

This formula is much simpler to think about, both for humans and for solvers.

5.4 Domain-Specific

The symbolic trace leverages the restricted nature of L_T to generate an efficiently solvable constraint system as an integer linear program.

5.4.1 Domain Insight. Collectively, the generated constraints must ensure that all *dependences* are satisfied, meaning that all reads from a location follow the write into that location. A straightforward encoding is to require that the `step` count at a read is higher than the `step` count at a write.

An easy — and consequential — optimization becomes apparent. Since step counters really only encode a partial order on trace events, symbolic control-flow joins may soundly over-approximate step counters by taking their maximum. Now, the solver may reason about a fixed, total order on every possible trace event, some of which may simply not happen. This gives a significant improvement to the performance of constraint solving, but one can do much better.

A less obvious encoding improves the solver's performance by several orders of magnitude. Rather than ensuring that all dependences are met, we pose the equivalent constraints ensuring

that no *antidependences* exist, meaning that a `write` must not happen after any `read` from the same location. The advantage of ruling out antidependences over only requiring dependences is that these constraints serve a purpose somewhat analagous to *conflict clauses* in the conflict-driven clause learning algorithm for Boolean satisfiability. More precisely, these antidependence constraints express to the solver what decisions to avoid, allowing it backtrack quickly after making an incorrect guess.

Note that the concept of antidependences does not exist in the original language of tree traversal schedules nor either of the interpreters. General-purpose symbolic compilation thus cannot switch from dependences to ruling out antidependences without global and nontrivial reasoning about the high-level problem.

5.4.2 Constraint Encoding. Each symbolic choice c created by `(choose` $v_1 \ldots v_n$`)` is represented as a collection of pairs (b_i, v_i), where b_i is a fresh symbolic binary variable (for communication with the ILP solver). The symbolic choice c takes on the value v_i for which $b_i = 1$. To ensure that a solution assigns exactly one concrete value to each symbolic choice, $\texttt{int}_T$ emits the constraint $\sum_i b_i = 1$ for each c.

As the program is interpreted, the symbolic trace logs every `read` or `write` with the current guard and the current value of a hidden program counter. The `step` operation is implemented simply by incrementing this program counter. For convenience, we will use $R[n, \ell]$ and $W[n, \ell]$ as mappings from a program counter n and a location ℓ to the set of guards under which the respective `read` or `write` occurs. A guard is implemented as a set of binary variables (originating from the binary variables of symbolic choices), the conjunction of which is the truth value of the guard.

The first correctness condition, that every location is written at most once, is ensured by the constraint $\sum_n \sum_{g \in W[n,\ell]} \wedge(g) \leq 1$ for every location ℓ, where $\wedge(g)$ is a helper procedure that creates the conjunction of a set g of binary variables in ILP. When $\wedge(g) = 1$ for a particular $g \in W[n, \ell]$ (for some n and ℓ), then that particular `write` (to location ℓ) will occur in the program trace (specifically at the program counter n). Since there should be no more than one `write` to ℓ, $\wedge(g')$ should be 0 for the other $g' \in W[n', \ell]$ (for any $n' \neq n$). Hence, the sum of all $\wedge(g)$ for $g \in W[n, \ell]$ for all n should be less than or equal to 1 for any particular ℓ.

The second correctness condition, that every `read` is preceded by a `write` to the same location, is ensured by the constraint $\wedge(g_r) + \sum_{n_w \geq n_r} \sum_{g_w \in W[n_w, \ell]} \wedge(g_w) \leq 1$ for each $g_r \in R[n_r, \ell]$ and each program counter n_r and location ℓ. Rather than directly stating that the program counter of a `read` must be less than (the expression for) that of the corresponding `write`, this constraint states that every antidependence must be avoided; that is, there should not exist a `write` to ℓ after any `read` to ℓ. Since locations have write-once semantics, all `write`s to the same location are mutually exclusive with each other as well as with any `read` earlier in the trace, because a later `write` rules out the possibility of scheduling an earlier `read`. Since the solver could satisfy these antidependence constraints by simply never scheduling a `write` in the first place, we generate dependency constraints of a similar nature. These antidependence constraints are key to efficient solver performance, as the solver can now recognize bad partial assignments much sooner.

The constraint for the third correctness condition (*i.e.*, data independence for parallel threads) is a simple refinement of the previous constraint. The parallelism `(parallel` e_1 e_2`)` construct is expanded such that e_1 happens before e_2, and then the symbolic trace treats writes in e_1 as antidependences with respect to reads e_2 for the purpose of constraint generation. This correctly encodes the semantics that e_1 and e_2 must exhibit data independence.

$$
\begin{array}{lcl}
\mathit{Child} & ::= & \texttt{child}\ \mathit{Id}_1\ \texttt{:}\ \texttt{[}\mathit{Id}_2\texttt{];} \mid \ldots \\
\mathit{Rule} & ::= & \mathit{Id}_1\texttt{.}\mathit{Id}_2\ \texttt{:=}\ \mathit{Expr}\texttt{;} \\
 & \mid & \mathit{Id}_1\texttt{.}\mathit{Id}_2\ \texttt{:=}\ \texttt{fold}\ \mathit{Expr}_1\ \texttt{..}\ \mathit{Expr}_2\texttt{;} \\
\mathit{Expr} & ::= & \mathit{Id}_1\texttt{[-].}\mathit{Id}_2 \mid \mathit{Id}_1\texttt{[0].}\mathit{Id}_2 \mid \mathit{Id}_1\texttt{[\$].}\mathit{Id}_2 \\
 & \mid & \ldots
\end{array}
$$

(a) Syntax of L'_{AG} relative to L_{AG}.

$$
\begin{array}{lcl}
\mathit{Slot} & ::= & \texttt{loop}\ \mathit{Id}\ \texttt{\{}\mathit{Subslot}^*\ \texttt{\}} \mid \mathit{Subslot} \\
\mathit{Subslot} & ::= & \mathit{Id}_1\texttt{.}\mathit{Id}_2
\end{array}
$$

(b) Syntax of L'_S relative to L_S.

Fig. 14. Differences in the syntax relative to the earlier originals.

6 EXTENSIBILITY

In this section, we demonstrate the extensibility of synthesis systems built on the symbolic trace with a nontrivial extension to the semantics of L_{AG} and L_S that we implement in the associated synthesizer. Specifically, we extend the syntax and semantics of L_{AG} and L_S to support a statically unbounded number of child nodes. We then show the needed modifications to the interpreter variants from Figure 11 and Figure 12 to synthesize programs in this extended language of tree traversal schedules, L'_S.

6.1 Syntax

Figure 14a shows the differences in syntax of L'_{AG}, the modified language of attribute grammars, from the original language, L_{AG}, from Figure 4. Two components of classes, child declarations (*Child*) and evaluation rules (*Rule* and *Expr*), now have additional syntactic forms and semantics for child node sequences. Figure 14b shows the corresponding difference in syntax relative to L_S, which is just to add an explicit loop within node visits.

6.2 Semantics

The differences in semantics for both $L_{AG'}$ and L'_S are low-level but worth discussing, as they are subtle. The new form of *Child* indicates the presence of a child node sequence (*i.e.*, list) named Id_1, each of which implements some class inheriting the named interface named Id_2. The first form of *Rule* is carried over from the original syntax but has new semantics for child node sequences. If Id_1 on the left-hand side is the name of a child node sequence, the rule defines the attribute on each child in the sequence, and the right-hand-side expression may refer to any other attribute Id_1 . Id_3 in the interface Id_2. The new form of *Rule* is fold, that computes a particular attribute Id_2 on each node in the child node sequence Id_1 in a context where the previous attribute's value is available as Id_1 `[-]` Id_2 (initialized with $Expr_1$ for the first node in the sequence). The new form of slot, `loop` *Id* `{` *Subslot** `}`, means to compute in lockstep (*i.e.*, each rule on the first node, then each rule on the second, and so on) the indicated evaluation rules for each node in the child node sequence named *Id*.

6.3 Interpreters

Figure 15 shows the interpreter to check tree traversal schedules modified from Figure 11 to support child node sequences and looped evaluation rules. Like the original, this interpreter assumes that the grammar and schedule passed a simple type checking process that ensures the semantics for the grammar and schedule are well-defined, now including the conditions for a looped evaluation rules. Figure 16 shows the corresponding differences for an interpreter implemented in the symbolic trace

```
(define (interp_check grammar sched tree)
  ...) ; unchanged

(define ((visitor grammar visits) node)
  ...) ; unchanged

(define ((visitor grammar visits) node)
  (let ([class (lookup grammar (classname node))]
        [slots (lookup visits (classname node))])
    (for ([slot slots])
      (for/all ([slot_i slot]) ; for each concrete value...
        (match slot_i
          [(loop child subslots)
           (define virtual (accumulator))

           (for ([subslot subslots])
             (for/all ([subslot_i subslot]) ; for each concrete value...
               (let ([rule (get-rule class subslot_i)])
                 (check null virtual node rule))))

           (for ([this (get-children node child)])
             (for ([subslot subslots])
               (for/all ([subslot_i subslot]) ; for each concrete value
                 (let ([rule (get-rule class subslot_i)])
                   (check virtual this node rule)))))]
          [subslot
           (check null null node (get-rule class subslot))])))))

(define (check prev this self rule)
  (let ([attr (lhs rule)]
        [expr (rhs rule)])
    (for ([dependency (dependencies expr)])
      (assert (ready? prev this self dependency)))
    (assert (not (ready? prev this self attr)))
    (set-ready! prev this self attr)))
```

Fig. 15. An interpreter modified for L'_S and L'_{AG}

language, modified from the interpreter given in Figure 12. As a caveat, we note that the helper procedure `accumulator` is implemented differently, but only to use an association list (*i.e.*, map data structure) provided by the symbolic trace language.

```
(define (interp_check grammar sched tree)
  ...) ; unchanged

(define ((visitor grammar visits) node)
  ...) ; unchanged

(define ((visitor grammar visits) node)
  ...) ; same as above, except calls (trace ...) instead of (check ...)

(define (trace prev this self rule)
  (let ([attr (lhs rule)]
        [expr (rhs rule)])
    (for ([dependency (dependencies expr)])
      (read (location-of prev this self dependency)))
    (write (location-of prev this self attr))
    (step)))
```

Fig. 16. The interpreter implemented in L_T modified for L'_S and L'_{AG}

7 EVALUATION

This section evaluates the schedule synthesizer against the original research goals: efficiency and programmability of the synthesis system.

7.1 Efficiency

We evaluate the performance of schedule synthesis against the current state of the art for efficient and programmable synthesis systems, solver-aided domain-specific languages [16]. We compare against two solver-aided domain-specific languages written in the Rosette programming language, a solver-aided programming language extending the normal Racket programming language [17]. The symbolic trace used the IBM CPLEX solver for integer linear programs, whereas the solver-aided domain-specific languages used the Z3 solver for satisfiability modulo theories (*i.e.*, first-order logic enriched with decision procedures).

The two solver-aided domain-specific languages demonstrate the difference in outcome with and without expert knowledge of the underlying symbolic compiler. The first, the Natural SDSL, is designed and implemented in a natural programmatic style, as one would idiomatically in the host language, Racket. The other, the Expert SDSL, is designed and implemented by an expert to optimize symbolic compilation. The key difference between these two implementations is in their low-level representation of attribute state: The Natural SDSL uses a functional association list, whereas the Expert SDSL uses an association list of mutable cells. This seemingly inconsequential distinction triggered enormous differences in outcome for general-purpose symbolic compilation.

Benchmarks. We evaluate performance for schedule synthesis on the attribute grammars listed in Table 2. These attribute grammars represent the layout computation for several data visualizations, taken from the literature [10]. The third, *Treemap*, also includes light data analysis, in the form of filtering and weighting of data.

Name	Interfaces	Classes	Evaluation Rules
HVBox	2	3	18
Sunburst	2	3	23
Treemap	2	7	109

Table 2. Benchmarks for evaluation.

Metrics. We measure the time taken in symbolic compilation and constraint solving for each system to find the first sequential schedule and the first parallel schedule. We use the same simple enumerator of schedules sketches and the same set of example trees for all systems. Each system is effectively run on an identical sequence of problems for each benchmark.

Results and Analysis. For efficiency of synthesis, the symbolic trace outperformed the SDSLs. The Natural SDSL was not performant enough to complete a single benchmark within the time out of 15min. Even the Expert SDSL even timed-out on the benchmark *Treemap* (Parallel).

7.2 Programmability

We evaluate the programmability of the symbolic trace against the solver-aided domain-specific languages. In particular, we analyze the added support for statically unbounded children to all three synthesis systems.

Results and Analysis. All three systems required changes with only superficial differences, as shown in Section 6.

Benchmark	Symbolic Trace	Natural SDSL	Expert SDSL
HVBox (Sequential)	6.241	-	27.870
HVBox (Parallel)	10.660	-	69.399
Sunburst (Sequential)	14.428	-	51.271
Sunburst (Parallel)	21.803	-	140.088
Treemap (Sequential)	69.264	-	834.479
Treemap (Parallel)	218.229	-	-

Table 3. Time (s) spent in symbolic compilation. Dashes indicate time-out after 15min.

Benchmark	Symbolic Trace	Natural SDSL	Expert SDSL
HVBox (Sequential)	0.07	N/A	0.127
HVBox (Parallel)	0.130	N/A	0.194
Sunburst (Sequential)	0.07	N/A	0.231
Sunburst (Parallel)	0.131	N/A	0.481
Treemap (Sequential)	6.11	N/A	32.256
Treemap (Parallel)	3.81	N/A	N/A

Table 4. Time (s) spent in constraint solving. N/A indicates time-out above.

8 RELATED WORK

Solver-aided domain-specific languages [16] have made it easier to link powerful constraint solvers with applications. The Rosette solver-aided programming language [17] uses Racket's metaprogramming features to provide a high-level interface to several solvers. In contrast, the traditional approach is to either write a custom, heuristics-based algorithm or to manually translate a problem into constraints for a specific existing solver.

Deterministic schedulers [6, 7] map a given attribute grammar to an evaluation schedule. As deterministic algorithms, these schedulers are typically fast but highly inflexible. Modifying the language of attribute grammars or schedules may require nontrivial redesign of the whole scheduling algorithm to account for every possible interaction of new and old language constructs. Moreover, imprecision of heuristics inhibits optimization for all but the simplest cost models. As a result of these drawbacks, all deterministic schedulers known to the authors generate simple, sequential evaluation schedules, impractical for modern performance demands.

A constraint generator encodes schedule search for an attribute grammar into constraints, whose solutions identify correct schedules. Manual constraint generation lacks programmability; a change in the language of attribute grammars or schedules requires reasoning across a complex indirection through the constraint language. Moreover, the constraint encoding must often be highly complex to achieve adequate performance, and this hurts programmability further. This complexity and the generally error-prone nature of manual constraint generation also makes it difficult to achieve high confidence in the correctness of the synthesis system.

For instance, the FTL system [10] translates the dependency constraints of an attribute grammar into a Prolog program, and uses the Prolog core as a solver to find a schedule that satisfies the constraints. The constraint solver, constraint encoding, attribute-grammar language, and schedule language are all fixed and changing any of them requires substantial modification to the system. Discussion with the authors indicated that this was indeed their own experience in the system's development, including the support for statically unbounded children. By contrast, the system presented here is more flexible in each of these dimensions.

9 CONCLUSION

This thesis has presented a novel system for schedule synthesis that achieves both efficiency and programmability. This was accomplished though a novel symbolic abstraction that extended the symbolic compilation process for efficient trace-based reasoning. The evaluation showed that synthesis performance favored this new approach, and that programmability was improved when efficiency is a concern. In general, we envision significant research potential down the path of domain-specialization in symbolic compilation, particularly towards increased interoperability and reliability of different symbolic abstractions.

REFERENCES

[1] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (Oct. 1987), 491–542. https://doi.org/10.1145/29873.29875

[2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*.

[3] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 775–788. https://doi.org/10.1145/2837614.2837666

[4] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem: Part I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (October 1992), 313–348. https://doi.org/10.1007/BF01407835

[5] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21 (1992), 389–420. Issue 6. http://dx.doi.org/10.1007/BF01379404 10.1007/BF01379404.

[6] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. 1990. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 209–222. https://doi.org/10.1145/93542.93568

[7] Ken Kennedy and Scott K. Warren. 1976. Automatic Generation of Efficient Evaluators for Attribute Grammars. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages (POPL '76)*. ACM, New York, NY, USA, 32–49. https://doi.org/10.1145/800168.811538

[8] Donald E. Knuth. 1968. Semantics of Context-Free Languages. In *In Mathematical Systems Theory*. 127–145.

[9] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU Scheduling and Execution of Tree Traversals. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 41, 2 pages. https://doi.org/10.1145/2851141.2851174

[10] Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. 2013. Parallel Schedule Synthesis for Attribute Grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2442516.2442535

[11] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[12] David A. Padua and Michael J. Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29, 12 (Dec. 1986), 1184–1201. https://doi.org/10.1145/7902.7904

[13] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

[14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[15] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient Execution of Recursive Programs on Commodity Vector Hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 509–520. https://doi.org/10.1145/2737924.2738004

[16] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[17] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[18] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. 2015. Tree Dependence Analysis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 314–325. https://doi.org/10.1145/2737924.2737972

[19] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal Instruction Scheduling Using Integer Programming. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/349299.349318