Parallel Layout Engines: Synthesis and Optimization of Tree Traversals

Leo Meyerovich

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-242
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-242.html

December 20, 2013

Parallel Layout Engines: Synthesis and Optimization of Tree Traversals

by

Leo Alexander Meyerovich

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodík, Chair
Professor George Necula
Professor David Wessel

Fall 2013

Parallel Layout Engines: Synthesis and Optimization of Tree Traversals

Abstract

Parallel Layout Engines: Synthesis and Optimization of Tree Traversals

by

Leo Alexander Meyerovich

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodík, Chair

Mobile web browsers and data visualization tools require a performance boost. Parallelization poses an opportunity because commodity computers feature fast and energy-efficient multicore, subword-SIMD, and GPU hardware. However, layout engines in both browsers and visualizations have resisted parallelization thus far. This thesis introduces techniques for parallel computing over trees and applies them to generating layout engines.

Our solution is twofold. First, we show how to specify layout languages in the attribute grammar model of constraints over trees. Second, we address outstanding challenges in parallelizing attribute grammars and computations over trees. Our resulting attribute grammar compiler generates layout engines for various layout languages and parallel hardware.

We provide several individual contributions:

1. We specify the functional and parallel behavior of common layout language primitives.
2. We present a language for scheduling parallel tree traversals and a static verifier for ensuring a schedule will solve the attributes of an arbitrary layout in a safe order.
3. We introduce a parallel programming model where programmers may partially specify the parallel schedule and call an optimizing synthesizer to complete the schedule.
4. We design a scheduling algorithm that is fast and modular over scheduling constructs.
5. We optimize tree traversals for SIMD hardware by automatically staging dynamic memory allocation and clustering diverging tasks.
6. We optimize parallel tree traversals for MIMD architectures through a load-balancing heuristic that approximates work stealing.

To evaluate our approach, we present two case studies. First, we generated the first parallel webpage layout engine that can for the most part render complex sites such as Wikipedia. Second, we generated interactive data visualizations that support up to 1,000,000 data points in real-time. Both case studies have been further validated industrially.

"The Hitchhiker's Guide to the Galaxy has this to say on the subject of flying. There is an art, it says, or rather a knack to flying. The knack lies in learning how to throw yourself at the ground and miss. Pick a nice day, it suggests, and try it. The first part is easy. All it requires is simply the ability to throw yourself forward with all your weight, and the willingness not to mind that it's going to hurt. That is, it's going to hurt if you fail to miss the ground. Most people fail to miss the ground, and if they are really trying properly, the likelihood is that they will fail to miss it fairly hard. Clearly, it's the second point, the missing, which presents the difficulties."

Life, the Universe, and Everything – Douglas Adams

Contents

List of Figures

List of Tables

Acknowledgments

When we first started, most people thought that parallelizing the browser was an impossible idea. That sounded like an idea that should be proven wrong, and I've been lucky to work with mentors who could keep such research paths clear: Ras Bodik and Shriram Krishnamurthi, and over several eye-opening summers, Roger Webster, Ben Livshits, Todd Mytkowicz, Wolfram Schulte, and Herman Venter.

All of the best work was achieved through close collaboration with others: Ari Rabkin, Raluca Sauciuc, and our ever-wonderful research assistants, Matthew Torok and Eric Atkinson. I hope to have been even a small fraction as helpful to the students in the Berkeley Parallelism Lab and the Open Source Quality group as they were to me. Our industrial collaborators were key to navigating the mysteries of browser internals and parallel architectures: the Mozilla team including Rob O'Callahan, Dave Herman, Brendan Eich, Boris Zbarsky, Intel researchers including Moh Haghighat and Gans Srinivasa, Nokia researchers such as Kimmo Kuusilinna and Per Ljung, and Samsung researchers including Tasneem Brutch, and Steven Eliuk. And, of course, thank you to the committee for bearing through this process with continued enthusiasm: George Necula, Krste Asanovic, and David Wessel.

Finally, to my family and Julie, Jono, and Lee: thank you for having traveled with me this far, and I'm excited to see what happens next!

Chapter 1

Introduction

At the most practical level, this thesis examines how to parallelize a language for laying out visualizations. By specifying important subsets of layout languages with a restricted form of constraints over trees, we find we can abstract the problem to automatically parallelizing systems of constraints. Our solution combines language design, program synthesis, and high-performance algorithms. We introduce innovatations in each of these areas in order to build our end-to-end system.

Two important applications of parallel layout guide our work. First, mobile web browsers need more efficient layout engines. Power and energy constraints prevent browsers from running on increasingly small devices but with the performance expected on today's bigger form factors. Second, as part of the rise of data science, interactive visualizations need to support magnitudes bigger datasets. We reduce the challenge behind both case studies to that of parallelization. For future small devices, strongly scaling parallelization provides a power- and energy-efficient way to exploit increasing processor capabilities (Asanovic et al., 2006; Jones et al., 2009). For scaling visualizations to bigger datasets, weakly scaling parallelization enables the visualizable dataset size to increase with the number of parallel processors.

Parallelizing layout faces a variety of technical challenges. First, correctly implementing even a sequential layout language already challenges developers. Second, performance concerns about runtime overheads, task scheduling, and data representation requires optimizing parallel tree traversals for different parallel hardware architectures. Third, our optimizations, in turn, impact the form of parallelism that our system must support reasoning about. Finally, in building our system, we needed to optimize the performance of the schedule synthesis algorithm. Prominently, we needed to extend the synthesizer to support different types of parallelism.

Our solution is a synthesizer that statically schedules an attribute grammar as a composition of parallel tree traversals. In the context of a web browser, the input to the synthesizer is the CSS layout language's semantics. The synthesizer generates a sequence of parallel tree traversals such that, given any syntactically well-formed tree with some attributes on nodes already defined, running the fixed schedule will solve all the remaining attributes according to CSS's attribute grammar. A compiler then takes the schedule and code generates a

layout engine that implements it with algorithms optimized for different parallel hardware architectures.

In contrast to our approach, current industrial systems rely upon natural language specifications and reference implementations. Informal specifications suffer from ambiguous and conflicting definitions. Reference implementations often require reverse engineering to understand. Relying upon informal specifications and general reference implementations presents challenges in extending, reimplementing, and aggressively optimizing a layout language.

Achieving automatic parallelization for our applications required novel techniques. First, our language of tree traversal schedules had to be flexible enough to describe useful parallelism in layout language constructs yet restricted enough to facilitate optimization. Second, we needed to create an algorithm to assist in verifying and even inferring the parallel schedule for an attribute grammar. Furthermore, as specifications grew in size, we needed new linguistic constructs to combine manual and automatic scheduling.

Finally, we had to optimize the data representation and runtime schedule of tree traversals in order to see significant speedups from parallelization. For SIMD architectures, we created new staged dynamic memory allocation and divergent task clustering techniques. We optimized for MIMD architectures by designing a semi-static scheduler that combines the low overheads and temporal data locality of static scheduling with the load balancing benefits of dynamic scheduling.

1.1 Dissertation Overview

The remainder of this chapter presents an example of how to build a layout language using existing techniques and then demonstrates how to improve the process with the contributions of each chapter. First, we outline the sequence of improvements.

Chapter 2: Mechanizing Layout Languages with Extended Attribute Grammars. Chapter 2 overviews the attribute grammar formalism and describes how we use it to address the question of how to specify the functional behavior of common layout language constructs. We found attribute grammars to be inexpressive, e.g., lacking facilities for loops and code sharing; therefore, Chapter 2 also introduces expressive extensions to attribute grammars and how we reduce reasoning about the extensions to reasoning about more canonical attribute grammars.

Chapter 3: Parallel Layout with Checkable Static Tree Traversal Schedules. Chapter 3 examines the structure of parallelism latent within layout solving. To express the parallelism, we augmented the attribute grammar formalism with a language for scheduling parallel tree traversals. Whereas an attribute grammar defines attribute values as a system of directed constraints, the scheduling language defines the order to execute those constraints. Using the scheduling language, Chapter 3 identifies and formalizes latent parallelism within common layout language primitives. Finally, in the spirit of the usual attribute grammar verification procedure, Chapter 3 introduces a static schedule verifier that verifies that solving any layout according to a parallel schedule will solve all of the layout attributes without races.

Chapter 4: Parallel Schedule Synthesis. Manually designing parallel schedules for our layout languages proved difficult. Small changes in an attribute grammar may require large changes in the schedule. These changes are non-obvious, tedious, and can even introduce critical performance bugs. Chapter 4 presents a new parallel programming model to address these problems and a schedule synthesis algorithm to perform the underlying automation. Programmers partially specify parts of the parallelization scheme that concerns them and then call an optimizing synthesizer to schedule the remainder. For example, the programmer may choose to specify that the first tree traversal iterates top-down and in parallel over the tree and that the last traversal is sequential and in order, and then rely upon the synthesizer to schedule the remaining traversals and what node attributes to compute within each traversal. The synthesizer will find a correct and optimized completion of the schedule. If it cannot, the synthesizer either signals the presence of potential logical bugs in the functional specification, e.g., missing definitions, or details the unorderable data dependencies in the partial schedule, e.g., cycles.

Chapter 5: Optimizing Parallel Tree Traversals for SIMD and MIMD. The last chapter examines how to optimize parallel tree traversals for SIMD and MIMD hardware. Applying preexisting techniques yielded few speedups and even slowdowns. For both types of hardware, we found the need to optimize the data representation of the tree and the order of nodes within one traversal. For SIMD architectures, we stage parallel memory allocation (Chapter 3) and, in this chapter, show how to cluster tasks that would otherwise diverge based on input. For MIMD architectures, we show how to combine the load balancing benefits of dynamic scheduling with the low-overheads and temporal locality of static scheduling. Our MIMD approach is semi-static. It partitions the tree into tiles and schedules the traversal over tiles with a heuristic that approximates work stealing.

1.2 Motivating Example

We present most of our techniques in terms of implementing `H-AG`, a simple layout language for positioning nested horizontal boxes. Figure 1.1 shows key concepts for a layout language. A layout engine implements a layout language and takes as input an attributed tree. For example, leaf nodes may be letters, images, or other media. Leaf nodes likely explicitly define their width and height attributes, but leave their position implicit. An intermediate box would leave both its size and position implicit. Figure 1.1a shows such a partially attributed tree. By defining the intermediate nodes to be horizontal boxes, the designer implicitly constrains their children to be positioned side-by-side, and their width to be the sum of their children widths. Different layout languages support different types of nodes.

The job of the layout engine is to compute all of the undefined attribute values (Figure 1.1b). The input format is typically more sophisticated, e.g., allows percentages rather than just absolute values. Likewise, the solution may solve for more than just the above attributes, e.g., by solving details about borders, margins, and how to render graphics. Because attribute values may depend upon one another, the challenge is to determine a correct

```
<S>
  <HBox>
    <HBox>
      <Leaf w=20 h=5/>
      <Leaf w=15 h=7/>
    </HBox>
    <Leaf  w=15 h=5/>
  </HBox>
</S>
```

(a) **Input tree.**

```
<S>
  <HBox w=50 h=7 x=0 y=0>
    <HBox w=35 h=7 x=0 y=0>
      <Leaf w=20 h=5 x =0 y=0/>
      <Leaf w=15 h=7 x=20 y=0/>
    </HBox>
    <Leaf  w=15 h=5 x=35 y=0/>
  </HBox>
</S>
```

(b) **Output tree.**

```
annotateSizes(tree.root);
annotatePositions(tree.root);
```

(c) **Schedule of traversals.**

```
def annotateSizes (v):
  if v.type == HBox:
    annotateSizes(v.child[0])
    annotateSizes(v.child[1])
    v.w = v.child[0].w + v.child[1].w
    v.h = max(v.child[0].h, v.child[1].h)
  elif v.type == S:
    annotateSizes(v.child[0])
    v.w = v.child[0].w
    v.h = v.child[0].h
```

(d) **Traversal 1 (bottom-up).**

```
def annotatePositions (v):
  if v.type == HBox:
    v.child[0].x = v.x
    v.child[1].x = v.x + v.child[0].w
    v.child[0].y = 0
    v.child[1].y = 0
    annotatePositions(v.child[0])
    annotatePositions(v.child[1])
  elif v.type == S:
    v.child[0].x = 0
    v.child[0].y = 0
    annotatePositions(v.child[0])
```

(e) **Traversal 2 (top-down).**

Figure 1.1: **Layout language of horizontal boxes.** Input layout tree, solved output, and parts of a statically scheduled and sequential layout engine.

order for solving the different attributes.

Figures 1.1c–e provide intuition for the implementation of a typical layout engine. Given any tree of horizontal boxes to lay out, the layout engine solves all the attributes in two traversals (Figure 1.1c). The first traversal computes all of the size attributes (Figure 1.1d); the second computes all of the positions (Figure 1.1e). The second traversals requires size information to compute the positions. Therefore, sequentially sequencing the traversals guarantees that all of the sizes are available before the second traversal runs. The first traversal recursively iterates over the tree to assign all of the width and height attributes based on neighboring attributes (Figure 1.1d). Due to data dependencies between attributes, the traversal only computes the attributes during the bottom-up phase of the recursion.

We call the combination of traversals and the order of attributes to compute within a traversal the *schedule*. A *dynamic* scheduler makes the decisions at runtime. Language implementors avoid dynamic scheduling due to high overheads from tracking every instance of every attribute and when individual data dependencies between attributes are satisfied. Our example uses the more common and efficient *static* scheduling approach, where the relative order of node traversals and attribute computations are fixed at compile time. Note

that because the input tree is not known statically, a static schedule must be correct for all possible input trees. Finally, as schedules such as `H-AG`'s typically require multiple traversals, we further define a *visit's schedule* as the schedule for one tree traversal: the order of node access within the traversal and the order that node attributes are computed within a visit to the node.

Much of our work is in determining how to find efficient schedules and optimize their implementation. Optimizing implementations requires determining how to represent a tree in memory and the order to traverse different nodes on different parallel architectures.

Why Parallelization

Layout engine developers, who have already optimized the low-hanging fruit, face diminishing returns from optimization efforts. This thesis explore a new direction by exploiting the shift to parallel architectures in commodity hardware. We explore two forms of parallelization: weak scaling and strong scaling. To optimize for the increasing workload sizes (weak scaling), we must identify significant amounts of concurrency. To optimize for more processors (strong scaling), we need to better utilize modern hardware. We briefly explain these points in turn.

Commodity parallel hardware is already available; it is generally more efficient then sequential alternatives. First, instead of using sequential optimizations with diminishing returns, parallel hardware duplicates simple designs for more linear scaling. For example, if the number of available transistors doubles, a sequential optimization might apply the extra transistors towards an additional 10% realizable performance improvement through a more clever instruction scheduler. The parallel architecture, however, would achieve a 2X improvement by doubling the number of processors. Second, *data parallel* architectures improve power and energy efficiency for repeated operations. For example, when adding two lists, the computer can fetch data in contiguous segments; instead of decoding a plus instruction for each pairwise sum, it need only decode a single vector addition instruction. We thus focused on exploiting parallelism, and, when possible, data parallelism in particular.

Achieving weak and strong scaling is critical for optimizing the performance of web browsers and data visualization:

1. **Weak scaling**. Layout engines should support bigger input trees. Consider data visualization, where data scientists analyze increasingly large datasets. They need interactive visualizations to likewise scale, which corresponds to handling bigger input trees in our system. Weak scaling fixes the workload size per processor and measures the performance for different workload sizes. Our challenge for achieving weak scaling was to identify significant sources of parallelism that relate to the workload size: *data parallelism*.

2. **Strong scaling.** Existing layouts require better performance. For example, we want to run today's webpages on smaller devices with lower power budgets. Strong scaling fixes the workload and measures the performance for different numbers of processors.

Thus, in addition to identifying parallelism in layout, we must also achieve an efficient implementation.

In summary, hardware trends increasingly point towards parallel architectures, and layout engines require both strong and weak scaling.

Why Schedule Synthesis

Schedule synthesis addresses several challenges in parallel programming of layout. Parallel layout was previously proposed by others, such as Brown (1988), but never fully implemented and adopted. This is not for lack of interest. Our proposals for parallelizing components upstream and downstream from the layout solver (Jones et al., 2009; Meyerovich and Bodík, 2010) are being pursued by commercial browser vendors. We have created a schedule synthesizer to solve several problems that have arisen:

1. **Reasoning about data dependencies.** Manually editing both the functional specification and the parallel schedule requires non-trivial reasoning. A small localized change to `H-AG`'s semantics may add a single local data dependency, e.g., an element's height depending on a width, but for an approximation of the global dependency graph, the change may cascade into adding many non-local transitive edges. In turn, the parallel schedule may require many global refactorings, e.g., separating width and height computations into separate traversals. Manually reasoning about and applying schedule changes induced by data dependency modifications is non-trivial.

2. **Aggressive low-level optimization.** Manually implementing a parallel schedule is difficult even if the schedule is fixed. Parallelization is key as a means to improving performance, so a parallel implementation should be faster than existing sequentially-optimized traversals. We found three key classes of optimizations to support: task scheduling, data representation, and existing sequential optimizations. Synthesis is attractive even for sequential tree traversals because manually writing aggressive optimizations requires many man-years of engineering effort.

3. **Reasoning about parallel schedules.** We often guided our automatic parallelizer in what schedule to select. For example, when extending a language with an additional construct, we generally wanted to verify that the previously accepted schedule could still be used. Likewise, for constructs with subtle data dependencies, we would have high-level ideas for how to schedule attributes and then verified our intuition. In both cases, we provided scheduling constraints to the automatic parallelizer.

Exacerbating the above problems is that browser layout engines span hundreds of thousands of lines of code, and for cases of requiring new kinds of data visualizations, we needed designers to build them, even though most designers are not trained in low-level programming.

$S \rightarrow HBOX$
{ HBOX.x = 0; HBOX.y = 0 }

$HBOX \rightarrow \epsilon$
{ HBOX.w = $\text{input}_w()$; HBOX.h = $\text{input}_h()$ }

$HBOX_0 \rightarrow HBOX_1\ HBOX_2$
{ HBOX_1.x = HBOX_0.x;
HBOX_2.x = HBOX_0.x + HBOX_1.w;
HBOX_1.y = HBOX_0.y;
HBOX_2.y = HBOX_0.y;
HBOX_0.h = max(HBOX_1.h, HBOX_2.h);
HBOX_0.w = HBOX_1.w + HBOX_2.w }

Figure 1.2: Attribute grammar defining a layout language of horizontal boxes.

1.3 Mechanizing Layout Languages as Sugared Attribute Grammars

Chapter 2 presents how we functionally specify layout languages. To aid in this process, it also introduces carefully restricted expressive extensions to the attribute grammar formalism.

Figure 1.2 shows how to declaratively define `H-AG` as an attribute grammar. It only specifies the functional behavior, such as the width of intermediate nodes being the sum of the widths of their children. In contrast, it does not specify how to schedule the computation as traversals over the tree.

To support automatic compilation into an executable implementation and various forms of program analysis, attribute grammars restrict specifications in three key ways (Kastens, 1980):

1. **Single assignment.** Every node attribute, such as HBOX_2.x, is defined exactly once. The assignment occurs either in one production where the attribute is attached to the node on the left-hand side of a production `HBOX` → `Y`, or on every production where `HBOX` occurs on the right-hand side. Other languages have similar restrictions, e.g., for functional programming, directed constraints, and dataflow variables.

2. **Local reads and writes.** A node may only access its own attributes or those of neighboring nodes. Our tools use this restriction to combine local reasoning into a global analysis.

3. **Pure functions.** Constraints may invoke arbitrary functions such as max as long as they cause no side effects.

In all cases, the restrictions are designed to reason about local and global data dependencies between attributes. With such understanding, attribute grammar tools can answer questions such as whether all attributes are unambiguously defined, and perform optimizations such as finding a safe parallel schedule for computing them.

In order to express common layout language constructs, we had to design expressive extensions to the attribute grammar formalism . For example, in the case of `H-AG`, our example shows how to compute over binary trees, but popular layout systems use n-ary trees. Many linguistic constructs exist for looping, so our design challenge was to restrict the loop construct enough such that our tree traversal scheduler could analyze uses of it. At the same time, the construct needed to be flexible enough so that we could perform computations such as reductions. For another example, to support designing increasingly large specifications, we needed to introduce constructs for code sharing and information hiding. We show that many of these constructs can be desugared into attribute grammars that resemble the above examples and thus not significantly impact our tool's ability to analyze specifications.

We evaluated our formalism by using it to specify a variety of layout language constructs (Figure 2.11 and Figure 2.13). The include charts, like tree maps, sunbursts, and line graphs, and document layouts, such as grids and nested text with word-wrapping. Finally, we show how to declarative specify rendering, animation, and 3D layouts. Closest to our work, Saraiva and Swierstra (2003) have demonstrated that *fixed* HTML table layouts can be expressed by attribute grammars. However, it is not clear whether (a) their approach expresses the more common and expressive *automatic* HTML table layout algorithm, and (b) if their higher-order attribute grammar formalism is restricted enough for performing our parallelization techniques.

1.4 Parallel Layout with Checkable Static Tree Traversal Schedules

Chapter 3 investigates parallelism latent within common layout language constructs and introduces our scheduling language. We restrict the scheduling language to parallel tree traversals in order to ensure that program analysis, transformation, and optimization is tractable. For example, this chapter shows how to statically verify whether following a particular schedule on an arbitrary input tree will always solve all of the node attributes. Chapters 4 and 5 further exploit the scheduling language restrictions in order to optimize the schedule selection and implementation.

Scheduling Language

Identifying the structure of parallelism within a layout language has been an on-going research challenge. For example, consider the proposal by Brown (1988) to parallelize document layout by computing over each page in a different process. The approach does not

work for LaTeXnor webpages due to word-wrapping, variably sized text, and complex figures introducing data dependencies that cross page boundaries. In H-AG, for example, the horizontal offset of a node depends on the sum of the widths of all of the preceding elements: any partitioning of the tree would cut across this transitive data dependency. Finally, even if some elements are known to be isolated from others, such as a frame in a webpage, our solver would have to identify enough isolated computations to saturate the available processors.

Instead, we split the computation into multiple tree traversals and exploit parallelism within and across traversals. For H-AG, the first traversal computes all the widths, which enables the second traversal to compute all of the horizontal positions in parallel using a top-down (preorder) traversal (Figure 1.1e). As soon as the position of an element and its children widths are known, all of its subtrees may be positioned independently of one another. Figure 3.4b shows the structure of the data dependencies for this second traversal.

We introduced a language for specifying parallel schedules that is orthogonal from the functional specification. For example, the following schedule shows how to parallelize H-AG:

```
1  postorder
2    HBOX0 → HBOX1 HBOX2 { HBOX0.w HBOX0.h }
3    HBOX → ϵ { HBOX.w HBOX.h }
4  ;
5  parPre
6    S → HBOX { HBOX.x HBOX.y }
7    HBOX0 → HBOX1 HBOX2 { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

The schedule specifies the overall parallel structure as a sequence (";") of two types of parallel traversals (postorder and parallel preorder). Within each traversal, it defines the sequence of attributes to evaluate when the traversal reaches a particular type of node (a *production* such as S → HBOX).

Separating H-AG's specification into an attribute grammar for functional behavior and the above schedule enables refactoring one part and then using our automation tools to check or even update the other. Consider the different types of edits in turn:

- **Schedule refactoring.** An enterprising developer might notice (correctly) that H-AG's computation of y positions may be fused into the first traversal and move it. Our verifier would check that no assignments were lost in the refactoring. Furthermore, if the developer assumes (erroneously) that the x position computations may be scheduled for earlier as well, our schedule verifier would throw an error that the computation of a node's x attribute fails due to a read-before-write on its parent node's x attribute.

- **Attribute grammar refactoring.** Layout languages evolve so their designers must reason about changes to implicit data dependencies. For example, extending H-AG with vertical boxes (swapping width computations with height ones and x computations for y ones) introduces a dependency between height and y attributes. In the original H-AG language, y attribute computations could be scheduled in the first traversal, but with the extension, they should be computed in the second traversal after heights are computed in the first (Figure 3.4b). Our verifier would check whether H-AG's extension

can be computed under H-AG's original parallel schedule. As part of addressing performance overheads explored in later chapters, we also use the verifier to ensure that extensions to the attribute grammar do not require adding expensive tree traversals.

In both cases, we find that local changes to one specification may globally impact the other. By separating the schedule from the functional specification and restricting both formalisms, we automate manual global reasoning and refactoring (Chapters 3 and 4).

Schedule Verification

To verify a schedule against an attribute grammar, we show a simple axiomatic system. Every scheduling construct corresponds to a logical judgment. For example, the following rule checks uses of operator ";" for sequencing some traversal p and then traversal q. In particular, it determines whether attributes C will be computed afterwards as long as attributes A are ready beforehand:

$$\frac{\{A\}\ p\ \{B\} \qquad \{B\}\ q\ \{C\}}{\{A\}\ p\ ;\ q\ \{C\}}\ (seq)$$

It recursively performs the check by invoking two more: it checks p computes attributes B and, given attributes B, that q computes C. An important result of the axiomatic formulation is the use of modular reasoning. While theoretically simple, this avoids implementation difficulties we encountered in trying to extend the analyses proposed by others (Kastens, 1980). To update our verifier to analyze an extension of our scheduling language, we generally only had to add or modify the corresponding judgment. Chapter 4 shows how to reformulate the verifier as a synthesizer, which similarly benefited our synthesizer.

Verification is $O(A \log A)$ in the number of attributes. The intuition is that an individual axiom check is linear in the number of attributes written, and every schedule AST node corresponds to a check. Intermediate schedule AST nodes partition the set of attributes so that every tree level checks all the attributes, and there are only $\log A$ levels.

Evaluation on Tricky Schedules

We conclude Chapter 3 by showing static parallel schedules for common layout language constructs. Some schedules were simply lengthy, but others required non-obvious designs; in several cases, this meant increasing the expressive power of our scheduling language.

As an example, supporting word-wrapped text led to introducing a scheduling construct for describing a restricted form of nested parallelism. Word wrapping places each word to the right of the previous one, except when doing so would overflow a paragraph's boundary, the word goes on the next line instead. The position of each word depends on the position of the previous one, which makes word wrapping a sequential traversal over words.

In general, a document contains many small paragraphs of text that are isolated from one another. Our approach is to sequentially lay out an individual paragraph but to compute

over different paragraphs in parallel. The nested traversal scheduling construct generalizes this approach to embedding one type of traversal within another. It consists of an overall traversal, such as parallel preorder, and when the outer traversal reaches a statically-defined contiguous subtree (e.g., a table consisting of row, column, and cell nodes), a region-specific traversal occurs for that region. When the region traversal terminates, the overall traversal continues for the remainder of the tree.

Two other examples in Chapter 3 include supporting dynamic memory allocation and automatic table layout. We vectorize dynamic memory allocations by grouping them into a prefix sum traversal. Unfortunately, a table forms a DAG rather than a tree, which we address by viewing a table as two minimum spanning trees: one where the parent of a cell is its row, and another where the parent is its column.

1.5 Parallel Schedule Synthesis

Designing a parallel schedule requires enough low-level and global reasoning that we explored automatic scheduling. Chapter 4 shows our solution of generalizing our schedule verifier into a synthesizer. Synthesis further enables us to introduce the new parallel programming abstraction of a *schedule hole* (a form of syntactic unification) and the optimization of schedule autotuning. Finally, we present an optimized synthesis algorithm that avoids exponential explosions.

Schedule holes

Programmers specify relaxed schedules by leaving *holes* (Solar-Lezama et al., 2006) for arbitrary terms in the schedule. For example, a programmer may choose to only *sketch* the parallel traversals for `H-AG`:

$$\textbf{parPost}\ ?hole_1\ ;\ \textbf{parPre}\ ?hole_2$$

The synthesizer is responsible for determining correct values for $?hole_1$ and $?hole_2$, which are the schedule terms defining the order of attributes for every type of node within the traversals. Leaving a hole for the entire schedule would correspond to fully automatic parallelization.

We generalized schedule holes to syntactic unification. For example, supporting membership queries enables specifying that the synthesizer may pick any type of traversal for each pass as long as it is parallel:

$$\begin{aligned}&\textbf{Sched} = ?hole_1\ ?hole_2\ ;\ ?hole_3\ ?hole_4,\\ &member(?hole_1, \{\textbf{parPre},\ \textbf{parPost}\}),\\ &member(?hole_3, \{\textbf{parPre},\ \textbf{parPost}\})\end{aligned}$$

We implemented our scheduling language as an embedded DSL in Prolog, and thereby generalized schedule synthesis to the full power of Prolog (Colmerauer, 1990). In our implementation, Prolog provides the member predicate and performs the overall unification.

Optimizing Synthesis and Schedules

The core synthesis algorithm enumerates syntactically valid schedules and tests them with the verifier. Such a design shares the modularity benefits of the axiomatic verifier of Chapter 3, which is critical for simplifying the implementation of our formalism and its extensions. However, brute force enumeration of schedules suffers an exponential explosion in the number of traversals and attributes. Our optimized algorithm addresses both sources of explosion:

- **Prefix expansion of traversals.** We observe that if the beginning of a schedule is correct and that a full schedule exists, there must be some correct suffix that, appended to the schedule prefix, will complete the schedule. For example, if the synthesizer finds that width and heights can be computed in an initial postorder traversal for `H-AG`, it can then need only find subsequent traversals for computing x and y positions. By limiting the search of traversal sequences to successive suffixes, we avoid backtracking and thus the first type of explosion.

- **Iterative refinement of attributes.** Our algorithm avoids examining all combinations of attributes for use within a traversal. It initially over-approximates the set of schedulable attributes within a traversal, such as guessing that "{w,h,x,y}" can all be scheduled in the first traversal. Our verifier will reject the schedule, and as an extension to the algorithm presented in Chapter 3, provides an error message of a non-empty subset of the unschedulable attributes, such as "{x}". The synthesizer removes the bad guesses from the over-approximation and repeats the process until either all attributes are eliminated or the verifier accepts the remaining ones. Because the number of refinement iterations is bounded by the number of attributes, our approach avoids the second source of explosion in automatic scheduling. Our iterative refinement differs from a CEGAR loop (Solar-Lezama et al., 2006) by guaranteeing a solution in polynomial time.

Chapter 4 describes additional optimizations, such as incrementalization in the prefix expansion (Bochmann, 1976), topological sorting (Kastens, 1980) to perform the dependency analysis within the verification step, and using a greedy heuristic for minimizing the number of traversals (Kastens, 1980). Our Prolog library implements all of the optimizations: each one represents pruning the enumeration and therefore a cut for Prolog's unifier.

We leveraged the synthesizer towards optimizing the schedule design through autotuning. As already shown, `H-AG`supports multiple schedules. We modified the synthesizer to enumerate all of them rather than stopping after finding the first one, and then tested each schedule on programmer-provided inputs and select the fastest. For example, parallel schedules are generally longer than sequential ones and rely upon parallel speedups to offset

the overheads from additional traversals: the hardware architecture impacts the schedule selection.

Evaluation

We evaluated the ability of our synthesizer to automatically find parallel schedules for common layout language constructs. It finds parallel schedules for all of them. The most notable case was CSS, which uses 9 passes. The difficulty in CSS was that while the solver found parallel traversals for most of the passes, it could not find the nested traversal because the nesting construct triggers an exponential explosion in the synthesis. By providing a sketch of the expected nesting, we guided the synthesizer to the desired result. Sketches were generally of only the traversal sequence and thus took one line of code. The exception was CSS, which required 3 lines for the traversal sequence and 2 lines to describe the nesting.

Our synthesis algorithm was fast enough for interactive use. In most cases, synthesis was within 30s. The CSS specification is a notable exception due to the nesting. Fully automatic parallelization of CSS took 30 minutes, but by providing the 5 line sketch, we reduced the time down to one minute.

1.6 Optimizing Parallel Tree Traversals for MIMD and SIMD

Chapter 5 shows how to optimize tree traversals for different parallel hardware architectures. Our problem was that our implementations of existing techniques such as Cilk-style (Blumofe et al., 1995) work stealing and data flattening as in NESL (Blelloch et al., 1994) gave little-to-no speedup on our benchmarks, and in many cases, performed worse than sequential execution due to high overheads and poor data locality. Our solution was to optimize the schedule of nodes within a traversal and the data representation of the tree.

We show different techniques for optimizing parallel tree traversals on MIMD and SIMD hardware:

1. **MIMD Tree Traversals: Semi-Static Work Stealing** We optimized MIMD traversals for low overheads, load balancing, and data locality. We drew inspiration from two sources: work stealing (Blumofe et al., 1995) and tiling (Irigoin and Triolet, 1988). Work stealing provides spatial locality, and tiling avoids further scheduling and data movement overheads. However, work stealing provides poor temporal locality across a sequence of tree traversals because the chance of assigning a node to the same processor across traversals is effectively random. Therefore, as the number of processors increases, the probability of temporal locality decreases and performance suffers. Our solution was to first load balance the initial traversal by dynamically scheduling tiles through a simulation of work stealing. All subsequent traversals then simply reuse the initial schedule. The intuition assumes that (1) the same load balancing applies

across traversals because every node visit generally executes few statements, and (2) by reusing the schedule's assignment of nodes to processors, temporal locality improves.

2. **SIMD Tree Traversals: Input-Dependent Clustering** The performance of SIMD evaluation suffers when parallel tasks diverge in instruction selection. For example, consider extending `H-AG` with vertical boxes and then vectorizing the computation of the widths of all nodes in a tree level. Vertical boxes will invoke a max function while horizontal boxes, a sum. If the distribution of vertical and horizontal is random, the instructions for adjacent nodes will diverge. We observe that nodes that share the same type can be computed over in parallel with little divergence. Our SIMD scheduler therefore clusters such nodes at load-time and, instead of vectorizing the loop over all of the nodes in a level, vectorizes smaller loops over clusters with the guarantee of no divergence. Our full implementation involves the selection of several input attributes to consider when clustering, not just the node type.

Note that while both algorithms optimize the traversal schedule, they do so as a runtime refinement of the overall parallel traversals identified by our synthesizer.

For `H-AG`, we achieved a 6.9X speedup on an 8-core device (9.3X when sequential optimizations are included). For a manual implementation of the SIMD algorithm for one pass of the CSS layout language, we saw a 3.5X speedup on SSE hardware with 4 vector lanes. The SIMD scheduler's load-time clustering cost is amortized over multiple traversals. In the case of the SIMD algorithm, we also measured the improvement in performance-per-Watt: 3.6X. As we expected for vectorization, it is close to the speedup. Finally, our MIMD and SIMD algorithms are complementary. The potential speedup of our approach on commodity hardware features heterogeneous parallelism is thus the product of the individual techniques.

An unexpected result arose for GPUs. GPU architectures suffer from instruction divergence similarly to the subword-SIMD ones we examined. However, in the case of data visualization on GPUs, we achieved significant speedups *without* clustering. We found that while different levels of the tree hold different types of nodes, most nodes on the same level of the tree are the same, such as the points of a time series chart. The SIMT model of GPUs automatically avoids slowdowns when all threads branch in the same direction. Therefore, laying out a tree map on a GPU achieves a 60.6X speedup under the default breadth-first order. The same property does not hold for document layout, which encounters significant divergence under the default ordering and thus benefited from clustering.

1.7 Collaborators and Publications

Work in this thesis was introduced in prior publications (Jones et al., 2009; Meyerovich and Bodík, 2010; Meyerovich et al., 2013) in joint work with wonderful collaborators: Rastislav Bodík, Krste Asanović, Rose Liu, Chris Jones, Todd Mytkowicz, Wolfram Schulte, Matthew Torok, and Eric Atkinson.

Chapter 2

Mechanizing Layout Languages with Extended Attribute Grammars

2.1 Motivation and Approach

This chapter examines the challenges inherent in designing and implementing layout languages and how to use attribute grammars to specify them. We use a running example of a simple layout language and show how our handling of it also applies to common layout language constructs and, more generally, computations over trees. By formalizing layout languages in our extended variant of attribute grammars, we can automate key tasks: checking the language's semantics for errors (Chapter 3), parallelizing it (Chapter 4), and performing aggressive low-level optimizations (Chapter 5). This chapter describes attribute grammars, how we specify layout language constructs with them, and expressive extensions we needed to add to the formalism.

Important Properties for Layout Languages

Layout languages extremely common – by one estimate, there are over 634 million websites live in 2012, with 51 million added that year [1]. Beyond the CSS and HTML languages used for webpage layout (Lie and Bos, 1997), designers also use LaTeX (Knuth and Bibby, 1986) for document layout, D3 (Bostock et al., 2011) for data visualization, and Swing (Eckstein et al., 1998) for GUI layout in addition to more specialized languages such as Markdown (Gruber, 2004) for simpler text formatting within webpages.

Popular layout languages foster designer productivity by providing many domain-specific abstractions. The alternative is analogous to asking a programmer to write in a low-level language such as assembly: designers should not manually specify the position on a canvas and the style for each element. Instead, layout languages resemble constraint systems where designers declare high-level properties. For example, the high-level program `hello world`

[1] http://news.netcraft.com/archives/2012/12/04/december-2012-web-server-survey.html

specifies that the words hello and world should be rendered, and word world should follow line-wrapping rules for its positioning after hello. Layout languages may provide quite complicated constraints – for example, most document layout languages resort to defining their line wrapping rule in a flexible low-level language. Likewise, they may provide many features, such as in the 250+ pages of rules for the CSS language. Adding to the sophistication, many languages support designers adding their own constraints, such as through macros in LaTeX, percentage constraints in CSS, and arbitrary functions in Adobe Flex.

The richness of popular layout languages comes at the cost of complicating their design and implementation:

- **Safe semantics.** Does every input layout have exactly one unique rendering, i.e., is layout deterministic? Are the constraints restricted enough such that an efficient implementation is feasible for low-power devices, big datasets, and fast animation? When a new layout primitive is added, do these properties still hold? We want an automated way to verify such properties.

- **Correct implementation.** As a layout language grows in popularity, it grows in features. Likewise, developers will port it to many platforms and optimize it, and in cases such as CSS, reimplement it from scratch. Does the implementation conform to the intended semantics? Conformance bugs for CSS plague developers, and failures to match LaTeX's semantics have killed multiple attempts to modernize the implementation. We want an automated way to ensure that the implementation matches the specification.

- **Advanced implementation.** Browser layout engines for CSS are currently over 100,000 lines of optimized C++ code. Rich layout languages thus far have resisted parallelization, and improving the speed, memory footprint, debugging support, and other implementation details is difficult for such a quantity of code. We want automation techniques to lower the implementation burden and perform more aggressive optimizations.

Our idea is to declaratively specify layout languages as attribute grammars and automatically compile them into an efficient implementation. At runtime, an instance of layout will be processed through the generated layout engine (Figure 2.1). Our attribute grammar compiler is responsible for checking the semantics of the layout constructs and, by construction, provides a correct implementation. Furthermore, instead of manually optimizing the code for uses of every individual construct, we write generic compiler optimizations. As a similar implementation benefit, we automatically target multiple platforms. For example, we generate a JavaScript layout engine in order to reuse existing browser debuggers, and layout engines in low-level multicore and GPU languages to gain magnitudes of speedups.

Whether attribute grammars are expressive enough for layout languages is unclear. For example, Saraiva and Swierstra (2003) explored expressing fixed HTML table layout, but used a higher-order extension to attribute grammars that challenges optimizations and did

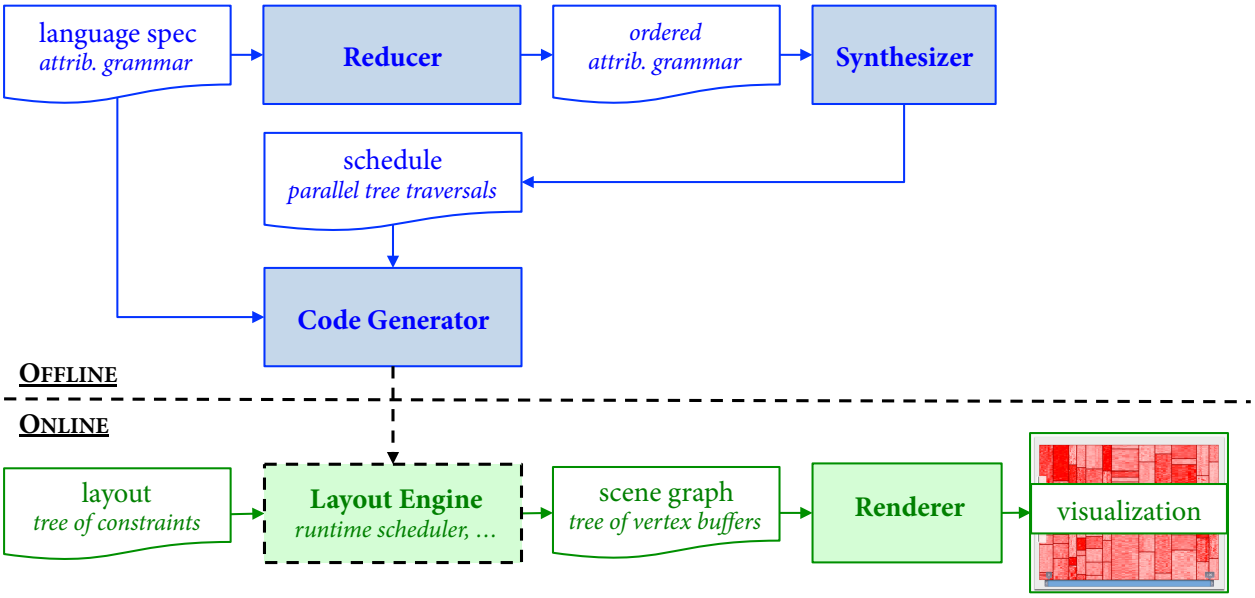


Figure 2.1: **Layout engine architecture.**

not report supporting automatic table layout. Expressive extensions are also required but we show how to restrict them enough to support reductions to typical attribute grammar machinery (Figure 2.1). The remainder of this chapter introduces the high-level attribute grammar formalism, how to specify layout languages using it, and an intuition for the reduction into a lower-level formalism.

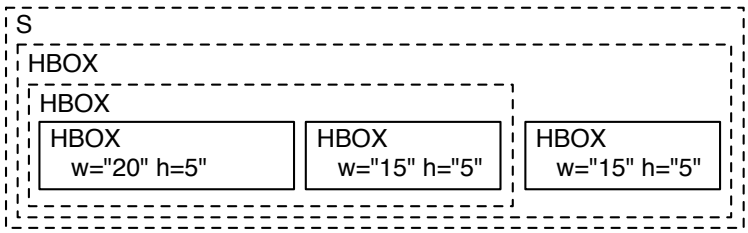
2.2 Background: Layout with Classical Attribute Grammar

We start by declaratively specifying a simple layout as an attribute grammar and then show two evaluation strategies for automatically implementing it.

Attribute Grammars

Consider programming the nested horizontal boxes shown in Figure 2.2a. As input, a webpage designer provides a tree with constraints (Figure 2.2b). Only some node attribute values are provided: in this case, only the widths and heights of leaf nodes (attributes `w` and `h`). By using an `HBox` node, the designer further specifies that the boxes will be placed side-by-side. The layout engine must solve for all remaining x, y, width, and height attributes.

Figure 2.2c declaratively specifies the layout language of horizontal boxes, `H-AG`, as an attribute grammar (Kastens, 1980; Meyerovich and Bodík, 2010; Saraiva and Swierstra, 2003). First, the specification defines the set of well-formed input trees as the derivations



(a) **Input tree.** Only some of the x, y, w, and h attributes are specified.

```
1 <S>
2   <HBox name=child>
3     <HBox name=left>
4       <HBox name=left w=20 h=5/>
5       <HBox name=right w=15 h=5/>
6     </HBox>
7     <HBox name=right w=15 h=5/>
8   </HBox>
9 </S>
```

(b) **Textual encoding of input tree.**

$S \rightarrow HBOX$
$\{ \text{HBOX.x} = 0; \text{HBOX.y} = 0 \}$

$HBOX \rightarrow \epsilon$
$\{ \text{HBOX.w} = \text{input}_w(); \text{HBOX.h} = \text{input}_h() \}$

$HBOX_0 \rightarrow HBOX_1\ HBOX_2$
$\{ \text{HBOX}_1\text{.x} = \text{HBOX}_0\text{.x};$
$\text{HBOX}_2\text{.x} = \text{HBOX}_0\text{.x} + \text{HBOX}_1\text{.w};$
$\text{HBOX}_1\text{.y} = \text{HBOX}_0\text{.y};$
$\text{HBOX}_2\text{.y} = \text{HBOX}_0\text{.y};$
$\text{HBOX}_0\text{.h} = \max(\text{HBOX}_1\text{.h}, \text{HBOX}_2\text{.h});$
$\text{HBOX}_0\text{.w} = \text{HBOX}_1\text{.w} + \text{HBOX}_2\text{.w} \}$

(c) **Attribute grammar for a language of horizontal boxes.**

$AG \rightarrow (Prod\ \{\ Stmnt?\ \})^*$

$Prod \rightarrow V \rightarrow V^*$

$Stmnt \rightarrow Attrib = id(Attrib^*) \mid Attrib = n \mid Stmnt\ ;\ Stmnt$

$Attrib \rightarrow id.id$

(d) **Language of attribute grammars.**

Figure 2.2: For a language of horizontal boxes: (a) visualized solution, (b) input tree to solve, and (c) attribute grammar specifying the layout language. Specification language of attribute grammars shown in (c). The language of attribute grammars is shown in (d).

of a context-free grammar. We use the standard notation. In this case, a document is an unbalanced binary tree of arbitrary depth where the root node is labeled `S` and intermediate nodes are labeled `HBOX`. Second, the specification defines semantic functions that relate attributes associated with each node. For example, the width of an intermediate horizontal node is the sum of its children widths. Likewise, the width of a leaf node is provided by the user, which is encoded by the nullary function call $input_w()$.

The specification does not define the evaluation order. For example, the specification does not state whether to compute a node's width before its height. Likewise, while our optimized approach computes the attributes over a sequence of tree traversals, the specification does not define the sequence of tree traversals. Leaving the evaluation order unspecified provides freedom for our compilers to make parallel scheduling decisions. Irrespective of whatever evaluation order is ultimately used to solve for the attribute values, the statements define constraints that must hold over the computed result. Attribute grammars can therefore be thought of as a single assignment language where attributes are dataflow variables.

The syntax of attribute grammars is defined in Figure 2.2d. In addition to defining the context free grammar, it includes terms for single-assignment constraints over attributes of nodes in a production. Our example uses the following encoding. Semantic functions are pure and left uninterpreted: for example, we encode the addition of widths as "HBOX_0.w = f(HBOX_1.w, HBOX_2.w)". Our program analysis techniques do not need to know the contents of the function, just that the output of a call depends purely on the inputs. For the same reason, we encode constant values as nullary function calls.

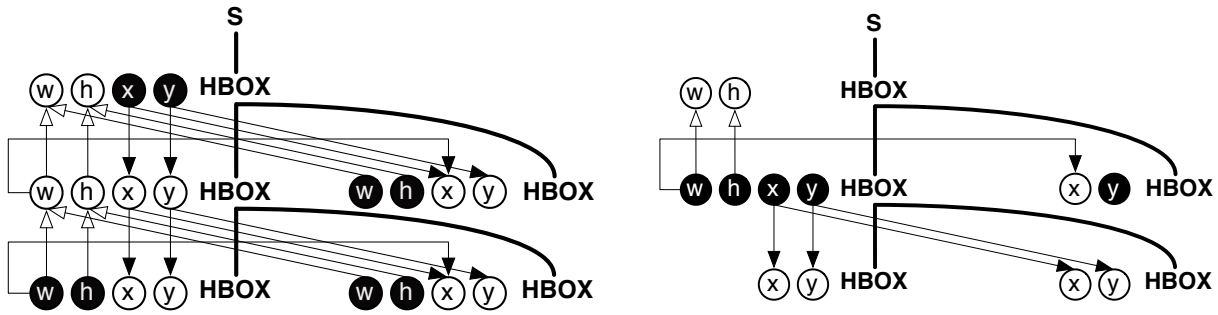
To specify grammars more complicated than `H-AG`, we describe two expressive extensions. This chapter add extensions to the functional specification language that desugar into this formalism (Section 2.3). To control the evaluation order, Chapters 3 and 4 introduce a complementary scheduling language.

Dynamic Data Dependency Graphs and Dynamic Evaluation

A simple and classic evaluation strategy is to *dynamically* compute over the attributes of a tree. The evaluator tracks the data dependencies between instances of attributes. The dynamic evaluation strategy is too slow for the cases presented herein, but it introduces the key concepts of dynamic data dependencies, the dynamic semantics of attributes grammars, and the corresponding interpreter.

An instance of a document corresponds to the dependency graph shown in Figure 2.3a. Each attribute of a tree node is either a source, meaning its value can be computed based on other known values, or it cannot be evaluated until other attribute values are known. It is a dynamic dependency graph in that each data dependency in the static code may be instantiated as multiple data dependencies given the input tree at runtime.

The dynamic data dependency graph leads to a simple semantics and interpreter design. The graph corresponds to a system of equations where edges link instance variables. For example, static code $\texttt{HBOX}_2.x = \texttt{HBOX}_0.x + \texttt{HBOX}_1.w$ instantiates twice for the example shown in Figure 2.3a: once for each x attribute with an incoming elbow connector. The values



(a) Initial dependencies.

(b) After first round of evaluation.

Figure 2.3: **Dynamic data dependency graphs and evaluation**. Shown for the constraint tree in Figure 2.2 (a). Circles denote attributes, with black circles denoting attributes whose dependencies are all resolved, such as input() invocations. Thin lines depict data dependencies and thick lines show production derivations. Chart (b) shows the dependency graph resulting from evaluating all source nodes and treating them as resolved.

```
1  input: G = (V, E)
2  output: Map
3  Map ← ∅
4  E′ ← E
5  V′ ← V
6  for a ∈ V′ where ∄(n, a) ∈ E′:
7     Map ← Map ∪ {a → eval(a)}
8     V′ ← V′ − {a}
9     E′ ← E′ − ({a} × V)
10 repeat until E′ = ∅
11 return Map
```

Figure 2.4: **Dynamic attribute grammar evaluator.** It selects attributes in a safe order by dynamically removing dependency edges as they are resolved.

of both xs are constrained by distinct instances of the above constraints. Note that if the dependency graph is a directed acyclic graph and each attribute appears on the left-hand side of exactly one equality statement (a *dataflow variable*), there is exactly one solution for every attribute.

A simple interpreter design is to iteratively remove nodes with no incoming data dependencies (Figure 2.4). The algorithm iteratively finds an attribute whose dependencies have all been previously resolved, evaluates the attribute, and repeats. If the input graph is a directed acyclic graph, this procedure is guaranteed to terminate. The insight is that if a directed acyclic graph has at least one fringe node, the loop removes them, and removing these nodes yields a smaller directed acyclic graph. As an optimization, a topological traversal is one such traversal order.

<Top> → *<Top>** | *<Trait>* | *<Class>* | *<Interface>*

<Interface> → **interface** *id* *<AttribDecl>**

<AttribDecl> → **var** *id* : *id*
| **input** *id* : *id*
| **input** *id* : **?** *id*
| **input** *id* : *id* = *val*

<Trait> → **trait** *id* *<Child>** *<AttribDecl>**<TopConstraint>**

<Class> → **class** *id* (*id**) : *id* *<Child>** *<AttribDecl>** *<TopConstraint>**

<Child> → *id* : (*id* | [*id*])

<TopConstraint> → *<Constraint>*
| **loop** *id* (*<Constraint>* | *<Lhs>* := **fold** *<Expr>* .. *<Expr>*)*

<Constraint> → *<Lhs>* := *<Expr>*

<Expr> → *<Rhs>* | *unop* *<Expr>* | *<Expr>* *binop* *<Expr>*

<Rhs> → *<Lhs>* | **self** *<Suffix>* . *id* | *id* *<Suffix>* . *id*

<Lhs> → *id* | *id* . *id*

<Suffix> → **$i** | **$-** | **$$**

Figure 2.5: **EBNF syntax for key forms of the functional specification language in Section 2.3.** We omit semicolons and other decorations; see the examples for more detailed forms.

The dynamic evaluation strategy provides an explanation for the natural semantics, but several challenges remain. First, runtime manipulation of a dynamic dependency graph introduces high overheads because every dynamic dependency edge must be manipulated at runtime. Second, it is unsafe. For example, a cycle in the dependency graph causes the above evaluation strategy to get stuck. Designers can build layout widgets that, depending on how they are invoked, fail to display!

2.3 Desugaring Loops and Other Modern Constructs

The attribute grammar formalism was invented for describing semantics (Knuth, 1990) and before many modern linguistic constructs became mainstream. As a result, we had to design extensions for improved expressiveness and maintainability. Our extensions exploit concepts from structured, object-oriented, and functional programming. Other language designers

have built such extensions as well (Koskimies, 1991; Vogt et al., 1989). Our challenge was to design expressive extensions that are restricted enough to to facilitate effective parallelization and not overly complicate implementation. The following subsections documents each extension and the motivation behind it, and leaves performance optimizations to subsequent chapters.

Our key insight was the realization that pre- and post-processing supports desugaring extended attribute grammars into the basic attribute grammar notation. Tools then operate at the most appropriate stage, such as our scheduler on the basic attribute grammar representation. Likewise, our code generators take a generated schedule and relate it back to a representation from early in the preprocessing stage. Many of our features are implemented as explicit compiler stages, but over time, we found that declarative tree rewriting systems such as ANTLR (Parr and Quong, 1995) and OMeta (Warth and Piumarta, 2007) support automating individual stages. Figure 2.5 shows the syntax of our functional specification language.

Interfaces for Encoding Tree Grammars

Our first extension targeted how to specify tree structure. Attribute grammars use tree grammars for defining input trees, so improving the abstraction capabilities of tree grammars also aids the ability to structure attribute grammars. In particular, we found the need to support abstracting over similar types of non-terminals. Our solution is to provide a notion of classes and interfaces. If class HBox and VBox both implement interface BoxI, we can write one production "X → `BoxI`" rather than a production for every variant: "X → `HBox`", "X → `VBox`", etc. The extension is expressible with attribute grammars and thereby reduces implementation requirements; it is still important enough, however, that it merits deeper compiler support.

Consider the code duplication performed when extending `H-AG` with vertical boxes. The children of a HBox could be a horizontal box or a vertical box, and the same for the children of a vertical box. Figure 2.6a shows that `H-AG`'s specification grows from 3 productions to 11 when defining the valid combinations of nodes and their children. The example highlights that basic attribute grammars cannot abstract over node types. Adding a new box type requires modifying all previous box classes, and in the presence multiple children, extension suffers exponential costs.

To abstract over node types, we introduced the notion of classes and interfaces (Figure 2.6b). Classes are similar to the productions of an attribute grammar: the class name specifies the production's left-hand side non-terminal and the children block specifies the production's right-hand side. Unlike attribute grammars, an interface name is used for the right-hand side rather than the class name. HBox and VBox implement interface BoxI, so any class specified to have a BoxI child can have a HBox or VBox child within the concrete tree.

Classes and interfaces are formally equivalent to tree grammars in the sense of a 1-to-1 correspondence between the trees that are described by both. First, a tree grammar can be expressed with classes and interfaces by treating all productions with the same left-hand-side

$$
\begin{aligned}
S &\rightarrow HBOX \mid VBOX \\
HBOX &\rightarrow \epsilon \\
HBOX_0 &\rightarrow HBOX_1\ HBOX_2 \\
HBOX_0 &\rightarrow VBOX_1\ HBOX_2 \\
HBOX_0 &\rightarrow HBOX_1\ VBOX_2 \\
HBOX_0 &\rightarrow VBOX_1\ VBOX_2 \\
VBOX &\rightarrow \epsilon \\
VBOX_0 &\rightarrow HBOX_1\ HBOX_2 \\
VBOX_0 &\rightarrow VBOX_1\ HBOX_2 \\
VBOX_0 &\rightarrow HBOX_1\ VBOX_2 \\
VBOX_0 &\rightarrow VBOX_1\ VBOX_2
\end{aligned}
$$

(a) **Canonical attribute grammar.**

```
1  interface BoxI { }
2  class HBoxLeaf : BoxI { }
3  class HBoxBinary : BoxI {
4    children {
5      left: BoxI;
6      right: BoxI;
7    }
8  }
9  class VBoxLeaf : BoxI { }
10 class VBoxBinary : BoxI {
11   children {
12     left: Box;
13     right: Box;
14   }
15 }
```

(b) **Interface sugar.**

$$
\begin{aligned}
S &\rightarrow BOX \\
BOX &\rightarrow HBOX \mid VBOX \\
HBOX &\rightarrow \epsilon \\
HBOX_0 &\rightarrow BOX_1\ BOX_2 \\
VBOX &\rightarrow \epsilon \\
VBOX_0 &\rightarrow BOX_1\ BOX_2
\end{aligned}
$$

(c) **Interface encoding.**

Figure 2.6: **Interfaces for tree grammars**. Subfigures show manually encoding multiple production right-hand sides, an encoding that uses a Box non-terminal for indirection, and the high-level encoding using interfaces and classes.

```
1 {"class": "HBox",
2  "children": {
3    "left": {
4      "class": "HBox",
5      "children": {
6        "left":  {"class": "HBox", "w": 20, "h": 5},
7        "right": {"class": "HBox", "w": 15, "h": 5}}},
8    "right": {
9      {"class": "HBox", "w": 15, "h": 5}}}}
```

Figure 2.7: **Input tree as a graph with labeled nodes and edges**. Specified in the JSON notation.

non-terminal as different classes belonging to the same interface. In the other direction, each interface can be expressed as a production that derives the classes, and the classes expand into productions. Figures 2.6b and 2.6c demonstrate the correspondence for `H-AG`. The induced implementation requirements are therefore slight in the sense that the construct is sugar for a pattern in attribute grammars.

We depart from the correspondence for the encoding of trees in two ways. First, we represent input as a tree with labeled nodes and edges. Node labels denote the class and edge labels specify child bindings. Figure 2.8 uses the JSON format common to dynamic languages for an instance of a tree in `H-AG`. By naming children, such as left and right, we eliminate sensitivity to their lexical order within a code block. With order sensitivity, adding a middle child center would needlessly require refactoring references to the repositioned element right. Likewise, reordering children in the input data does not require refactoring the attribute grammar.

Our second departure from the canonical attribute grammar encoding optimizes the data representation by eliding intermediate interface nodes. The reduction to attribute grammars suggests adding a new non-terminal node for each interface (Figure 2.6c), but doing so in the data representation doubles the number of nodes in the concrete tree. Making the interface pattern a language construct with compiler support eliminates associated costs, such as cutting file size for runtime parsing of big data visualizations.

Interfaces for Attributes and Information Hiding.

Node interfaces also support lightweight specification annotations for different types of attributes, and coupled with the overall interface construct, it supports defining relationships between attributes across different classes.

Each static attribute is annotated with its assignment type and its embedded value type:

- **Assignment types.** The assignment type denotes whether the input tree defines the value, such as in input w, or whether the attribute grammar defines it, as in var x. Assignments to an input type are illegal, and multiple assignments to a variable type are also illegal. A simple type checker detects these violations.

```
1  interface BoxI {
2    var x : float;
3  }
4  class HBoxLeaf : BoxI {
5    attributes {
6      var y : int;
7      input w : ? int;
8      input h : int = 10;
9    }
10 }
```

Figure 2.8: **Input tree as graph with labeled nodes and edges**. Specified in the JSON notation.

If an input tree fails to provide an input attribute, a runtime error will be thrown. To still provide an interpretation of such trees, we also support annotating input attribute declarations with "?", which enables inspection at runtime through through the functions `maybeReady` :: () → `boolean` and `maybeValue` :: () → α. Alternatively, for the common scenario of using a fixed default value, a default value can instead be defined as in input h : int = 10. If the input tree does not provide the value, the default value will be automatically substituted.

Without our extensions, attribute grammars can still encode input attributes. First, semantic functions with no parameters can encode attributes with no dependencies. Alternatively, for finite domains, the set of tree grammar productions can expand to include attribute nodes. The second encoding more faithfully describes our implementation approach because, like our system, it feeds into an automatic tree parser generator. For each tree node, our generated parser scans for the expected set of input attributes.

- **Value types.** The system also supports type annotations used for embeddings. Generated code typically compiles as part of a project in a more static language, such as C++, which may require a static typing discipline. The annotations can be user-defined, such as OpenGL's *vertex buffer object* VBO, which is not defined within our system.

  Our analyzer ignores the value type annotations such as x : float and y : int, and passes them along through the low-level code generator. The embedded design simplifies implementation because value type checking is then performed by the host language's compiler.

In practice, we use attribute definitions in interfaces for information hiding across classes and lightweight specification of relationships between similar classes. An attribute declared inside of a class is *local* to constraints in the class: only the class's constraints can read or write to the attribute. Conversely, declaring a *var* inside of an interface hints that it is meant to be reused by outside classes, i.e., part of a tree traversal.

```
1  trait Rectangle {
2    attributes { render : int; }
3    actions { render := paintRect(x,y,w,h, "black"); }
4  }
5  class HBox(Rectangle) : BoxI { ... }
```

Figure 2.9: **Trait construct**. Adds shared rendering code to the HBox class.

```
1  interface BoxI {
2    var w : int;
3    var h : int;
4    var right : int;
5    var bottom : int;
6  }
7  class HBox : BoxI {
8    children {
9      childs : [ BoxI ]
10   }
11   actions {
12       loop childs {
13         w := fold 0 .. self$ .w + childs$i.w;
14         h := fold 0 .. max(self$ .h, childs$i.h)
15         childs.right := fold x .. childs$ .right + childs$i.w;
16         childs.bottom := fold y .. childs$ .bottom + childs$i.h;
17       }
18   }
19 }
```

Figure 2.10: **Input tree as graph with labeled nodes and edges**. Specified in the JSON notation.

Traits: Reusing Cross-Cutting Code

As with many object systems, we support a trait construct for cross-cutting code that should be shared across classes. It statically expands like a macro, providing no formal expressive power. For example, Figure 2.9 defines how to render a rectangle given several attributes and then adds that functionality to class HBox. If the language was extended with class VBox, the class definition of VBox could also use trait Rectangle.

Loops

We extend our language with declarative loops for computing attributes of a statistically unbounded number of child nodes. They are an expressive extension over the uniform recurrence equations (Karp et al., 1967). One loop in our extension may correspond a collection of loops in a traditional functional or imperative programming language.

The loop construct, loop, specifies a block of loop body statements. It acts over a sequence of nodes declared with the same interface, such as childs : [BoxI] in Figure 2.10. The looping order is restricted to forward iteration, though our approach generalizes to other loop orders.

A statement in a loop body will execute for each element of the list. For example, the following statement assigns the attribute w the sum of the children widths:

$$w := \texttt{fold}\ 0\ ..\ self\$ - .w + childs\$i.w$$

Similar to array index notation, the suffix on right-hand side variable names for loop statements provide a restricted form of relative indexing. In particular:

- $i: the "current" loop step
- $-: the previous loop step
- $$: the last loop step

Use of suffix "$-" in a fold can be thought of as an accumulator in functional programming.

One loop statement can refer to the accumulator of another, which fold statements in most languages do not support. For example, two loop counters can be intertwined:

```
1 loop childs {
2   childs.counter1 := fold 0 .. childs$ .counter2 + 1;
3 }
4 loop childs {
5   childs.counter2 := fold 0 .. childs$ .counter1 + 1;
6 }
```

The programmer does not manually order the statements. For the above loops, our system would infer that their imperative implementation is just one loop that fuses them together. The incorrect alternative of implementing the declarations as a different imperative loop for each would lead to unfulfilled data dependencies. Likewise, if a set of loop statements must be implemented using a sequence of loops, the programmer has the liberty of defining them in one loop nest and relying upon the compiler to disentangle them.

We reduce scheduling loops to scheduling basic attribute grammars. For a restricted language of relative indices, we are able to schedule several unrolled loop steps and generalize to the rest of the schedule. Section 3.3 discusses this in more detail.

The declarative nature of the loop construct provides two key benefits. First, coupled with the restricted indexing language, underspecification of the statement order provides freedom for automatic parallelization (Section 3.3). Second, it allows programmers to choose how to structure the program. For example, separating loop statements as above might improve legibility if they are for two different purposes, but as the computation is more intertwined, the programmer has the freedom to choose the following formulation instead:

```
1 loop childs {
2   childs.counter2 := fold 0 .. childs$ .counter1 + 1;
3   childs.counter1 := fold 0 .. childs$ .counter2 + 1;
4 }
```

The formulation brings the two statements together and changes their lexical order. Our language guarantees that such a refactoring does not change the semantic meaning of the code.

Embedding Functional Rendering Calls

We designed our system for interaction with other tools and languages. A key ability is to invoke externally-defined functions, such as max() of Figure 2.10 for the maximum of two numbers and paintRect() of Figure 2.9 to draw a rectangle on the screen. Our system compiles attribute grammars to run in various hosts, such as JavaScript or OpenCL, and any function in scope to the generated code may therefore be called.

Functions can be safely invoked as long as they provide a *pure* interface. In particular, the returned output should only depend on the inputs. Likewise, functions should be reentrant for use in automatic parallelization In the case of embedding in statically checked languages, the host's static checker is responsible for checking usage.

2.4 Evaluation: Mechanized Layout Features

We specified many common layout language features with our extended form of attribute grammars. Most examples were written with few, if any, modifications to the generated code. This experience shows that our restricted form of attribute grammars are a viable formalism for specifying layout languages. The following subsections present highlights from our case studies in document layout and data visualization; the appendix contains the full specifications.

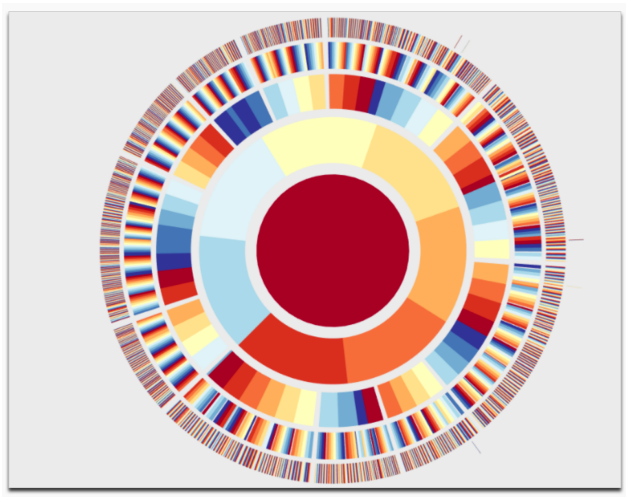
Rendering and Interaction

We found several rendering patterns to be important for many visualizations. A library of functional graphics primitives, such as function paintRect() in Figure 2.9, sufficiently augmented our attribute grammar language.

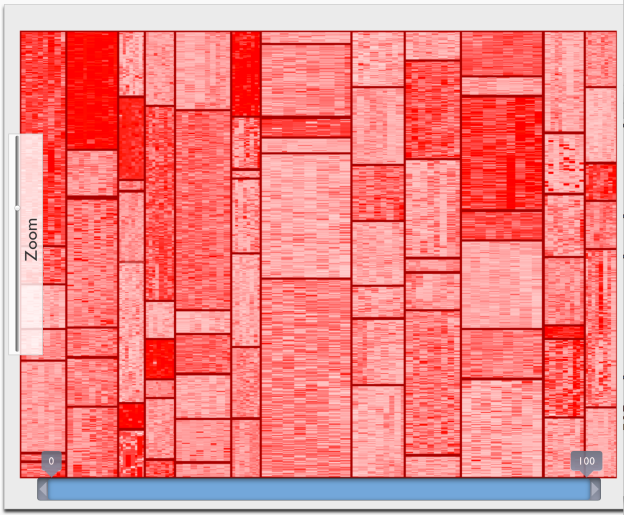
- **2D and 3D.** Calls to rendering functions provide coordinates in 2D or 3D space. The use of different coordinate spaces does not impact the attribute grammar formalism.
- **Color.** Our functional graphics primitives take an RGBA value as input, which enables the attribute grammar to control the hue, luminosity, and opacity.
- **Linked view**. Multiple renderable objects can be associated with one node, which we can use for providing different views of the same data:

  ```
  1 render := Circle(x,y,r) + Circle(offsetX + abs(x), offsetY + abs(y), r);
  ```

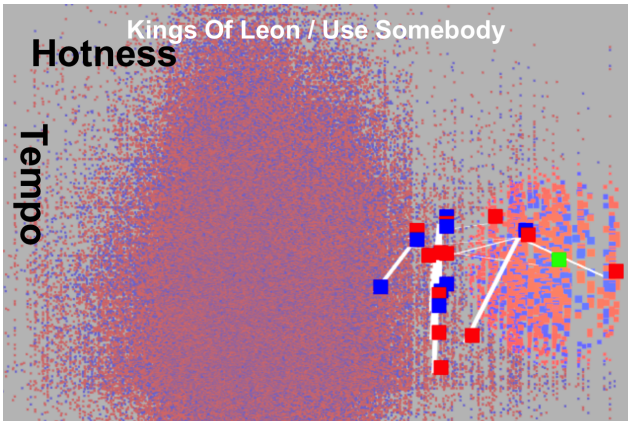
  Statistical analysis software may use this feature for layouts such as scatterplot matrices.
- **Zooming.** We can use the same multiple representation capability for a live zoomed out view ("picture-in-picture"):



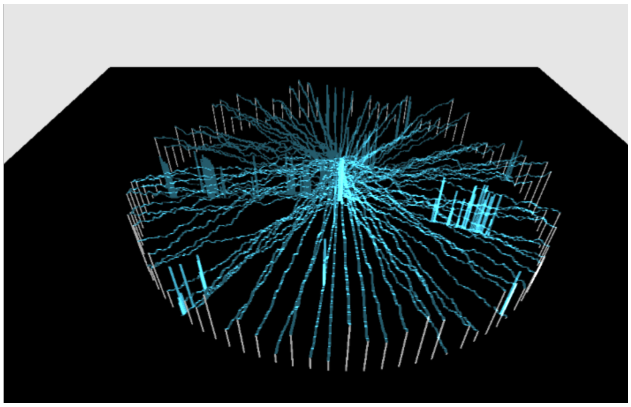
(a) **Sunburst**



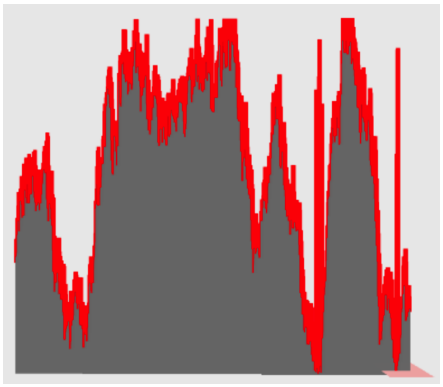
(b) **Treemap**



(c) **Linked Scatter**

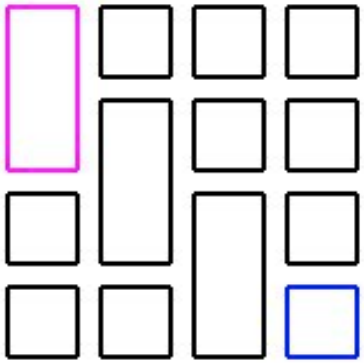


(d) **3D Multiple Time Series**



(e) **Line Graph**

Figure 2.11: **Visualization screenshots.** All except are interactive or animated. Each one was declaratively specified with our extended form of attribute grammars and automatically parallelized. Labels describe whether GPU or multicore code generation was used.



(a) **HTML Tables** Grid-based

(b) **CSS** Flow-based

Figure 2.12: **Document layout screenshots.**

```
render :=
  Circle(x, y, radius)
   + Circle(xFrame + x*zoom, yFrame + y*zoom, radius *zoom);
```

We found it useful for heatmaps with many dimensions.

- **Visibility toggles.** Our system support conditional expressions, which enables controlling whether to render an object. For example, a boolean input attribute can control whether to show a circle: `render := isOn ? Circle(0,0,10) : 0;`
- **Alternative representations.** Conditional expressions also enable choosing between multiple representations, not just on/off visibility:

```
render :=
  isOff ? 0
    : mouseHover ? CircleOutline(0,0,10)
    : Circle(0,0,10,5) ;
```

Non-Euclidean Layouts: Sunburst Diagram

Visualizations often require non-Euclidean layouts, such as the polar layout for a sunburst diagram. Instead of propagating and computing over Euclidean values such as the x and y coordinates of `H-AG`, the visualization can use its own coordinate system. In a sunburst diagram (Figure 2.11a), a node should be rendered far from the center of the chart if its level is high. Our implementation defines each node's radius as a function of its parent's radius. Likewise, the center of visualization propagates from parent to child, with the root node representing the center:

```
class Radial : Node {
  ...
      loop child {
          child.parentTotR := parentTotR + r;

          child.rootCenterX := rootCenterX;
          child.rootCenterY := rootCenterY;
      }
      ... Arc(rootCenterX, rootCenterY, show * (parentTotR + r), ...);
}
```

The full example is available in Appendix A.1.

Charting: Line Graphs and Scatterplots

We specified several types of charts with attribute grammars. For example, Figure 2.11c depicts an X/Y scatterplot, and Figure 2.11e depicts a line graph. We represent every data point as a leaf node in the tree. Tree traversals will compute details such as the X and Y ranges of a dataset, which facilitates features such as normalization and centering.

Time series charts used several of the above techniques. First, multiple time series data should often be represented at the same time, such as for a server farm, the output of each

server as the days pass. Figure 2.11d depicts one such multiple time series chart. Our approach was to represent each line as an intermediate node:

```
1  class Root : Root I {
2    children {
3       lines : [ LineI ];
4    }
5  }
6  class Line : LineI {
7    children {
8      points: [ PointI ]
9    }
10 }
11 class Point : PointI { }
```

Second, we found the above (Section 2.4) rendering features such as zooming, panning, and 3D representations to be important for visualizing big time series datasets.

Animation and Interaction: Treemap

We declaratively encoded various animation effects with attribute grammars. For example, the fisheye effect enlarges the size of an element the closer the mouse draws near to it. Our core pattern is to encode time-varying values such as the mouse position as input attributes and rerun the layout solver whenever the inputs change.

Beyond human interaction, we also support reaction to time. For example, for the treemap shown in Figure 2.11b, users may change the dataset shown. Instead of immediately showing the new dataset, we introduce a tween attribute that an animation increments over time from 0 up to 1. The treemap interpolates the layout position based on the time, which yields a smooth transition for each data point:

```
1  class Point : PointI {
2    attributes {
3      input startW : int;
4      input endW : int;
5      var w : float;
6      var tween : float;
7    }
8    actions {
9      ...
10     w := startW * tween + endW * (1.0f  - tween);
11     render := paintRect(x, y, w, h, ...
```

Visualizations like the treemap require recompilation of most of the attributes for such animations, which can become a bottleneck and thus benefits from the acceleration provided by our tool.

Grid-based: Tables

We now examine one of our most difficult case studies: specifying CSS table layout constructs (Lie and Bos, 1997). Tables appear in most rich document layout languages, e.g., CSS and LaTeX, and are an instance of *grid-based layout*, which is popular for user interfaces

and data tables. In conversations with commercial browser developers, we found that the proposed standards for the layout language features were reverse-engineered from earlier implementations. Furthermore, at the time of writing, two competing standards were proposed for table layout in CSS, and with unclear notions of completeness nor differences.

We had to address several challenges to specify table layout:

- **Dynamic data structure.** Layout constraints guide the mapping from a cell node to its column index. The computed result of attribute constraints therefore determines the underlying graph structure rather than being provided as part of the input.
- **Computing over a DAG rather than a tree.** Each cell of a table has two parent nodes: its row and its column. Attribute grammars are more typically designed for computations over trees, where each node has at most one parent. Static reasoning about dependencies must take into account this more general structure.
- **Non-linear constraints.** Static attribute grammars linearly bound the computation size in terms of the number of attribute instances. A more iterative process is instead used to compute dimensions for CSS's automatic table layout algorithm.

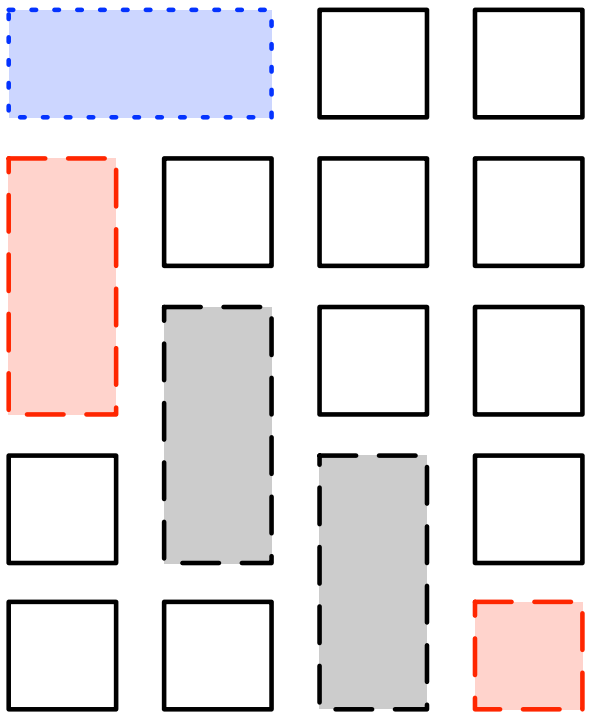
Ultimately, we wrote table-specific code in the specification (see above) and the runtime, but no table-specific code in our scheduler nor code generator. For an example of logic in the specification, the specification constructs the grid data structure by manipulating functional lists rather than just numbers. Likewise, to ensure a column's computations over its cells are scheduled after the grid is constructed, we included this dependency in the specification.

Our runtime edits were to use a breadth-first traversal for traversing a table and, to lookup the children of a column, search table rows for cells with the corresponding column number attribute. We did not have to add table-specific code into the synthesizer (the offline scheduling analysis) nor the code generator.

We address each problem in turn.

Dynamic data structure.

Figure 2.13 illustrates why the mapping from table cells to table column is dynamically computed. The placement of a cell is complicated by preceding cells that span multiple rows ("rowspan=n") and columns ("colspan=n"). Ultimately, the cell must be placed in the first column such that an earlier cell in a top-down, left-to-right ordering does not overlap it. The figure illustrates two important cases. First, the second cell of the first row is placed in the third column because its left sibling spans two rows: a cell's column is a function of the rowSpan attributes of its siblings to the left. The second case is shown for the bottom right cell. Even though it is the third cell of its row in the parse tree, it is not placed in the third column. The reason is that the red dashed rectangular cell in the second row transitively impacts the placement of the cells after it. The colSpan attributes of cells in rows above a cell therefore further determine its column.



(a) **Table Rendering**

```
<table>
  <row>
    <cell rowSpan="2"></cell>
    <cell></cell>
    <cell></cell>
  </row>
  <row>
    <cell colSpan="2"></cell>
    <cell></cell>
    <cell></cell>
  </row>
  <row>
    <cell colSpan="2"></cell>
    <cell></cell>
    <cell></cell>
  </row>
  <row>
    <cell></cell>
    <cell colSpan="2"></cell>
    <cell></cell>
  </row>
  <row>
    <cell></cell>
    <cell></cell>
    <cell></cell>
  </row>
</table>
```

(b) **Textual encoding of input tree.**

Figure 2.13: **Document layout screenshots.**

Our specification loops over the rows to perform functional updates to the column assignments. For each row, it computes what columns its cells are placed in as a function of the list of columns that are still occupied by preceding cells. The next row is given the columns that are occupied after adding cells on the current row, etc. Our specification of this behavior is interesting in that it is just calls to functional list manipulation methods written in our host language:

```
class TableBox
  ...
  loop rows {
    rows.colAssignment :=
      fold
        emptyColumnList(colCount)
        ..
        columnsAppendRow(
          rows$ .colAssignment,
          rows$i.cells,
          rows$i.rowNum);
```

The columnsAppendRow function computes the column position during placement, so subsequent reads can look it up through another list manipulation function.

A column computes the x coordinates for each cell, but column cells are not known

```
1 Schedule {
2   Col.childs[i].relX < Col.cellsready
3   Col.childs[i].absX < Col.cellsready
```

(a) **Surface Syntax** (Proposed)

```
1 Schedule {
2   asserta(assignment(col, self, childs_relx_step, self, cellsready)),
3   asserta(assignment(col, self, childs_absx_step, self, cellsready))
```

(b) **Low-level constraint**

Figure 2.14: **Specifying dynamic dependencies.**

before the last columnsAppendRow() call. To ensure a column computes over its cells after the mapping occurs, we explicitly declare the dynamic data dependency in the specification. First, the grid is stored in an attribute, so we simply propagate the grid to all the table nodes as an attribute (cellsready). We then state the implicit data dependency (Figure 2.14). The scheduler now knows to run column computations over cells only after the cellsready is computed. Currently, we directly specify the constraints in terms of the desugared grammar (Figure 2.14b), which might be directly generated from surface syntax (Figure 2.14a).

Computing over a DAG

Computing over a table means computing over a DAG, not a tree: a cell has both a row and a column as its parents. This impacts both our runtime and our specification strategy. Demonstrating the flexibility of attribute grammars, we did not have to modify the scheduler nor the code generator. Instead, we modified the runtime and the specification.

We modified the runtime to generalize an important invariant from tree traversals to DAG traversals. In a top-down traversal of a tree, a node's parent is visited before the node itself. A valid implementation of a top-down traversal for trees is depth first. However, consider a depth first traversal of a parse tree of the following table:

```
1 <table>
2   <row>
3     <cell></cell>
4   </row>
5   <column></column
6 </table>
```

The depth-first traversal would visit the table, the row, the cell, and then the column. The cell is visited before its parent column!

Our modification was simple: we edited the runtime to visit the nodes of a table with a breadth first subtraversal. We kept the overall document traversal as depth-first for performance reasons. Extending the set of traversal type primitives of the next chapter to include breadth-first traversals would allow declaratively specifying this choice as part of the schedule specification rather than having to manipulate the schedule implementation. The next chapter discusses subtraversals in greater detail.

We also modified the specification to pass our attribute grammar static checker. The changes enables relaxing the scheduler's obligation to guarantee that visiting a cell's parent row and column would set all the attributes needed by the cell (unambiguous) and without conflicting with each other. For example, a column defines the relX attribute of its child cell, and a row, its relY. By default, our checker would rightfully reject such a specification because, if a cell has only one parent, only one of those attributes would be set.

We extended the specification language for instructing the scheduler that external code defines certain attributes:

```
1  class Col : ColI {
2    phantom {
3      childs.relY;
4      childs.absY;
5  ...
6  class Row : RowI {
7    phantom {
8      childs.relX;
9      childs.absX;
10 ...
```

The scheduler now assumes that the external code provides definitions for a column's childs.relY and childs.absY and a row's childs.relX and childs.absX. Unimportant to the synthesizer, the definitions just happen to come from elsewhere in the same specification, such as class Row defining the phantom attributes not set by Column.

Non-linear constraints

The table specification defines a dynamically determined number of loops over a table's column to determine column widths. Such dynamism is beyond the pure static attribute grammar formalism, but our foreign function interface sufficed while still allowing overall specification and scheduling through attribute grammars.

Flow-based: CSS Box Model

Document layout languages generally feature a *flow-based* layout model where the position of one element is largely a function of the previous one. For example, line wrapping places one word after another in a paragraph, and a column will stack one paragraph after another. However, ambiguity quickly arises once constraints are added to such systems. We found that, before being able to address our interest in parallelizing the CSS language, that creating a functional specification of it was already a challenge to itself. This section focuses on the ability to express the CSS specification, and defers discussion of correctness of functional and parallel behavior to Chapter 3.

Challenging specification, the CSS standard provides only a few explicit formulas such as min(max(intrinsMinWidth,maxWidth),intrinsPrefWidth) for the shrink-to-fit calculation. It generally does not fully define the intrinsic dimensions to plug into the formula. We incorporated what we found, and for the rest, spent significant time reengineering the semantics by examining

the standard and experimenting with existing browsers. While it is unclear how to evaluate faithfulness, we encoded enough features to render a resemblance of the Wikipedia main page (Figure 2.12b) and a popular blog.

Our attribute grammar describes the layout solving features of the informally written CSS 2.1 standard. It also includes automatic table layout, which was only more completely defined in later CSS standards. It does not include preprocessing steps, such as the CSS cascade that annotates the HTML tree with attributes, nor anonymous content generation, which normalizes the annotated tree to guarantee that spans of sibling nodes are homogeneous. The former is largely a combination of a simple extension to regular expressions and prioritization constraints. We found we could include parts of the cascade in our approach, such as handling units, and thus do. Normalization is a bottom-up tree rewriting pass, and an implementation optimization avoids performing it before layout and instead makes it an on-demand part of layout solving. We primarily focus in the core box model: normal flow (blocks and inlines), out of flow (relative and absolute positioning, floats), and borders, padding, and margins.

Our specification largely follows the style of the above grammars. Part of the intuition for the feasibility of specifying CSS in this way is that CSS was designed with restrictions that avoid requiring slow evaluation with techniques such as iterative constraint solving. In our encoding, each CSS display type is represented by one or more classes in our system. CSS's normalization algorithm largely leads to our set of interfaces, such as grouping the `inline` and `inline block` display types under interface `inline`. We make heavy use of traits and interfaces, which compromise 23% and 32% of the code, respectively. The automatic table layout algorithm was an extension of the above techniques. Finally, similar to the issue with table cells having two parents, a row and a column, out of flow elements also required encodings to support DAG behavior.

Several differences distinguish our experience with specifying CSS layout from the other case studies. Many features were difficult to specify because of many cases or cross-cutting in their semantics. Discussed in Chapter 3, we rely upon automatic checking to assist development, and discussed in Chapter 4, we specify schedule sketches to improve compiler speed and more quickly experiment with parallelization schemes. To further simplify development, we wrote several increasingly large specifications and manually integrated them.

One particularly challenging feature to disentangle relates to ambiguity. CSS solves seemingly inconsistent input constraints instead of returning an error. For example, if `H-AG` was extended to support input heights on intermediate nodes, the following conflict would require a graceful interpretation rather than refusing to render:

```
1 <hbox height="5">
2   <hbox height="500"></hbox>
3 </hbox>
```

By the original attribute grammar, the outer `<hbox>` should be the size of the biggest child, which would be 500. However, that conflicts with the input constraint of the outer box only being 5 tall. Our CSS grammar inspects for the presence of input attributes and prioritizes them. The analogous resolution for the `H-AG`example is the following:

```
1 loop children {
```

```
2     h :=
3       fold (maybeReady(height) ? maybeValue(height) : 0)
4       ..
5       maybeReady(height) ? maybeValue(height) : max($.h, child.h)
6   }
```

The grammar uses "5" and "500" because they were explicitly specified instead of solving for them.

We found other features to be difficult because they purposefully stray from the direct mathematical interpretation. For example, CSS supports input constraints where a node's width is defined as a proportion of its parent's. If we naïvely extended `H-AG` with such a feature, evaluation of the following layout would lead to a degenerate solution:

```
1  <hbox>
2    <hbox width="50%">
3      <hbox w="20"></hbox>
4    </hbox>
5  </hbox>
```

The root node shrinks to fit the middle node, but the middle node must be 50% of the parent. Direct interpretation leads to a solution of 0 for both widths, but CSS instead leaves the result up to the layout engine implementation. The first reason is that the result looks unappealing: the containers of the leaf node do not appear. The second reason is that, while iterative solvers may avoid some such situations, but only at the expense of performance. Implementations instead use non-iterative heuristics, and as seen with tables, their implementors struggle to understand the behavior.

In summary, our attribute grammar formalism was sufficiently expressive for specifying a non-trivial subset of the widely used layout language constructs.

2.5 Related Work

This chapter relates to three broad bodies of work: declarative languages in general, attribute grammars in particular, and constraint-based specification of layout.

Our formalism for declarative specification descend from that of attribute grammar literature. Attribute grammars were originally proposed as a formalism for describing language semantics by Knuth and Wegner (Knuth, 1990) and have since been applied to tasks such as developing Pascal compilers and implementing spreadsheet languages (Saraiva and Swierstra, 2003). Expressive extensions such as object orientation (Koskimies, 1991) and higher-order values (Vogt et al., 1989) have been previously examined. We want such expressive extensions for specifying our computations, and our challenge is in how to restrict them enough to support the parallelization techniques shown in the subsequent chapters. This chapter establishes that our formalism supports many key layout features, and by showing how to translate our extensions such as the class system into more basic attribute grammars, enable much of our subsequent static reasoning.

Statically unbounded loops in attribute grammars may be supported by encoding a list of children as a degenerate subtree – a chain. Klaiber and Gokhale (1992) show a way

to automatically detect this case in a BNF and transform it into one supporting Kleene stars (EBNF). We instead allow the specification writer to start with a EBNF and directly declare and operate over an unbounded number of children. A key difference is the runtime performance implications: a loop runs sequentially while nested subtrees may be parallelized: our approach gives programmers control over performance by providing two distinct choices. Our loop form resembles the uniform recurrence equations described by Karp et al. (1967). We integrated them into our overall language for computing over trees and add the ability to escape a synthesized loop mid-iteration in order to support recurring down the tree mid-loop.

Declarative layout has been a goal of computer science as early as Sutherland's Sketchpad from the "Mother of all demos" (Sutherland, 1963). Popular systems such as LaTeX (Knuth and Bibby, 1986) and CSS (Lie and Bos, 1997) provide significant high-level control for common design tasks. However, the declarative meaning of the constraints is unknown, which has challenged making multiple conformant implementations and correctly reasoning about them for purposes such as optimization and tool building. In practice, layout language designers manually implement and reason about their languages.

We are aware of several especially significant attempts for declarative definitions of layout that support automated reasoning. First, Heckmann and Wilhelm (1997) specify LaTeX's formula layout language in ML (Milner et al., 1997). Their description helps decompose the specification and provides equational reasoning due to the functional style. That said, the generality of ML provides few immediately provable properties relevant to our work. The specification of a non-automatic HTML table layout by Saraiva and Swierstra (2003) is similarly interesting in the use of the more restricted functional formalism of higher-order attribute grammars, though dynamic scheduling is generally associated with such grammars. Finally, Badros et al. (2001) built the Cassowary linear constraint solver and provide the language of linear constraints as the layout language. The restricted formalism enables significant automated reasoning, though it is not clear how to encode typical layout features such as line wrapping (Lin, 2006). In contrast, we support reasoning about much of the core CSS layout language and only escape the system to guarantee the correctness of the occasional use of directed acyclic graphs.

Chapter 3

Parallel Layout with Checkable Static Tree Traversal Schedules

We now describe parallelism in layout language constructs by introducing a static scheduling language for parallel tree traversals. For example, we run tessellation on a GPU by grouping dynamic memory allocation requests into one tree traversal and distribute allocated addresses in the next. Likewise, we parallelize word-wrapping by using a restricted form of nested parallelism where a different schedule is used based on the type of subtree. Neither encoding is obvious and both benefit from linguistic support.

Reasoning about such schedules is difficult. This chapter focuses on reasoning about correctness, showing how we verify that following a schedule will correctly implement the corresponding attribute grammar. To make the analysis tractable, we restrict the primitives in the attribute grammar and scheduling languages. For example, we restrict the looping construct to reduce reasoning about the correctness of loop code to non-looping code. The idea is to unroll loops several times so that, given a schedule for the unrolled statements, we can infer the schedule for the loops. The enabling restriction limits which array indices a loop may access. Likewise, by focusing on different types of tree traversals, we can create efficient implementations of each one (Chapter 5).

3.1 Design Goals

Our overall challenge in this chapter was to balance restricting the scheduling language enough to facilitate optimization while still providing the flexibility for expressing document layout and data visualization languages. Finding parallelism in CSS is already a novel concept; being able to optimize it with techniques associated with small formulas for physical models is especially surprising. Although layout computations have too many data dependencies to be solved with one simple tree traversal, we found that sequences of 3–5 traversals often suffice for data visualizations and 9 for CSS. Therefore, our scheduling language consists of traversal patterns, such as parallel top-down (*preorder*) traversal of the tree, and

ways of combining them, such as in a sequence. It cannot express all schedules, such as fixedpoint computations, but it can express common cases.

Our second challenge was in the correctness needs arising due to layout specifications being magnitudes bigger than stencil (Datta et al., 2008) and skeleton (Matsuzaki et al., 2006b) formulas. For stencil computations, the verification challenge lies more in correctly optimizing the implementation of a traversal schedule. Our layout computations encountered a challenge before that point: the size of the functional specification and the ensuing tangle of data dependencies require ensuring that the parallel schedule itself is safe to implement. Running the schedule on any input should compute the expected result. We used a variant of existing static dependency analyses of attribute grammars to verify that the schedule is race-free.

The dependencies that complicate reasoning about correctness of parallel code actually also complicate sequential code. Our use of static analysis for the attribute grammars led to an important result for layout languages: we demonstrate how to statically verify three important properties about a language and its schedule:

- **Totality** The layout language defines a solution for every syntactically well-formed input tree; the expected layout is unambiguous.
- **Determinism** Following a schedule always returns the same result for a given input. We check that the schedule respects the data dependencies in the attribute grammar.
- **Linearity (Single Assignment)** Every attribute is assigned exactly once. Layout languages often perform *reflow* to iteratively solve constraints or incremental computation, so this property bounds the need for it.

The first property demonstrates the ability to reason about *functional correctness* and the last two about *behavioral correctness*. Put together, we verify that a language is unambiguous, supports parallelization, and with bounded asymptotic complexity.

In addition, this chapter demonstrates how to parallelize common layout language constructs such as box models, word wrapping, tables, and even functional graphics (via tessellation).

3.2 Language of Static Schedules

This section focuses on defining our full language of traversal schedules. A schedule is the input for our code generators. Programmers may use this approach to automate specifying a schedule.

Statically scheduled evaluation departs from the dynamic evaluation strategy of Section 2.2. Static scheduling solves the performance problem of dynamic evaluation by repeatedly manipulating the data dependencies of every attribute at runtime, i.e., what would be a direct sequence of arithmetic statements in a static language becomes an interleaving of graph manipulations and arithmetic with the dynamic evaluator.

We use a schedule to statically declare most of the scheduling decisions. It specifies a sequence of tree traversals and the order of statements to use within each traversal. During a traversal at runtime, the order of nodes to traverse is based on the traversal pattern, such as top-down, rather than by inspecting data dependencies. Likewise, the statements required to execute a node are efficiently looked up based on the node's type rather than according to its data dependencies.

Our scheduling language is a more compositional variant of others. For example, Kastens (1980) defines a schedule as a mapping from node type to the sequence of functions to execute on successive visits to it. The choice of traversals is implicit within the definition, which complicates compiler optimization and language extension. Instead, our approach builds subtree traversals out of node visits, a traversal out of subtraversals, and the full schedule out of multiple traversals. Every scheduling unit is exposed to our downstream compilers. Likewise, when adding a new variant of any of the scheduling constructs such as a new tree traversal order, it is straightforward to add it to the verifier introduced in this chapter and the synthesizer introduced in the next.

Sequential Schedules

We start by examining how to specify a safe static schedule for `H-AG` that respects any possible dependencies in an input tree (Figure 2.3a).

Figure 3.1 shows a sequential implementation of `H-AG` decomposed into several pieces. The layout engine solves an input tree over a sequence of two traversals (Figure 3.1a). The first traverses the tree in postorder, meaning from the leaves up to the root ("bottom-up"), and the second performs a preorder traversal, meaning from the root down to the leaves ("top-down"). Figure 3.1b provides a sample implementation of generic traversal code. During a during traversal, each node is *visited* exactly once in order to compute the attributes whose dependencies have been satisfied. Figure 3.1c shows that the first pass computes widths and heights, and the second pass computes the "x" and "y" positions.

The example follows a static schedule rather than manipulating a dynamic data dependency graph. The sequence of traversal invocations and the code used for the different cases for each traversal's visitor determine the schedule. Each traversal now only performs dynamic scheduling in the sense of maintaining a stack for recurring down the tree, which is a cost proportional to the number of nodes rather than the size of the dynamic dependency graph between attributes. (Chapter 5's compiler and runtime optimizations even eliminate the implicit use of a call stack.)

We abstracted the schedule out of the implementation by introducing a compositional scheduling language (Figure 3.3). The schedule for the above computation would be appear as:

```
1  postorder
2    HBOX0 → HBOX1 HBOX2 { HBOX0.w HBOX0.h }
3    HBOX → ϵ { HBOX.w HBOX.h }
4  ;
5  preorder
```

```
1  postorder(visit1, start);
2  preorder(visit2, start);
```

(a) **Sequential sequence of traversals**

```
1  void preorder(void (*visit)(Prod &), Prod &p) {
2    visit(p);
3    for (Prod rhs in p)
4      preorder(visit, rhs);
5  }
6  void postorder(void (*visit)(Prod &), Prod &p) {
7    for (Prod rhs in p)
8      postorder(visit, rhs);
9    visit(p);
10 }
11 void recursive(void (*visit)(Prod &, int), Prod &p) {
12     int step = 0;
13     visit(p, step++);
14     for (Prod rhs in p) {
15       recursive(visit, rhs);
16       visit(p, step++); //repeat visit to p
17     }
18 }
```

(b) **Three sequential traversal patterns**

```
1  void visit1 (Prod &p) {
2    switch (p.type) {
3      case S → HBOX: break;
4      case HBOX → ϵ:
5        HBOX.w = input(); HBOX.h = input(); break;
6      case HBOX → HBOX1 HBOX2:
7        HBOX0.w = HBOX1.w + HBOX2.w;
8        HBOX0.h = MAX(HBOX1.h, HBOX2.h);
9        break;
10   }
11 }
12 void visit2 (Prod &p) {
13   switch (p.type) {
14     case S → HBOX:
15       HBOX.x = input(); HBOX.y = input(); break;
16     case HBOX → ϵ: break;
17     case HBOX → HBOX1 HBOX2:
18       HBOX1.x = HBOX0.x;
19       HBOX2.x = HBOX0.x + HBOX1.w;
20       HBOX1.y = HBOX0.y;
21       HBOX2.y = HBOX0.y;
22       break;
23   }
24 }
```

(c) **Scheduled and compiled visits for `H-AG`.**

Figure 3.1: **Sequentially scheduled and compiled layout engine for `H-AG`.**

```
6     S → HBOX { HBOX.x HBOX.y }
7     HBOX0 → HBOX1 HBOX2
8       { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

It specifies a sequence (";") of two traversals of node visit order postorder and preorder. For each type of node visited within a traversal, the schedule specifies the sequential sequence of attributes to evaluate. Note that, due to the desugaring of our class system in Section 2.3, the dispatches in the above examples are based on grammar productions in the desugared representation. In terms of the fronted language, the dispatches are based on node class.

We consider each level of abstraction in the schedule in turn. First, the schedule performs a *sequence* of two different *types of traversals*:

```
1       postorder(visit1, start); preorder(visit2, start)
```

Later, we discuss parallel composition ("——").

Next, the schedule specifies traversal types. Just one bottom-up traversal cannot compute all of the attributes, such as all the "x" and "y" attributes that flow downwards (Figure 2.3a), so the schedule carefully orders multiple traversals of different types. Other traversal types are possible: this section also explores recursive, and nesting is described in Section 3.2.

The third control abstraction within a schedule specifies different orders of statements for different types of nodes. For example, when visiting an HBox node as part of visit2, the schedule includes the following fragment:

```
1  HBOX0 → HBOX1 HBOX2 {
2        HBOX1.x; //Semantic function: λHBOX0.x : HBOX0.x
3        HBOX2.x  //Semantic function:  λHBOX0.x, HBOX1.w : HBOX0.x + HBOX1.w
4        ...
```

The schedule specifies that $\text{HBOX}_2.\text{x}$ can (and should) be immediately evaluated after $\text{HBOX}_1.\text{x}$ without fear of unsatisfied data dependencies for any of the arguments needed by its semantic function. In summary, there are three parts to a schedule: the staging of traversals, the node visit order for every individual traversal, and the statement order for different types of nodes within a specific traversal.

Generally, a single attribute grammar may be scheduled in many ways. For example, the width and height computations share no dependencies, so the first postorder traversal might be partitioned into two postorder traversals:

```
1  postorder
2    HBOX0 → HBOX1 HBOX2 { HBOX0.w }
3    HBOX → ϵ { HBOX.w }
4  ;
5  postorder
6    HBOX0 → HBOX1 HBOX2 { HBOX0.h }
7    HBOX → ϵ { HBOX.h }
8  ;
9  preorder
10   S → HBOX { HBOX.x HBOX.y }
11   HBOX0 → HBOX1 HBOX2
12     { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

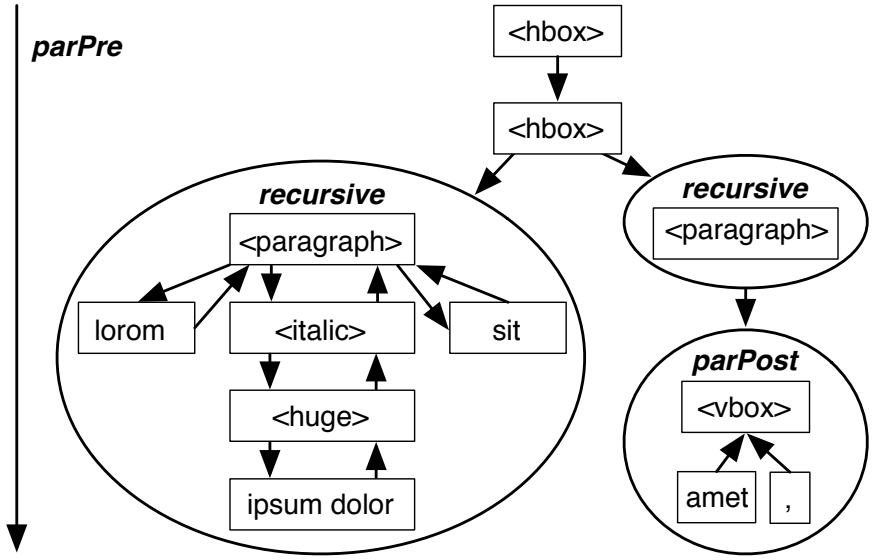


Figure 3.2: **Nested traversal for line breaking**. The two paragraphs are traversed in parallel as part of a preorder traversal. A sequential recursive traversal places the words within a paragraph. Circles denote nested regions and arrows show data dependencies between nodes and/or regions.

Rescheduling in this way may improve performance on small devices with little memory because the schedule cuts the working set size in half for each traversal. Verified changes to the schedule only optimizes execution; it does not change the result of evaluation.

Sequential execution supports a traversal type that can compute more than postorder or preorder, which we call a recursive traversal (Figure 3.1b). For example, we use a recursive traversal for line breaking in our document layout case study. Consider inserting line breaks into the following stylized paragraph of XML strings (Figure 3.2):

```
lorom <italic><huge>ipsum dolor</huge></italic> sit
```

Due to `<huge>`, the paragraph may need a line break between "ipsum" and "dolor." Identifying the line break position involves visiting the subtree `<italic>...</italic>`; the resulting line break position is a data dependency influencing line breaks in the remainder of the text. The sequence of arrows in the big circle of Figure 3.2 shows a trace of performing a recursive traversal over the paragraph. The traversal visits a node n, then visits n's first child, revisits n, and repeats this process for the remaining children before returning to the parent.

The relationship between recursive traversals and postorder and preorder merits examination. First, a sequence of a preorder traversal followed by a postorder traversal may be merged into one recursive traversals. Traversing a tree induces overhead costs, so such fusion may be beneficial. The reverse relationship is not true, however. As happens with the case of line breaking, long-running sequential dependencies may prevent splitting a recursive traversal into a preorder and postorder traversal. These dependencies arise because the the same node

```
parPost
  HBOX0 → HBOX1 HBOX2 { HBOX0.w HBOX0.h }
  HBOX → ϵ { HBOX.w HBOX.h }
;
parPre
  S → HBOX { HBOX.x HBOX.y }
  HBOX0 → HBOX1 HBOX2
    { HBOX1.x HBOX2.x HBOX1.y HBOX2.y }
```

(a) **One explicit parallel schedule for** `H-AG`**.**

```
void parPre(void (*visit)(Prod &), Prod &p) {
  visit(p);
  for (Prod rhs in p)
    spawn parPre(visit, rhs);
  join;
}
void parPost(void (*visit)(Prod &), Prod &p) {
  for (Prod rhs in p)
    spawn parPost(visit, rhs);
  join;
  visit(p);
}
```

(b) **Naïve traversal implementations** with Cilk's Blumofe et al. (1995) **spawn** and **join**.

```
parPost(visit1, start); parPre(visit2, start);
```

(c) **Scheduled and compiled layout engine for** `H-AG`**.**

<Sched> → *<Sched>* ; *<Sched>* | *<Sched>* || *<Sched>* | *<Trav>*

<Trav> → *<TravAtomic>* *<Visit>**{(*<TravAtomic>* ↦ *<Visit>**)*}?

<TravAtomic> → **preorder** | **postorder** | **parPre** | **parPost** | **recursive**

<Visit> → *<Prod>* { *<Step>** }

<Step> → *attrib* | **recur** *v*

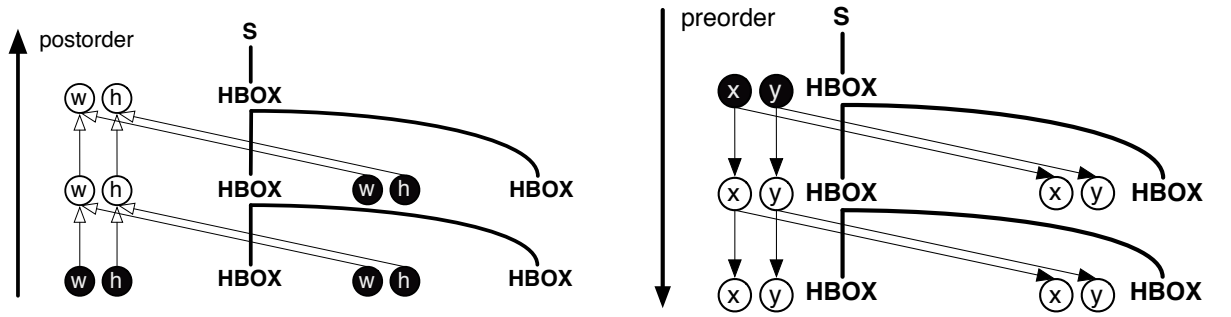
(d) **Language of schedules** (without holes)

Figure 3.3: **Scheduled and compiled layout engine for** `H-AG`**.**

is visited multiple times in a traversal: once before a child subtree is traversed and again after. The result of computing over one subtree may therefore be used to compute another, which supports long-running sequential dependencies.

Parallel Schedules: Same Traversal

A schedule exposes structured parallelism both within a traversal and across them.



(a) First traversal: parallel postorder

(b) Second traversal: parallel preorder.

Figure 3.4: **Parallel traversal**. Shown for constraint tree in Figure 2.2. Circles denote attributes, with black circles denoting attributes with resolved dependencies such as input()s. Thin lines show data dependencies and thick lines show production derivations. First diagram shows dependencies followed by first traversal, and second for the following traversal.

For an example of parallelism within a traversal, the first postorder traversal for `H-AG` features latent parallelism. The widths and heights for one subtree can be computed independently of the widths and heights of another distinct subtree. Figure 3.4a shows an example where different (logical) threads may compute on the leaf nodes and implicit barriers force a join at every intermediate node. Likewise, the second traversal (Figure 3.4b) may be changed to a parallel preorder traversal wherever intermediate node acts as a logical fork. Figure 3.3b depicts naïve parallel implementations using Cilk's (Blumofe et al., 1995) spawn and join primitives. We formulate the schedule by changing the specification from postorder and preorder to parPost and parPre (Figure 3.3a).

Our *nested* traversal feature supports exploiting parallelism within a traversal even if some nodes require sequential evaluation. With nesting, the tree is partitioned into an outer region and disjoint inner regions according to statically defined rules. The outer and inner regions are evaluated with different traversals, and both may exploit parallelism. We can think of the inner regions as macro-nodes that are evaluated in full (with their particular traversal type) when the outer traversal encounters them.

To motivate the need for nested traversals, we revisit line breaking. Even though line breaking of a single paragraph is sequential, distinct paragraphs of text can be handled in parallel. To avoid locally sequential computations from forcing the entire tree traversal to be sequential, we allow the outer region to be parallel, while each paragraph forms an inner region that is handled with the sequential recursive traversal. Figure 3.2 shows using parallel evaluation to compute across different **recursive** paragraphs. Likewise, it shows a hypothetical VBox subtree that uses parallel postorder evaluation for traversing its subtree as soon as the outer parallel preorder traversal reaches it.

To partition a tree into regions, the schedule maps each grammar production (and thus

each node of the tree) to a traversal type. A subtree composed from nodes of the same traversal types form an inner region. For example, a nested traversal of paragraphs with sequential traversals of nested text subtrees is described as follows:

```
1  parPre
2    P → W { W.relativeX }
3    { recursive ↦
4        W0 → W1 W2 {
5          W1.relativeX recur W1
6          W2.relativeX recur W2 } }
```

Different traversals may use different partitionings.

Parallel Schedules: Across Travesals

Our scheduling language also supports exploiting parallelism across traversals. For example, just as we created a different but functionally equivalent sequential schedule for `H-AG`, we can also design a schedule that is parallel:

```
1  (    parPost
2         HBOX0 → HBOX1 HBOX2 { HBOX0.w }
3         HBOX → ϵ { HBOX.w }
4     ||
5       parPost
6         HBOX0 → HBOX1 HBOX2 { HBOX0.h }
7         HBOX → ϵ { HBOX.h })
8  ; parPre ... /* same as before */
```

The "||" construct specifies that one traversal may be run concurrently with another. Neither traversal traversal depends on attributes written by the other, so the parallelizaton is safe. Even if we cannot exploit parallelism within a traversal, using "||" enables us to parallelize across them.

Compilation

Compilation only requires an attribute grammar and its schedule. For example, the traversal staging postorder _ ; preorder _ directly translates to the executable fragment in Figure 3.1a. Likewise, the mapping from traversal productions to statement sequences, such as HBOX → ϵ { HBOX.w HBOX.h }, directly translate to the visit functions of Figure 3.1c. The translation matches an attribute in the schedule with the left-hand side attribute of an equation in the attribute grammar and outputs the full assignment statement in its place.

Our code generation pipeline is more complicated but conceptually similar. The schedule is combined with the attribute grammar to form an intermediate representation, and different code generators target different backends such as JavaScript, OpenCL, and C++. Furthermore, some of the reductions of Section 2.3 require augmenting or rewriting the intermediate representation, such as reinserting loops that were unrolled during scheduling (Section 3.3). Our code generator typically runs after the schedule synthesizer and autotuner (Chapter 4).

3.3 Desugaring Loops

Many of the difficulties in computer science stem from handling loops. In our case, how can we statically schedule uses of the declarative loop construct of Section 2.3? The construct extends the language of statements to include non-nested loops, and an attribute computed in one step of one loop may depend on that of another. To avoid implementation complexity, we schedule loops through a reduction to a language without loops. Our insight was to finitely unroll any loop a fixed number of times in such a way that any schedule for the unrolled steps generalizes to a loop over an arbitrary number of items at runtime. Chapter 4 examines how to synthesize schedules; the current concern is the definition and checking of such a schedule.

Our problem is distinct from that of classical attribute grammar languages for two reasons. First, modern formalisms focusing on expressivity generally rely upon dynamic scheduling. They require little from the schedule. Second, for the formalisms that support static scheduling, loops would be over the tree rather than as part of the statement language. For example, a list of values would be encoded as a chain in the tree:

<BinaryNode> → *<ValueList> <BinaryNode> <BinaryNode>* | ϵ

<ValueList> → *number <ValueList>* | ϵ

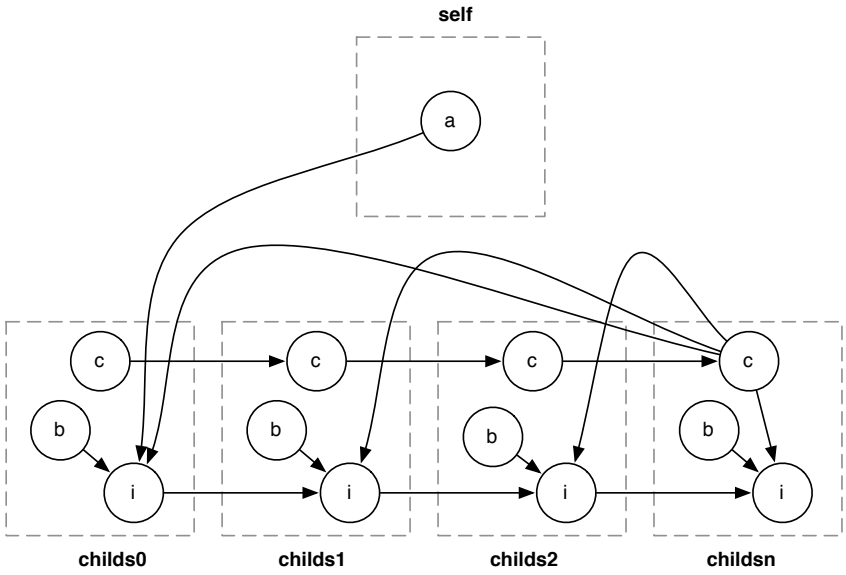
The position of a number in a ValueList chain corresponds to the tree level; therefore, a loop over the chain involves a full tree traversal. However, data dependencies for loops over a node list are generally local to that list. If the list must be computed over sequentially, being able to treat it as a single visit step simplifies computing over it as part of an overall parallel traversal. Our nested traversal construct addresses the need in theory, but local loops are convenient.

In terms of the above encoding, our support of loops corresponds to extending ordered attribute grammars with a Kleene star. The above program could then keep a list of values local to a single production:

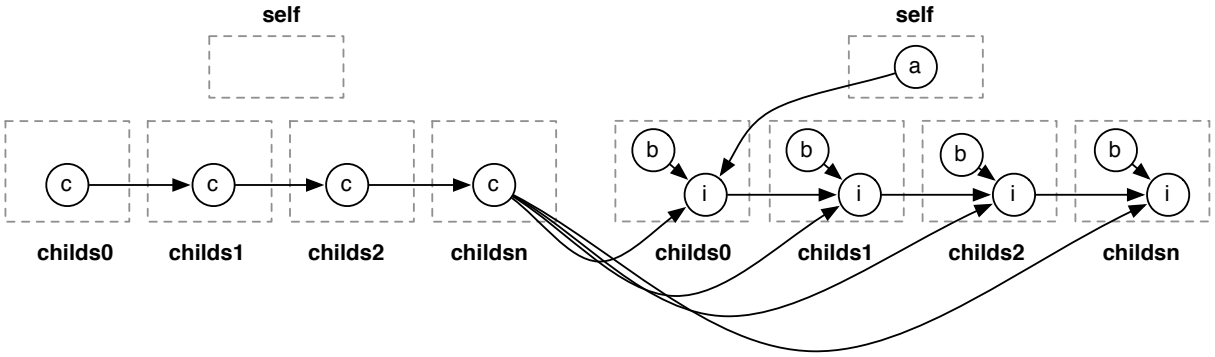
<BinaryNode> → *number* <BinaryNode> <BinaryNode>* | ϵ

The language of constraints over loop node attributes are recurrence relations (Karp et al., 1967). Every attribute in the list sequence may be defined in terms of previous ones. Our embedding of recurrence relations into a system of tree traversals leads to subtle interactions, however. For example, in a recursive traversal (Section 3.2), each loop step may require recurring through a subtree before performing the loop step. We need a different recurrence relation scheduler than existing ones.

Our approach is to divide the problem into two steps. First, we transform an attribute grammar with loops into one without them by unrolling several steps of the loop. Second, after scheduling the loopless grammar, we recover loops from the schedule through an inference procedure. Our approach guarantees that, if the synthesizer reports a loopless schedule, dependency-preserving loops will be recovered from it.



(a) **Unrolled Loop Dependencies.**



(b) **Staging as Two Loops.**

Figure 3.5: **Loop scheduling**. The loops may be scheduled for the same traversal if both attributes a and b are available ahead of time.

Reduction to OAGs by Unrolling

See below for how to schedule the following loop:

```
1  interface NodeI {
2    var c : int;
3    var i : int;
4    input a : int;
5    input b : int;
6  }
7  class NodeC : NodeI {
8    children {  childs : [ NodeI ]; }
9    actions {
10     loop childs {
11       childs.c := fold 0 .. childs$-.c + 1;
12       childs.i :=
13         fold
14         a
15         ..
16         childs$-.i + childs$i.b + childs$.c;
17     }
18   }
19 }
```

Our reduction unrolls the loop into 4 steps (0, 1, 2, and n):

```
1  interface NodeI {
2    var c : int;
3    var i : int;
4    input a : int;
5    input b : int;
6  }
7  class NodeC : NodeI {
8    children {
9      childs0, childs1, childs2, childsn : NodeI;
10   }
11   actions {
12     childs0.c := 0 + 1 + 0;
13     childs1.c := childs0.c + 1 + childs0.c;
14     childs2.c := childs1.c + 1 + childs1.c;
15     childsn.c := childs2.c + 1 + childs2.c;

17     childs0.i := a + childs0.b + childsn.c + a;
18     childs1.i := childs0.i + childs1.b + childsn.c + childs0.i;
19     childs2.i := childs1.i + childs2.b + childsn.c + childs1.i;
20     childsn.i := childs2.i + childsn.b + childsn.c + childs2.i;
21   }
22 }
```

The reduction performs several rewrites that unroll loops and then substitutes variable names in the unrolled statements (Figure 3.6). The first key property that the unrolling preserves is that the dependencies are preserved. The unrolling does this in several ways:

- **Schema unrolling.** It unrolls every declaration "child : [NodeI]" into the following form: child0, child1, child2, childn : NodeI
- **Substitution.** The first step of a loop unfolds by replacing references of the form "child$-.fld" to use the initial value specified in the first part of a "fold" expression.

$$
\begin{aligned}
[\![\ child : [interface]\ ]\!] &\to child0, child1, child2, childn : interface \\
[\![child.fld := \texttt{fold}\ e_{\texttt{init},fld}\ ..\ e_{\texttt{step}}]\!] &\to \\
& child0.fld = [\![\ e_{\texttt{step}}[\forall\ f :\ e_{\texttt{init},f}\ /\ child\$\text{-}.f\ ]\!]_0 + e_{\texttt{init},fld}; \\
& child1.fld = [\![\ e_{\texttt{step}}\ ]\!]_1 + child0.fld; \\
& child2.fld = [\![\ e_{\texttt{step}}\ ]\!]_2 + child1.fld; \\
& childn.fld = [\![\ e_{\texttt{step}}\ ]\!]_{\texttt{n}} + child2.fld; \\
[\![\ child\$\$.fld\ ]\!]_\alpha &\to child\texttt{n}.fld \\
[\![\ child\$i.fld]\!]_\alpha &\to child\alpha.fld \\
[\![\ child\$\text{-}.fld]\!]_1 &\to child0.fld \\
[\![\ child\$\text{-}.fld]\!]_2 &\to child1.fld \\
[\![\ child\$\text{-}.fld]\!]_{\texttt{n}} &\to child2.fld
\end{aligned}
$$

Figure 3.6: **Rewrite rules for loop reduction**. Cases of $[\![\cdot]\!]$ that simply recur are elided.

Likewise, step 1 will substitute the reference with "child0.fld", and "child$i.fld" for "child1.fld". Finally, it replaces every reference to last value "child$$.fld" with "child.fld".

- **Forward Loop Direction.** The rewriting enforces a forward loop direction by making child1.fld depend on child0.fld, child2.fld depend on child1.fld, etc. The dependencies simplify later analysis by eliminating concerns in safely reordering steps of a loop. To support an alternative loop order, such as backwards, these dependencies would be elided or encode multiple options, and the recovery algorithm would perform more reasoning.

The result of the rewriting is an attribute grammar without loops. Our full implementation differs in two significant ways. First, it performs static checks such as that the fold initialization expression does not refer to step variables "child$i.fld" nor "child$-.fld". Likewise, statements looping over one collection are checked for references to intermediate elements of another. Second, the rewriting supports loops that may temporarily escape as part of a recursive traversal. Each loop step over an element may require traversal into the element's subtree, so we expand child attributes with a local and transfer version in order to reason about safe placement of the recursive call.

Recovery by Commuting Abstractions

If the rewritten grammar can be scheduled, so can the original grammar with loops. We extract loops from the scheduled grammar and guarantee that any dependency in the original grammar is safely obeyed by the extracted loops. A difficult part of the guarantee is proving

that the procedure for recovering loops from the schedule will not get "stuck." This section describes the loop recovery process and its correctness.

The algorithm first rearranges loop statements into distinct blocks. The scheduler may interleave statements from loops over distinct sets of children, but code generation needs them to be separated. All non-loop assignments must also be taken out and fit between loop blocks. The procedure iteratively partitions a scheduled sequence of attributes into several subsequences until it reaches a normal form and cannot proceed further.

For each iteration, the algorithm selects a sequences of attributes starting with "child0.fld", which represents the beginning of a loop. It partitions the attributes following it into those that must occur before the loop, after the loop, or during it. The loop's partition is then ready for code generation, and the algorithm repeats on the other two partitions. Because the algorithm finalizes a loop with at least one attribute in each step, we guarantee that the algorithm terminates.

The partitions for one step of the algorithm are determined by applying the following three transformations. The intuition for each is that, due to the loop unrolling, any valid schedule for dependent variables will obey certain properties. A schedule matching them *may* be due to loop dependencies, and allowing the transformations, a schedule that does not *cannot*.

- **Extract non-loop assignments from a loop**. The beginning of a loop corresponds to an assignment to "child0.fld" and the end by an assignment to "childn.fld". Assignments to non-loop variables that occur within the loop range are moved to be before the beginning of the loop. They are moved out in case multiple statements are scheduled for the same loop and some of them depend on the non-loop variables. All non-loop assignments in a loop range are moved out with their relative ordering preserved.

  Moving a non-loop assignment to before the loop is safe. If the assignment depended on a loop variable, syntactic restrictions guarantee that the variable could only have been the final one, child$$.fld, and a valid schedule could not have placed the assignment inside of the loop range. Conversely, if a loop statement depends on the non-loop assignment, moving the assignment earlier preserves the ordering. Finally, moved non-loop assignment statements may have mutual dependencies, so maintaining their relative ordering during the movement preserves any read-after-write dependencies.

  The process is guaranteed to terminate at a fixpoint. First, moving statements out of one loop completes in time linear in the size of the range of the loop. Second, the finite number of loops means that the process only repeats a finite number of times because movement only occurs in one direction.

- **Separate loops over different collections.** The algorithm iteratively separates loops over different collections. First, it detects mutually dependent statements that must be scheduled as part of the same loop. Then, it examines the loop span for statements belonging to another type of loop and moves them either to before or after

the base range of mutually dependent statements. The moved statements maintain their ordering relative to other statements moved to the same side of the range.

The complexity of the operation stems from mutually dependent loop variables. For example, the following code must be scheduled into the same loop:

```
1 loop childs {
2   childs.a := fold 0 .. childs$-.b + 1;
3   childs.b := fold 0 .. childs$-.a + 1 + otherChilds$$.c;
4 }
```

The partial order for the resulting schedule is "(a0|b0) (a1|b1) (a2|b2) (an|bn)". If "cn" is scheduled as "a0 c0 d0 c1 d1 c2 d2 cn b0 dn", the "c" computations do not depend on "a" nor "b" ones, but not vice-versa. The "c" loop must be moved ahead. Doing so is safe relative to "a,b,.." because "c" statements do not depend on them. Furthermore, if "c" is dependent on other statements in the range, those would also be moved with it, such as seen with "d". For the remaining statements, "a,b,..." do not depend on them and the algorithm moves them after.

The algorithm terminates because it recursively operates on successively smaller partitions: statements moved earlier, the current loop range, and statements moved after.

- **Separate staged loops over the same collection** Loops over the same collection may still need to be separated. Consider the following loop:

```
1 loop childs {
2   childs.a := fold 0 .. childs$-.b + 1;
3   childs.b := fold 0 .. childs$-.a + 1 + childs$$.c;
4   childs.c := fold 0 .. childs$-.c + 1;
5 }
```

A valid resultant schedule would be the same as the above case: a loop computing c attributes must run before a loop computing all a and b attributes. In fact, the same reasoning as applied above applies to this case, except now the recurrences are all over the same variable *childs*.

Due to dependencies across statements being moved before or after a loop, each partitioning step performs all of the above separations. The relative order of statements moved out of a loop is thereby preserved.

Once the partitioning completes, the code generator receives a a list of blocks. Each block is for all loop statements or all non-loop statements. The code generator handles blocks of non-loop statements as usual. A block of loop statements will translate into a single loop. For example, handling the above code yields:

```
1 for (int i = 0; i < childs.length; i++) {
2         childs[i].c = (i == 0 ? 0 : childs[i - 1].c) + 1;
3 }
4 for (int i = 0; i < childs.length; i++) {
5         childs[i].a = (i == 0 ? 0 : childs[i - 1].b) + 1;
6         childs[i].b = (i == 0 ? 0 : childs[i - 1].a) + 1;
7 }
```

Note that translation of references of the form "childs$−.fld" require tracking the loop step in order to select the initial value or the previous node's value.

The end result is that, given an attribute grammar with loops and schedule for an unrolled version, we can recover a schedule for an attribute grammar with loops and then perform code generation.

3.4 Verification

We automatically check an attribute grammar and its schedule for safety. This section focuses on two aspects of our approach: the properties to verify and the modular design of the verification procedure. The properties are significant in that they cover both functional and behavioral correctness, and are typically desired but not proven for layout languages and pattern programs. Furthermore, we check the properties through axiomatic reasoning parameterized by a local dependency analysis. This proof structure simplifies extending the language of statements and of schedules because most additions correspond to an isolated and composable axiom. In Chapter 4, we change the verifier into a synthesizer and thereby achieve fully automatic and computer-assisted parallelization.

Our approach automatically checks three properties:

- **Totality** The attribute grammar defines one and only one solution for every well-formed input tree.
- **Determinism** The schedule evaluates the constraints of the attribute grammar without any data races.
- **Linearity (Single Assignment)** Every attribute is assigned exactly once. Layout languages often *reflow* by iteratively and incrementally solving constraints, so linearity provides an important monotonicity property for optimization.

In this chapter, we illustrate how to check race freedom. The check for linearity is similar, and totality is a consequence of the determinism and linearity properties.

We use a modular checking strategy for two reasons. First, we encountered implementation challenges without it. Our initial attempts to adapt the OAG algorith (Kastens, 1980), which is a search over a global dependency graph, suffered from many implementation bugs and we abandoned it. Instead, our new approach decouples verification from synthesis and pattern checking from dependency analysis. Second, challenging our OAG implementation and the basic premise of our approach, we needed to support adding new types of traversals. New schedule combinators, such as nested traversals, and individual patterns, such as recursive, should be simple to add as new types of parallel patterns are understood. Adding a parallel pattern should not require refactoring the entire verifier or synthesizer. Our new approach phrases each pattern as an independent axiom and automatically incorporates it into the checking procedure.

$$\frac{\{A\}\ p\ \{B\} \qquad \{B\}\ q\ \{C\}}{\{A\}\ p\ ;\ q\ \{C\}}\ (seq)$$

$$\frac{\{A\}\ p\ \{B\} \quad \{A\}\ q\ \{C\}}{\{A\}\ p\ ||\ q\ \{B \cup C\}}\ (par)$$

$$\frac{\begin{array}{l} Regions = \{\alpha \mapsto Visit*_\alpha\} \ \cup \ \bigcup_i \{\beta_i \mapsto Visit_i*\} \\ \forall\ \ (\gamma \mapsto Visit*) \in Regions : \\ \quad C_\gamma = \texttt{alwaysCommunicate}_\alpha(\gamma, B, Regions) \\ \quad \{A, C_\gamma\}\ \ \gamma\ Visit*\ \ \{A \cup B_\gamma\} \end{array}}{\{A\}\ \ \alpha\ \ Visit*_\alpha \{(\beta_i \mapsto Visit_i)*\}?\ \ \{A \cup \ \bigcup B_\gamma\}}\ (nest_\alpha)$$

$$\frac{\begin{array}{c} P = \cup Prod_i \quad Steps = \cup Step_j \\ B = \bigcup_i \texttt{reachable}_\beta(Prod_i, P, A, Steps, C) \end{array}}{\{A, C\}\ \ \beta\ (Prod_i\ \{\ Step_j*\ \})*\ \ \{A \cup B\}}\ (check_\beta)$$

Figure 3.7: **Correctness axioms for checking a schedule.**

Axiomatic Checking for Modularity and Correctness

Correctness axioms for checking an entire schedule are shown in Figure 3.7. The judgments recursively check a composition of traversals until reaching the traversal-specific checks of Figure 3.8. Checking is worst-case time linear in the number of attributes and the number of their local dependencies. As a reminder, Figure 3.3d defines the language of schedules.

We introduce a small amount of notation. Variables p and q denote schedules (<Sched>), A and B are sets of attributes, and α and β are traversal types (<travAtomic>). Attribute $a_{W,V\to W}$ is decorated with its production ($V \to W$) and the non-terminal within it (W). We write $a_{*,V\to W}$ if a can be associated with a non-terminal on either side of the production.

The rules to check composition and individual traversals are as follows:

- **Sequential and parallel composition: ";" and "||"** The simplest composition check is for sequencing: Hoare triple "{A} p ; q {C}" (rule **seq**). If attributes A are solved before traversal "p ; q", then attributes C will be solved after. The conditions above the judgment bar state this is true if p can always compute attributes B given attributes A, and q can always then compute C. The judgment is recursive. Analogous reasoning explains "||" (rule **par**).

$$\texttt{alwaysCommunicate}_{\texttt{parPre}}(\beta, B, M) = \{a_{W,W\to X} \mid (W\to X\ B_\beta) \in M[\beta] \bigwedge_{(V\to W\ B_\gamma)\in M[\gamma\neq\beta]} a_{W,V\to W} \in B \cup A\}$$

(a) Communication check for region boundaries in a **parPre** traversal

```
set reachable_parPre(W→X, P, A, B, C):
  reach :=
      {a_{*,W→X} | a_{*,W→X} ∈ A} ∪ (C ∩ {a_{W,W→X} | ⋀_{V→W∈P} W.a_{V→W} ∈ B}) ∪ (C ∩ {a_{X,W→X} | ¬∃X→Y ∈ P})
  while true:
    progress := {a_{*,W→X} | a_{*,W→X} = f(b_0, ..., b_n) ∈ F
                  ∧ a_{*,W→X} ∈ B ∧ ⋀ b_i ∈ reach}
    reach := reach ∪ progress
    if progress = ∅:
      break
  return reach
```

(b) Unoptimized production visit check for **parPre** traversal

Figure 3.8: **Inter- and intra-region checkers for parPre.**

- **Nested composition:** $\mapsto$ Rule $\textbf{nest}_\alpha$ checks outer traversal type α over regions where each one may have its own traversal type γ. Consider an outer traversal type of **parPre**: as it progresses top-down, every region might be guaranteed to have attributes of its root node solved before evaluation proceeds within it. For each region (the set of productions mapped to region traversal type γ), the rule calls $\texttt{alwaysCommunicate}_{\texttt{parPre}}$ to find the set C_γ of attributes that are externally set before the region is traversed. Rule $\textbf{nest}_\alpha$ calls checks for every region under the assumption that C_γ is already solved.

  The first line of rule $\textbf{nest}_\alpha$ means that, for any outer traversal α, attributes scheduled for the outer region are treated as if they were in their own region ($\gamma = \alpha$). Traversals that do not use nesting are degenerate: all the productions belong to one region ($\gamma = \alpha$).

- **Traversal over a region (e.g., parPre)** The schedule for a traversal of type β over a region is correct if every production visit schedule is correct (rule $\textbf{check}_\beta$). A production visit schedule $Prod_i\ \{\ Step_j\ *\ \}$ is correct when there is an order for computing its scheduled attributes $Step_j*$ along which all of the data dependencies of the corresponding semantic functions are satisfied.

  Traversals that do not perform nesting, such as a single occurrence of **parPre**, are handled as degenerate nested composition with one region: the entire tree.

- **Production visit**

  A fast and simple checking algorithm would involve marking each attribute of a production as dirty or clean inside a structure that persists across checks of different visits to the same production. For each successive attribute in a visit's sequence, if all of

its dependencies are met (dirty), mark the attribute, and otherwise fail the check. Non-local dependencies can be handled as below.

To optimize the synthesis algorithm of Chapter 4, we use a slightly indirect algorithm to check the correctness of visiting a production. The intuition is that it relaxes the specification of visit's attributes by treating the ordered sequence as an unordered set because the underlying check is for reachability in the set's dependency graph. Instead of checking all permutations of some attribute sequence (a,b,c,d) against the topological order of their dependencies, it sorts set {a,b,c,d} and checks once.

Figure 3.8b shows an unoptimized reachability computation for visiting a production inside a **parPre** region. It is the standard transitive closure except for two subtleties:

1. Only attributes that are meant to be scheduled are considered reachable (B membership checks). Incorrectly including unscheduled attributes would erroneously allow attributes with unresolved dependencies to also be included.
2. Attributes computed by visits to adjacent productions must be distinguished. Adjacent productions may be in the same region or in another. In a **parPre** region, consider when W is always an intermediate node of the region and attribute $a_{W,W\to X} \in B$ is always set by a parent production $V\to W$ in the same region. For this intra-region case, $a_{W,W\to X}$ is guaranteed to be reachable at the beginning of the visit to $W\to X$. However, if W can be the root node of the region, we must also check $a_{W,V\to W}$ is set by adjacent regions before the root is visited.

Checking an explicit sequence reduces to checking that the transitive closure can be performed in the specified order rather then the declarative definition shown in Figure 3.8. The synthesizer of Chapter 4 does not need to check for ordering, so we omitted this check.

Property Proofs

The axioms check for determinism, which can be adapted to check the two other properties.

First, the axioms check determinism, which means that rerunning the schedule will yield the same result. We can check determinism by ensuring that a schedule computes the attributes of an attribute grammar without races. More precisely, it tracks what attributes are guaranteed to have been computed by any particular point of the schedule, and uses that to check that every step of the computation only relies on what is guaranteed to have been computed.

Next, linearity requires that every instance of an attribute is only assigned to once. We can check linearity by extending the axioms in two ways. First, they must check that for any given attribute X_a, it is either defined by all productions $X \to W$ or by all productions $W \to X$. Second, for every attribute assigned in production $X \to W$, it must only be scheduled for one visit.

Totality guarantees that every well-formed input tree yields one and only one result. It is a property of the language because any schedule must reach the same result. In contrast, determinism is a property of a schedule because, for the same language, rerunning one schedule may always return the same result even as the same might not be guaranteed for another schedule. The proof of totality lies in the proof of linearity. Given a linear schedule, the dynamic dependency graph of every input document is directed and acyclic. The DAG property guarantees that the value of every attribute is a pure function of the values of the dominating attribute in the dependency graph, and therefore the language has a (total) functional interpretation. Checking totality adds an additional step beyond checking linearity: totality requires that every attribute in the grammar appears in the schedule.

Verification is $O(A \texttt{ log } A)$

Verifying a schedule for race-freedom takes time linear in the number of attributes A. Our description of the checker in Figures 3.7 and 3.8 does not show the optimizations that led to this bound; we highlight the key ideas here.

First, we observe the axioms to check form a tree. The fringe represents traversals, and intermediate nodes partition the attributes among them. The number of levels is $\texttt{log } A$.

Second, the time to check an axiom is linearly bounded by the number of attributes (and their dependencies) to be scheduled by that axiom. For example, checking the visit to a production is effectively a topological sort of the local dependency graph restricted to the attributes evaluated during the visit. Topologically sorting dependency graph $G = (E, V)$ is $O(|E| + |V|)$. Because an attribute has at most A local dependencies, verification takes time $O(A)$. For simplicity, our implementation does not use the topological sort optimization, and we only encountered performance issues in one case study affected by that.

Combining these observations yields the complexity. The time to check a level is A and there are $\texttt{log } A$ levels, therefore, the complexity of verification is $O(A \texttt{ log } A)$.

Similar reasoning applies to deriving the same complexity for checking the two other properties. To verify linearity, each attribute is labeled based on the type of production that solves it and rule $\textbf{check}_\beta$ checks that the labels of attributes in $Step_j$ match production $Prod_i$. The time to check the axiom is therefore still bounded by the number of scheduled attributes. Finally, verifying totality involves checking that the computed set of attributes matches the total set, and comparing two sets is also linear in the number of attributes.

3.5 Case Study: Automatically Staging Memory Allocation for SIMD Rendering

The static language of traversals is restricted, eliciting concern with whether it is too restricted to express common cases. Prominent in our case studies, many programs use dynamic memory allocation, but it is unclear how to perform it on a GPU without significant performance penalties. Our solution, based on the idea of scan operators (Chatterjee et al.,

```
1 float *drawCircle (float x, float y, float radius) {
2   float *buffer = malloc( (2 * sizeof(float) ) * round(radius))
3   for (int i = 0; i < round(radius); i++) {
4     buffer[2 * i] = x + cos(i * PI/radius);
5     buffer[2 * i + i] = y + sin(i * PI/radius);
6   }
7   return buffer;
8 }
```

(a) **Naıve drawing primitive.**

```
1 int allocCircle (float x, float y, float radius) {
2   return round(radius);
3 }
```

(b) **Allocation phase of drawing**.

```
1 int fillCircle(float x, float y, float radius, float *buffer) {
2   for (int i = 0; i < round(radius); i++) {
3     buffer[2 * i] = x + cos(i * PI/radius);
4     buffer[2 * i + i] = y + sin(i * PI/radius);
5   }
6   return 0;
7 }
```

(c) **Tessellation phase of drawing**.

Figure 3.9: **Partitioning of a library function that uses dynamic memory allocation into parallelizable stages.**

1990) for labeling nodes, performs many allocations together with two traversals. We automate using this pattern similar to how we handle automate other scheduling of traversals (Chapter 4), as well as with a macro to syntactically hide part of the encoding.

Demonstrating the technique, we use it to optimize the hand-off between layout and rendering in our graphics pipeline. All nodes that render as a circle will call some form of drawCircle in Figure 3.9a. Depending on the size of the circle, which is computed as part of the layout traversals, a different amount of memory must be allocated for recording its vertices: bigger circles need more vertices to describe their perimeter. Once the shapes are thus tessellated, the rendering engine can then paint them on the screen. As we now outline, we use the optimization to vectorize the memory allocation and computation of positions that, together, dominate tessellation.

Staged Parallel Memory Allocation

We stage the use of dynamic memory into four logical phases:

1. Parallel request (bottom-up tree traversal to gather memory sizes)
2. Physical memory allocation
3. Parallel response (top-down tree traversal to scatter buffer offsets)

```
1  CBOX → BOX1 BOX2
2  {
3    ...
4    CBOX.render =
5      drawCircle(CBOX.x, CBOX.y, CBOX.radius)
6        + drawCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
7  }
```

(a) **Call into inefficient library.**

```
1  CBOX → BOX1 BOX2
2  {
3    ...
4    CBOX.sizeSelf =
5      allocCircle(CBOX.x, CBOX.y, CBOX.radius)
6        + allocCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
7    CBOX.size = CBOX.sizeSelf +BOX1.size + BOX2.size;
8    BOX1.buffer = CBOX.buffer + CBOX.sizeSelf;
9    BOX2.buffer = BOX1.buffer + BOX1.size;
10   CBOX.render =
11     fillCircle(CBOX.x, CBOX.y, CBOX.radius, CBOX.buffer)
12       + fillCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5,
13             CBOX.buffer + allocCircle(CBOX.x, CBOX.y, CBOX.radius));
14 }
```

(b) **Macro-expanded calls into staged library**.

```
1  CBOX → BOX1 BOX2
2  {
3    ...
4    render =
5        @Circle(CBOX.x, CBOX.y, CBOX.radius)
6        + @Circle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
7  }
```

(c) **Sugared calls into staged library**.

Figure 3.10: **Use of dynamic memory allocation in a grammar for rendering two circles.**

4. Computations that consume dynamic memory (arbitrary parallel tree traversals)

The staging allows us to parallelize the request and response stages as tree traversals. The actual allocation of physical memory in stage 2 is fast because it is a single call in the global context. Figure 3.11 shows the dynamic data dependencies and two parallel tree traversals for an instance of staged parallel memory allocation.

We manually split library functions that perform dynamic memory allocation into two new functions: the allocation request (Figure 3.9b) and the memory use (Figure 3.9c). The transformation was not onerous to perform on our library primitives and, in the future, might be automated. Invocations of the original call in the attribute grammar must be rewritten to use the new forms. For example, drawing two circles (Figure 3.10a) is split into calls for allocation requests, buffer pointer manipulation, and buffer usage (Figure 3.10b). The transformation increases memory consumption constants due to bookkeeping of allocation sizes.

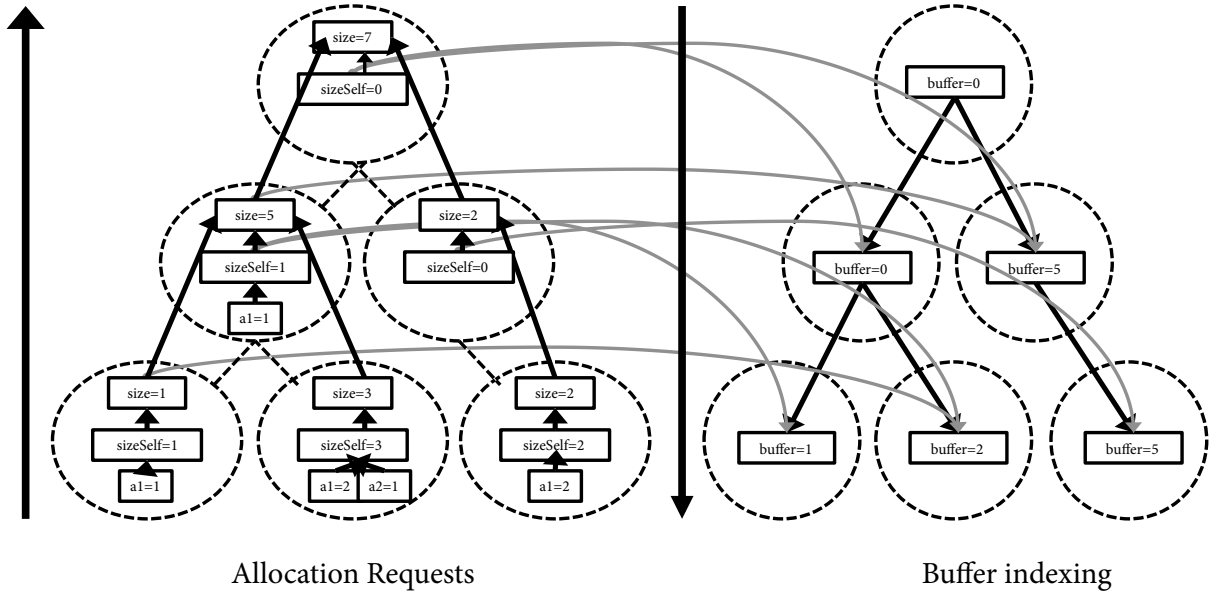


Figure 3.11: **Staged parallel memory allocation as two tree traversals.** The first pass is a parallel bottom-up traversal that computes the sum of allocation requests, and the second pass is a parallel top-down traversal that computes buffer indices. Lines with arrows indicate dynamic data dependencies.

The result of our staging is three logical parallel traversals – allocate, distribute, and use – and implementations generally merge the latter two. The first pass is bottom-up, similar to a prefix sum: each node computes its allocation requirements, adds that to the allocation requirements of its children,and then the process repeats for the next level of the tree. The sizeSelf and *size* attributes are used for the first pass. Once the cumulative memory need is computed, a bulk memory allocation occurs, and then a parallel top-down traversal assigns each node a memory span from buffer to buffer + selfSize. Finally, the memory is ready for use by actual computations in parallel passes. Memory use can occur immediately upon computation of the buffer index, so the last two logical stages are safely merged into one pass for rendering.

Automation with Automatic Scheduling and Macros

Manually manipulating the allocation requests, buffer pointers, and staging is error prone. We eliminated the problems through two automation techniques: automatic scheduling to enforce correct parallelization, and macro expansion to encapsulate buffer manipulation.

To enforce proper parallelization, we relied upon our synthesizer (Chapter 4) to schedule the calls. If the synthesizer cannot schedule allocation calls and buffer propagation, it reports an error. Our insight is that, implicit to our staged representation, we could faithfully ab-

stract the memory manipulations as foreign function calls. Our synthesizer simply performs its usual scheduling procedure.

To encapsulate buffer manipulation, we introduced the macro "@". Code that uses the macro is similar to code that assumes dynamic memory allocation primitives: the slight syntactic difference can be seen by comparing Figure 3.10a and Figure 3.10c: calls to drawCircle are now to @drawCircle. Our macros, implemented in OMetaJS (Warth and Piumarta, 2007), automatically expand into the form seen in Figure 3.10b.

Our use case only required one allocation stage, but multiple allocation stages may be needed. For example, a final logging stage might be added that should run after all other computations, including rendering. However, the "@" calls described above expand to contribute to one attribute (size): no allocation is made until all of the sizes are known, which prevents making an allocation after using dynamic memory. To support multiple allocation stages, the "@" macro could be expanded to include logical group names: @[render]Circle (...) would contribute to sizeRender, @[log]error (...) to sizeLog, and @[render,log]Strange (...) to both sizeRender and sizeLog. Parallel traversals would be created for each logical name, and the synthesizer would be responsible for determining if the traversals can be merged in the final schedule and implementation.

3.6 Evaluation: Layout as Structured Parallel Visits

We show that our static language of parallel schedules is expressive enough to support common layout tasks. Not all computations can be expressed as a statically bounded number of tree traversals, such as fixed point computations, so this result is not obvious. In particular, we show how to parallelize document layout (box models and nested text), table layout (user interfaces and data tables), and rendering (tessellation). Our document layout and table examples describe supporting a subset of CSS. Our rendering example highlights optimizing dynamic memory allocation on a GPU. The attribute grammars in the appendix include sketches (Chapter 4) of the schedules described here.

Box Model

Document languages provide nested box models where intermediate nodes are boxes and leaf nodes are text and images. For example, a box may represent a page, column, or paragraph. The `H-AG` example provides the basic insight, except a language such as CSS extends it with features. Of most relevance to parallelization, we describe supporting the following common features with non-trivial data dependencies:

- Intrinsic preferences. Document content leads to intrinsic preferences, such as having a box big enough to contain its content. These must be combined with external constraints, such as overriding preferences set by the designer on the element or its container.

- Relative positioning. Based on the size preferences of a node and its content, the content must be positioned relative to each other and the node.
- Absolute positioning. Based on the relative positioning between a node and its parent, transitive reasoning must be applied to position the node relative to the tree's root node.

Our static box model schedule loosely corresponds to the above list by devoting 1-2 parallel passes for each item.

We stage the computations with the following sequence of parallel traversals (Appendix A.5):

1. **Bottom-up: intrinsic widths and concrete overriding constraints.** For example, the intrinsic width of a horizontal box is the sum of intrinsic widths of its children. If the user specifies a concrete width value such as 2 pixels, that value is used instead.
2. **Top-down: percent widths.** Constraints such as a width being a percent of its parent are computed next. Notably, the CSS standard defines percent widths that cannot be computed at this point as being undefined. The definition by the CSS standard makes whatever interpretation we use safe.
3. **Bottom-up: heights and relative positioning.** Once the size of a node's children is known, their placement relative to the node can be computed. For example, a horizontal box would place them side by side, and a vertical box would stack them. Likewise, the relative positioning of a node's children, their heights, and any overriding user constraints are sufficient for computing the node's height.
4. **Top-down: absolute positioning.** When the absolute position of a node becomes available, the absolute positions of its children may be computed. The process proceeds recursively.

Nested Text

Our core approach to supporting nested text is described in Section 3.2, and the code is part of Appendix A.5. As a reminder, the problem is to perform word-wrapping on subtrees such as paragraphs with stylized text. The idea is to identify subtrees that require sequential evaluation but can be computed in parallel with other subtrees. Given the basic insight of performing such a nesting, we use our tool to design and verify the schedule.

A non-obvious aspect of using nested traversals for text layout is that we use the nesting for just one pass. The computations relating to text layout span several tree traversals. The intrinsic and computed width passes execute using the parallel traversals described above. We only use the nesting for height and relative position computations. Thus, our strategy of using nested traversals achieves coarse-grained parallelism for the traversal with the difficult word-wrapping dependency, and features the usual fine-grained parallelism for all others.

Grids

We scheduled the automatic layout algorithm used in CSS and HTML as parallel tree traversals. Section 2.4 describes the functional specification, and the code is available in Appendix A.2. The primary dependencies challenging parallelization relate to supporting topological traversals over a DAG rather than a tree because cells have two parents: the row and the column. In a top-down traversal, both the row and column should be visited before the cell. Our solution for parallelization is a level-synchronous breadth-first evaluation order. Finally, our example propagates information between rows and columns using several intermediate parallel traversals. A nested traversal or the ability to reason about attributes of grandchildren rather than just children helps eliminate those traversals.

SIMD Rendering through Staged Memory Allocation

We evaluated three dimensions of our staged memory allocation approach: flexibility, productivity, and performance. First, it needs to be able to express the rendering tasks that we encounter in GPU data visualization. Second, it should provide some form of productivity benefit for these tasks. Finally, the performance on those tasks must be fast enough to support real-time animations and interactions of big datasets.

Productivity

Productivity is difficult to measure. Before using the automation extensions for rendering, we repeatedly encountered bugs in manipulating the allocation calls and memory buffers. The bugs related both to incorrect scheduling and to incorrect pointer arithmetic. Our new design eliminates the possibility of both bugs.

One weak productivity measure is of how many lines of code the macro abstraction eliminates from our visualizations. We measured the impact on using it for three of our visualizations. The first visualization is our HBox language extended with rendering calls, while the other two are interactive reimplementations of popular visualizations: a treemap and multiple 3D line graphs.

Table 3.1: **Lines of code before/after invoking the "@" macro.**

Visualization	Before (loc)	After (loc)	Decrease
HBox	97	54	44%
Treemap	296	241	19%
GE	337	269	20%

Table 3.1 compares the lines of code in visualizations before and after we added the macros. Using the macros eliminated 19–44% of the code. Note that we are *not* measuring the macro-expanded code, but comparing to code that a human wrote.

As shown in Figure 3.10, switching to manually staged allocation introduces boilerplate code. With the macro, porting unstaged functional graphics calls to use the new staged library effectively only requires renaming called methods. The "@" macro eliminates 19–44% of the code that would have otherwise been introduced and completely eliminates two classes of bugs (scheduling and pointer arithmetic); the productivity benefit is non-trivial.

Discussion

One concern we had is the completeness of the specification formalism with respect to parallel traversals. By restricting specifications to verifiable attribute grammars, we disallow safe programs that could be manually expressed with tree traversals. For example, our CSS schedule contains traversals that a pen-and-paper proof can safely fuse by performing the equivalent of *loop skewing* for allowing a node to read and write to its grandchildren in the same traversal. An important future direction would therefore be to improve the reasoning over attribute grammars while maintaining the set of parallel traversal types.

3.7 Related Work

This section relates to several significant bodies of work: static parallel scheduling of attribute grammars, static parallel scheduling of traversals over data structures, optimization of layout languages, and parallel rendering.

Attribute Grammars

Some of the earliest results for attribute grammars were in how to safely and effectively statically schedule them. Our checking algorithm is similar to the inference algorithm of Bochmann (1976). However, we decouple checking a schedule from searching for it and use axiomatic reasoning to make the approach modular over different types of schedule constructs. We defer questions of automatic parallelization to the next chapter.

Alblas (1991) surveyed various methods and provides a taxonomy of sequential traversal types. Many of the results apply to our static scheduling approach and our case study of layout languages:

- **Single and multi-pass:** some computations can be performed using one traversal. Klein and Koskimies (1990) and Noll and Romanith (1996) identify the opportunity of fusing parsing with evaluation for the efficient class of single-pass languages. For example, LL(1) languages support single-pass top-down parsing. If the evaluator only requires one "left-to-right" traversal, the entire computation is single-pass. Likewise, LR(1) languages support bottom-up parsing, which in turn can be fused with "right-to-left" traversals. We faced related questions when examining how to stage layout preprocessing of CSS selector matching and cascading before layout solving (Meyerovich and Bodík, 2010).

- **Bounded traversals:** Some languages may be solved with a static number of traversals. The proof often involves showing that particular attribute types will be solved in particular traversal steps. The community refines this notions with definitions of *purity*, *simplicity*, and *ordering* where ordered grammars subsume the other classes. The traversals in our static scheduling language are *simple.*

- **Rigidity:** Our discussion of *structured* traversals relates to that of Alblas on *rigid* versus *flexible*. We show how to make a composable language of rigid traversals that facilitate effective parallelization.

- **Multi-visit:** Many of the scheduling challenges in attribute grammars stem from allowing nodes to be visited multiple times in the same traversal. For example, using ordered attribute grammars (Kastens, 1980), we can define a traversal where the top subtree is visited many times but descendants are never visited. Optimal static scheduling becomes NP, however, so systems instead perform dynamic scheduling (Jourdan and Parigot, 1991).

Additional topics in static scheduling such as incrementalization and non-local references complement the above techniques.

Our focus on a compositional language of traversals helps unify parallelization techniques. For example, Vogt et al. (1989) proposed executing traversals concurrently such that they operate on independent sets of attributes, while others typically focus on parallelism within a traversal (Jourdan, 1991). We support both, such as would be exercised in schedule "$parPre||parPost$". Likewise, our nested region construct relate to partitioning constructs for load management (Boehm and Zwaenepoel, 1987) and remote reasoning (Reps et al., 1986). We discuss load management in the next chapter as it involves runtime techniques that exploit the structure exposed here. Our axiomatic approach shows how to combine such patterns in a way that provides modular reasoning.

Parallel Traversals over Data Structures

Many researchers examine abstractions for encapsulating parallel strategies for computing over a data structure.

An important type of abstraction is for describing a traversal over a data structure. For example, a skeleton (Matsuzaki et al., 2006a,b) is a traversal over a data structure parameterized by a first-class function. Skeletons do not use knowledge of the computation beyond high-level properties such as associativity. Similar to skeletons, stencils (Datta et al., 2008) typically exploit additional knowledge about the computation such as what nearby nodes will be accessed. Both skeletons and stencils have been applied towards optimizing traversals over a variety of data structures.

Our computations require multiple traversals, not just one. Frameworks such as MapReduce (Dean and Ghemawat, 2008) support creating a pipeline of structured traversals over data structures. MapReduce in particular computes over lists. It automates optimizations

over multiple traversals such as fusing map and reduce calls in order to avoid excessive data movement.

Our focus in this chapter is on designing the language of traversal patterns for trees, rather than the grids of stencils and lists of MapReduce. Likewise, the above systems require programmers to manually schedule their computations as a composition of traversals, while the next chapter shows how to automatically do it.

Prountzos et al. (2012) examine the language of schedules for a breadth-first graph traversal. While we support choosing between multiple fixed implementations of the same traversal, such as implementing a parallel preorder traversal with a sequential variant or an inorder traversal, they provide controls for creating new variants, e.g., for task queue prioritization. Our approaches are complementary in the sense that our language might be extended to support decorating a traversal instance with their scheduling options.

Automatic Layout Optimization

Brown (1988) and others have previously proposed parallelizing document layout, though we appear to be the first to implement it. A key stumbling block was correctly identifying latent parallelism in layout language specifications. For example, challenging the intuition of Brown (1988) that different pages might be solved in parallel is the data dependencies that we identified that span attributes of different pages. To break them, we introduced multiple traversals and nested traversals. Mai et al. (2012) instead use an unsound heuristic to perform an unsafe data partitioning and accept that layouts may appear incorrectly. Their approach may be made safe by asking layout authors to manually identify independent regions; this does, however, introduce the additional burden of requiring developers to manually solve constraints near boundary regions. More fundamentally, the approach limits the granularity of exposed parallelism.

Industrial layout systems typically use hand-written code. We hypothesize that this is due to challenges stemming from the requirement for a combination optimizations while still supporting many features and maintainability. For example, discussions with commercial browser developers show the need for both parallel evaluation for bulk workloads and incrementalization for reevaluation (*reflow*). Automating one optimization is insufficient if the technique cannot express all the language features or compose with other optimizations. We demonstrated that many features can be expressed and ways of integrating foreign functions when they cannot be supported natively, and attribute grammar literature demonstrates the feasibility of many other optimizations such as incrementalization (Demers et al., 1981).

Some experimental systems now support parallel layout (Burckhardt et al., 2011) by adapting the techniques we introduced (Meyerovich and Bodík, 2010).

Parallel Tesselation

Parallel tessellation, also known as mesh refinement, is a long-standing problem. Close to our work is that of Shiue et al. (2005), who perform subdivision on a GPU. Follow-up research by

Patney and Owens (2008) describe related algorithms in terms of prefix sum and reduction primitives. We generalize the underlying observation to batching dynamic memory allocation and provide automation support.

Chapter 4

Parallel Schedule Synthesis

Programmers struggle to map application logic into parallel algorithms. Going beyond the automatic schedule verification of the Chapter 3, we now examine how to automatically generate a schedule. Consider two of the decisions that a programmer faces in manually designing a schedule:

- **Scheduling a single traversal.** Many computations contain sequential dependencies between nodes. One correct traversal over the full tree might then be sequential. However, if the sequential dependencies can be isolated to a subtree, an overall parallel traversal would be possible if it invokes a sequential traversal for just the isolated subtree. Whether such isolation is always possible is not obvious.
- **Scheduling multiple traversals.** Programs such as browsers perform many traversals. Traversals might run one after another, concurrently, or be fused into one. These choices optimize for different aspects of the computation. Running two traversals in parallel improves scaling, but fusing them into one parallel traversal avoids overheads: the choice may depend on both the hardware and tree size. Which traversal sequence to use is not obvious.

These decisions explode the space of schedules. Today, programmers manually navigate the space by selecting a parallel schedule, judging its correctness, and comparing its efficiency to alternative schedules. Each task consumes time: programmers globally reason about dependencies, develop prototypes for profiling, and whenever the functional specification changes, restart the process.

This chapter explores the design of an attribute grammar synthesizer and its implications on the design of the attribute grammar language. We examine several questions:

- What programming constructs are enabled by schedule synthesis?
- What is an algorithm to *quickly* find a *correct* schedule?
- If multiple schedules are possible, how do we find a *fast* one?

The following sections explore each question in turn.

4.1 Computer-Aided Programming with Schedule Sketching

Automatic parallel schedule synthesis enables new abstractions for parallel programming. The utility of these constructs is not immediately obvious. Because automation tools will automatically find a parallel schedule, a natural conclusion would be to assume that the programming interface should hide all parallelization concerns and rely upon automatic parallelization internally. We found this to be largely true when writing small amounts of declarative data visualization code. However, when parallelizing the larger and more complicated CSS layout language, we encountered cases where the visualization designer needed to guide (or be guided by) the automation procedure. Likewise, we encountered the need for one programmer to communicate parallel structure to another. Automatic parallelization is insufficient in that it hides all parallelization details and controls, yet manual scheduling is too low-level and brittle.

Our solution is to provide a *sketching* construct for specifying constraints on the schedule that the automatic parallelization algorithm must respect. The programmer chooses which terms in a schedule to specify and relies upon the synthesizer to fill in the rest. We routinely sketched schedules in order to *override schedule selection*, *test* and *debug* parallelization ideas, and *enforceably communicate parallelization decisions* when sharing code with others. Discussed in Section 4.3, providing a sketch also speeds up compilation because it changes the synthesis problem into a verification one.

We revisit the specification of `H-AG` to demonstrate the sketching construct and its use for the above scenarios. First, in response to a low amount of memory size on prospective hardware, a programmer may specify a longer schedule with a smaller set of attributes to compute in each one. Compare the three following schedule sketches:

$$?hole_1 \tag{4.1}$$

$$\textbf{parPost}\ ?hole_2\ ;\ \textbf{parPre}\ ?hole_3 \tag{4.2}$$

$$(\textbf{parPost}\ ?hole_4\ ;\ \textbf{parPost}\ ?hole_5)\ ;\ ?hole_6 \tag{4.3}$$

The first specification leaves a *hole* for the entire schedule. The synthesizer fills in every hole with a valid schedule term so that the resulting schedule is correct. The entire first schedule is left as a hole, which is equivalent to requesting fully automatic parallelization. The second specification hardcodes the traversals but leaves holes for the attributes to schedule for each one. The final schedule sketch splits the parPost traversal in two in order to decrease the memory consumption in the first traversal. Like the second sketch, it does not specify the attributes, and like the first sketch, it does not specify the sequence of traversals to place at the end of the schedule.

The ability to run a sketch through the synthesizer enables several forms of parallel program debugging. First, the synthesizer rejects programs that it cannot parallelize, so programmers can test their intuitions with sketches. For example, they could test the validity of the above idea of splitting apart the first parPost traversal. We could more explicitly test the underlying insight that the w and h attributes are separable:

$$(\textbf{parPost}\ \{w\}\ ;\ \textbf{parPost}\ \{h\})\ ;\ ?hole_6$$

The synthesizer fills in $?hole_6$ to yield a complete schedule. It outputs an error if it cannot: the longest schedule prefix of traversals it could schedule. For the above example, the error distinguishes two possible mistakes. First, if it fails with a prefix containing parPost { w } ; parPost { h }, the first traversal can be split but the rest of the schedule has an unsatisfiable dependency. Otherwise, the output prefix is empty and the traversals could not be split. We found the ability to test scheduling ideas to be particularly useful, e.g., in determining partitions for nested text.

We provide another mechanism for debugging. The programmer may ask the synthesizer to *enumerate* all valid solutions for a schedule sketch. The previous examples restricted themselves to only asking for one completion. However, for `H-AG`, the space of valid schedules is small enough that programmer could manually page through all possibles schedules.

As our attribute grammars grew, we wrote sketches to help share code between programmers. Consider a program with a sketch such as the above. For a grammar associated with it, a programmer now knows the desired parallelization scheme. Furthermore, the synthesizer checks that edits to the functional specification do not violate the schedule. For example, the synthesizer would detect the addition of a feature that requires the addition of an extra traversal, or if it serializes a parallel one. We typically ignored changes that do not change the traversal sequence and applied more careful reasoning whenever a sketch was violated. In this way, the ability to communicate and enforce schedule specifications helped separate concerns between defining layout feature logic and optimizing layout scheduling.

4.2 Generalizing Holes to Syntactic Unification

We provided a more expressive variant of holes for cases that require additional control. For example, we may want to specify that both the width and height are computed in the first traversal over a tree. The programmer should not have to specify the relative order of attributes for every type of node that computes them. Instead, we generalized the sketching construct to syntactic unification over scheduling terms.

Programmers may specify constraints over schedule terms. For example, the following specification declares that the first traversal of a sequence computes the width and height

attributes, but it does not define their relative order:

$$
\begin{aligned}
&member(\text{w}, ?hole_2), member(\text{w}, ?hole_3),\\
&member(\text{h}, ?hole_2), member(\text{h}, ?hole_3),\\
\textbf{\textit{Sched}} = \ &[\ [?hole_1,\ [[\text{HBox}, ?hole_2], [\text{VBox}, ?hole_3]]],\\
&\textbf{seq},\\
&[\textbf{parPost},\ ?hole_4]\ ]
\end{aligned}
$$

Term $?hole_1$ will unify with a traversal type and $?hole_2$ and $?hole_3$ will unify with a sequence of attributes that includes w and h. Finally, $?hole_4$ will unify with another sequence of terms where each specifies a node type and the sequence of attributes to schedule for it. Note the change in syntax.

Our scheduling language is an embedded domain specific language in Prolog (Colmerauer, 1990). The language of constraints is arbitrary Prolog. Thus, in the above example, Sched is a named Prolog variable that must be unified with the schedule constraints and the attribute grammar's functional dependencies. Likewise, unnamed variables $?hole_1$, $?hole_2$, and $?hole_3$ must unify with a correct schedule. Our system provides a library of traversal types such as parPost and combinators such as seq. The attribute grammar introduces attribute terms such as w and h. The programmer then uses built-in Prolog predicates to constrain the schedule, e.g., by using member for list membership. Likewise, they may use Prolog's "," operator for conjunction and ";" for disjunction.

We made several notable uses of the extended sketching constructs:

- **Attribute sets.** As in the above example, we specify an *unordered set* of attributes for a traversal rather than an *ordered sequence*. The synthesizer determines the order and any additional attributes. Furthermore, as different classes implementing the same interface share attributes of the same name, we wrote helper functions for specifying the interface attribute once rather than all of its instances in classes.

- **Requiring parallelization.** We may specify that a traversal type unifies with a parallel form:

  $$
  \begin{aligned}
  &(?hole_1 = \textbf{parPost}; ?hole_1 = \textbf{parPre}),\\
  &\textbf{\textit{Sched}} = [\ [?hole_1,\ ?hole_2],\ \textbf{seq},\ [\textbf{parPost},\ ?hole_3]]
  \end{aligned}
  $$

  The sketch specifies a sequence of two traversals where the first traversal type ($?hole_1$) unifies with either a parPost or parPre traversal. The schedule does not specify what attributes are computed within the first traversal ($?hole_2$). Furthermore, instead of manually specifying the choice for every pass, we wrote a function that does so automatically.

- **Nesting.** We use predicates to test the validity of partitioning into nested traversals. The challenge is to minimize the number of traversals put into a sequential partition.

For example, if we thought a node belonged in a parallel partition, we would include that in the sketch;

$$member([\text{HBox}, ?hole_1], TopDownVisits),$$
$$\textbf{\textit{Sched}} = [\ [\textbf{nested}, [\textbf{parPre}, TopDownVisits],\ ?hole_2]\ |\ ?hole_3]$$

The schedule specifies that the first traversal is nested within an overall parallel preorder structure. The preorder portion must handle HBox nodes and may include other as well. The other partitions are defined by $?hole_2$, which has no additional constraints. The specification leaves remaining traversals unconstrained by $?hole_3$.

- **Schedule heuristics.** For a simple optimization heuristic, we biased parallel scheduling to use an alternating sequence of parPre ancd parPost traversals. The intuition is that long-running dependencies can often be satisfied under these two traversals so that the shortest schedule would be such an alternation. Less obviously, some dependencies require repetition of the same traversal pattern, so the full heuristic is to bias to use of the alternate of whatever worked for the previous traversal (and otherwise prioritize any other parallel traversal).

In all of the above cases, reasoning is in terms of the syntactic form. For example, the alternating traversal heuristic biases towards one traversal based on equality with the syntactic value of the the previous one. Richer forms of unification that extend beyond syntax may be applicable. As is, syntactic unification for guiding schedules already supports several key tasks.

4.3 Fast Algorithm for Schedule Synthesis

Our synthesizer takes an attribute grammar and a sketch as input, and outputs a set of schedules. We designed it to support multiple traversal types, multiple solutions, and rich attribute grammar and schedule sketching languages. Our initial implementation used the dependency analysis of Kastens (1980), but it was difficult to implement and extend. Our new algorithm optimizes for modularity and speed by using the following design:

Simple enumerate-and-check The algorithm enumerates schedules and checks which are correct. Checkers examine the use of individual traversal types and traversal compositors, and we wrote them to function independently of one another. Enumeration is simply syntactic. Combined, adding a new traversal type involves writing a checker and binding it to the proper place.

Optimization Naïve enumerate-and-check is too slow. Without significantly changing the interface for adding checkers, we optimize synthesizing one schedule to be $O(n^3)$ in the number of attributes. Some features are still slow, such as nested traversals, so we introduced the optimizations of incrementalization, greediness, and greedy sketch unification.

```
 1  parPre{x,y,w,h}                              incorrect: unsat {x,w,h}
 2  parPre{y}                                    correct: continue
    ... /* expand subtree to schedule x, w, h */ ...
 3  parPost{x,y,w,h}                             incorrect: unsat {x,y}
 4  parPost{w,h}                                 correct: continue
 5    _ ; parPre{x,y}                            correct: complete
 6    _ ; parPost{x,y}                           incorrect: unsat {x,y}
 7    _ ; (parPre{x} || _)                       correct: continue
 8      _ ; (_ || parPre{y})                     correct: complete
 9      _ ; (_ || parPost{y})                    incorrect: unsat {y}
10    _ ; (parPre{y} || _)                       correct: continue
11      _ ; (_ || parPre{x})                     correct: complete
12      _ ; (_ || parPost{x})                    incorrect: unsat {x}
13    _ ; (parPost{y} || _)                      incorrect: unsat {y}
14    _ || parPre{x,y}                           incorrect: unsat {x}
15    _ || (parPre{y} ; _)                       correct: continue
16      _ || (_ ; parPre{x})                     incorrect: unsat {x}
17      _ || (_ ; parPost{x})                    incorrect: unsat {x}
    ...
18  parPost{w}                                   correct: continue
19    _ || parPre{x,y,h}                         incorrect: unsat {x,h}
    ...
```

Figure 4.1: **Trace of synthesizing schedules for** `H-AG`. Note that scheduling of "||" does not use the optional greedy heuristic.

The Algorithm

We first discuss optimizations for finding one correct schedule before considering finding many. Figure 4.1 demonstrates an algorithm trace for enumerating schedules of `H-AG`. Figure 4.2 shows the full algorithm.

Synthesizing one schedule is $O(A^3)$ in the number of attributes. The algorithm finds an increasingly long and correct prefix of the schedule (*prefix expansion*). At each step, it tries different suffixes until one succeeds, where a suffix such as "**parPre**{x,y}" is a traversal type and attributes to compute in it. When a correct suffix is found, it is appended to the prefix and the loop continues on to the next suffix. Finding one suffix involves trying different traversal types, and for each one, different attributes. Only the suffix needs to be checked (*incremental checking*), and checking a suffix is fast (*topological sort*). Finally, finding a set of attributes computable by a particular traversal type only requires $O(A)$ attempts (*iterative refinement*).

We consider each optimization in turn:

1. **Prefix expansion.** The synthesizer searches for an increasingly large *correct* schedule prefix. Every line of the trace represents a prefix. If a prefix is incorrect, no suffix will yield a correct schedule. Therefore, the only prefixes that get expanded are those that succeed (lines 2, 4, 7, 10, 15, 18).

   To synthesize only one schedule, only one increasingly large prefix is expanded. Line 2 has a correct prefix, so only "**parPre**{y}" would be explored. Either no schedule is

possible at all, or if there are any, one is guaranteed to exist in the expansion. In this case, "**parPre**{y} ; **parPost**{w,h} ; **parPre**{x}" would be found.

2. **Incremental checking.** Line 4 checks prefix "**parPost**{w,h}" for attributes "w" and "h." Therefore, lines 5-17 can check the suffix added at each line without rechecking "**parPost**{w,h}".

3. **Topological sort.** We optimize checking a suffix by topologically sorting the dependency graph of its attributes (rule $\textbf{c}check_\beta$ in the next subsection). Topologically sorting a graph is $O(V + E)$. It is $O(A)$ in this case because $V = A$, and as the arity of semantic functions is generally small, E is $O(A)$.

4. **Iterative refinement.** The algorithm iteratively refines an over-approximation of what attributes can be computed in a suffix by removing under-approximations of what cannot. For example, the check in line 1 for **parPre**{x,y,w,h} fails with error {x,w,h}, which details the attributes with unsatisfiable dependencies. Computing fewer attributes cannot satisfy more dependencies; therefore, no subset of {x,w,h} has satisfiable dependencies either. Therefore, the next check is on a set without them: {y}.

   Subtraction of attributes repeats at most A times before finding a solution or terminating on the empty set. Checking one refinement invokes the $O(A)$ topological sort. Put together, finding the attributes computable by a suffix is $O(A^2)$.

Because every traversal computes at least one attribute, there are at most A traversals. A constant number of traversal types are examined for each suffix, and synthesizing each one is $O(A^2)$. Synthesizing one schedule is therefore $O(A^3)$. The *greedy sketch unification* optimization presented in the next section may further optimize the synthesis of a single schedule.

4.4 Schedule Enumeration

We provide and optimize the ability to examine many schedules. Our approach aids several scenarios: picking a fast schedule when many are possible, scheduling language extensions that otherwise resist fast synthesis, and improving synthesis time when partial schedule knowledge is known.

We consider each scenario in turn:

Autotuning There may be an exponential number of safe schedules, and the choice of a fast one is non-obvious. For example, shorter schedules incur less traversal overhead, but also generally expose less parallelism. Likewise, a short sequence of parallel traversals may perform worse than a long sequence when performed on hardware with limited memory. By enumerating all schedules, we can build an *autotuner*: an autotuner runs performance tests

to pick the best configuration for a particular environment. As there may be an exponential number of schedules, we must somehow optimize the enumeration of those to test.

Scheduling extensions. We provide optional scheduling language extensions, and fast synthesis in their presence requires optimization. For example, nested traversals require partitioning the set of nodes into distinct regions, but many partitions are possible. Unfortunately, how one partition fails does not inform the guess for the next, and thus does not enjoy the monotonicity property we used for iterative refinement. Brute force synthesis of partitions is slow.

Faster synthesis. If the programmer provides knowledge of the schedule, such as when recompiling the grammar occurs, synthesis should execute faster. In the limit, providing full schedule knowledge should reduce the cubic time of synthesis to the $O(A \texttt{ log } A)$ for verification.

We introduce several optimizations that, together, address the above scenarios. They optimize for when multiple schedules may be valid schedules, and except for backtracking may also improve the process of finding one schedule.

- **Backtracking.** To emit multiple schedules, we extend prefix expansion to also perform backtracking. After a schedule is fully completed or a suffix is rejected, the synthesizer backtracks to the most recent correct prefix. For example, line 8 of Figure 4.1 reaches a complete and correct schedule. Backtracking returns to the earlier correct prefix of line 7 and tries the alternative suffix of line 9.

- **Greedy sketch unification.** We use sketches to prune the search. For example, sketch "**parPost** $?hole_1$ || $?hole_2$" enables skipping lines 1-3 because they do not start with a **parPost** traversal. Lines 5-13 could also be skipped because the compositor is not "||".

  A sketch that provides a full schedule reduces synthesis to verification. Sketching also enable features that otherwise require exponential search to still synthesize in $O(A^3)$. For example, scheduling nested regions is exponential in the number of productions, but if just the production partitioning is sketched, synthesis for the remaining schedule terms is still only $O(A^3)$.

- **Greedy attribute heuristic.** For any schedule "p ; q", solving fewer attributes in p will not enable solving q with fewer traversals. Thus, to minimize the number of traversals, all such subsets are pruned. For example, as line 4 found **parPost**{w,h}, line 19 skips "**parPost**{w} ; _ " and proceeds to "**parPost**{w} || _".

  Greediness reduces enumerating all schedules to only being exponential in the number of traversals. This is significant because our schedule for CSS has only 9 traversals, for example.

```
1  def synthFast(sketch):
2    yield synth(∅, Attributes, sketch)

4  def synth(prev,rest,sketch):
5    choose  ⊗ ∈ { ";", "||" }
6    if ⊗ = ";":
7      choose α ∈ { "parPre", "parPost", ... }
8      A := iterativeRefine(α, prev, rest)
9      if A = rest:
10       unify(sketch, α A)
11       yield α A
12     else if A = ∅:
13       backtrack
14     else:
15       unify(sketch, α A ; rhs1)
16       yield α A ; synth(prev ∪ A, rest − A, rhs1)
17   else:
18     unify(sketch, lhs2 || rhs2)
19     choose A ⊂ rest
20     p := synth(prev, A, lhs2)
21     q := synth(prev, rest   A, rhs2)
22     yield p || q

24 def iterativeRefine(α, prev, rest):
25   overapproxA = rest
26   do:
27     X = check_α(prev, overapproxA)
28     overapproxA = overapproxA − X
29   while X ≠ ∅
30   yield overapproxA
31   if nonGreedy:
32     choose overapproxA′ ⊂ overapproxA
33     yield iterativeRefine(α, prev, overapproxA′)
```

Figure 4.2: **Optimized synthesis algorithm.** Lines 10,15,18: early unification with sketches. Lines 8,27: incremental checking. Line 26: iterative refinement. Line 31: toggle minimal length schedules. Lines 12,28: pruning of traversals with unsatisfiable dependencies.

> Unlike our other optimizations, this one is a heuristic and eliminates correct results. While greedy heuristic is guaranteed to return at least one schedule should any exist, it may prune schedules useful for autotuning and other tasks.

In summary, synthesizing one schedule in our base language is $O(A^3)$, but emitting all of them is exponential. Likewise, scheduling language extensions such as nested traversals still support fast synthesis of surrounding terms when guided by sketches. Our optimizations optimize the process, such as by reducing synthesis complexity to that of verification when increasingly detailed sketches are provided.

name	loc	1st	sketch	found	avg
hbox++	305	5.6s	9.6s	54	2.7s
spiral	144	0.7s	0.9s	12	0.4s
votes	327	15.4s	22.0s	36	8.0s
css	1132	1919.6s	65.1s	100	445.4s

Figure 4.3: **Synthesizer speed.** `1st` is the time to first schedule without using a sketch, `sketch` is the time to first schedule using a sketch of the traversal sequence, `found` is the number of schedules found, and `avg` is the average time to find a sketch.

4.5 Evaluation

We evaluated the automation capabilities of our schedule synthesizer for our case studies of data visualization and document layout. First, we examined whether the synthesizer could find parallelism and how much guidance it needed. Second, we evaluated whether our synthesis algorithm can achieve interactive or same-day compile times. Finally, we examined the quality of schedules: we measured the benefit of autotuning and the cost of our greedy heuristic.

Automatic Parallelization

We first evaluate whether the synthesizer automatically detected parallelism and the amount of schedule guidance we provided.

For all of the data visualizations (tree map, single and multiple time series, hbox, and sunburst), we successfully relied upon the synthesizer to automatically find parallelism. We performed an iterative design process where we would alternate between adding code to an attribute grammar and checking that the compiler could automatically parallelize it. Once the compiler accepted one functional specification, we would extend the specification with the next feature. When satisfied with the visualization, we would specify the sketch of parallel traversals but only for communicating requirements to future programmers.

The CSS specification required guidance. On its own, the synthesizer would find a sequence of parallel preorder and postorder traversals. The exception is one traversal that requires nested partitions for parallelization. To improve synthesis times, we specified the structure of that traversal. Furthermore, due to the many cross-cutting data dependencies in CSS, we specified schedule sketches throughout the design process. The sketches ensured that extensions to the functional specification did not violate our understanding of the parallel behavior.

Figure 4.3 shows the number of lines of declarative code for each specification. The generated code was over a magnitude more depending on the compiler backend. The number of parallel traversals ranged from 3 to 9.

Synthesis Speed

Performing synthesis in less than a minute enables interactive use by programmers, and even faster times would support runtime compilation. We measured the time to synthesize several attribute grammars. Figure 4.3 shows the lines of code for each one and various timings on a 2.66GHz Intel Core i7 with 4GB of RAM.

Generally, synthesizing a schedule, whether an arbitrary one (`1st`) or from a sketch specifying the traversal sequence (`sketch`), takes less than 30 seconds. The exception was CSS, which was still fast but not as fast; it is discussed later.

Emitting all schedules is even faster per emitted schedule (`avg`) than just finding the first. While the total time to emit all schedules can be slow, we note that enumeration is for offline autotuning. Finally, the greedy heuristic was necessary for enumerating schedules. Even after one day of running the non-greedy algorithm for CSS, most of the greedy CSS schedules were still not emitted.

Overall, we see that synthesis is fast enough for interactive use by the programmer.

Autotuning

We evaluated schedule autotuning speedups for `hbox++` on the same machine:

Comparing greedy schedules We enumerated greedy schedules for `hbox++` and compared performance on 1 and 2 cores. The relative standard deviation for performance of different schedules (σ/μ) is 8%. The best schedules for 1 and 2 cores are different. Swapping them leads to 20-30% performance degradation, and the difference between the best and worst schedules for the two scenarios are 32% and 42%, respectively. Autotuning schedules improves performance.

Comparing greedy to non-greedy Our schedule enumeration is not exhaustive because of the greedy heuristic, and, therefore, may miss fast schedules. For a fixed schedule of traversals with a greedy attribute schedule, non-greedy attribute schedules were 0-6% faster. On average, however, non-greedy schedules were 5% slower. Greedy scheduling was a safe heuristic for `hbox++` .

In our case studies, much of the benefit of autotuning derives from trying two greedy schedules: one that starts with a parallel preorder traversal and then alternates with parallel postorder ones, and vice-versa. We did see exceptions, however, such as CSS benefiting from a nested traversal. Likewise, some grammars require repetitions of preorder or postorder traversals due to the lack of a loop skewing optimization for safely fusing them.

4.6 Related Work

The ideas presented in this chapter descend from research in synthesis, logic programming, attribute grammars, and autotuning.

Schedule Sketching

Our approach of identifying the parallelism in a program is more flexible than many parallelism annotations. For example, the *spawn* primitive for work stealing (Blumofe et al., 1995) identifies latent parallelism in a structured program. As in our approach, how to exploit that parallelism is left up to the language implementation. However, unlike our approach, changing the schedule requires refactoring the logical program. We make the parallel schedule an orthogonal specification.

Our schedule sketching construct descends from the program sketching approach of Solar-Lezama et al. (2006). Program sketching constructs take a total functional specification of a program, such as a naı"e implementation, and a partial specification of the desired implementation. A program synthesizer then fills in any partially specified *holes* in the implementation sketch such that they satisfy the functional specification for all inputs. In our case, the declarative attribute grammar defines the total functional specification. The partial implementation is the schedule with holes. The more general approach of Solar-Lezama et al. (2006) only provides correctness guarantees equivalent to bounded model checking and struggles with scaling. We use the restricted domain of attribute grammars to guarantee full correctness and polynomial synthesis times.

Schedule holes generalize to syntactic unification. Our embedded Prolog (Colmerauer, 1990) DSL for specifying additional constraints over schedule holes exploits this connection by reusing Prolog's unifier and standard library.

Recently, Prountzos et al. (2012) has also examined schedule synthesis. The focus is on behavior within a traversal, such as task prioritization. It is complementary to our focus on a compositional language of traversals. In particular, we might refine the definition of a traversal as parPre to also instruct the runtime scheduler to prioritize certain types of nodes.

Schedule Synthesis Algorithm

Using iterative refinement to quickly determine whether a traversal can compute attributes most closely resembles the work of Bochmann (1976). Many later static attribute grammar systems instead use the approach of Kastens (1980). Kastens targeted the bigger class of ordered attribute grammars but at the expense of slower synthesis and less structured traversal patterns. We focused on a smaller class of grammars where each node is visited only once per traversals. Unlike Bochmann, we provide an axiomatic decomposition of the algorithm, which enables composing traversals of different types. We hypothesize that this is why the static systems we examined do not support a variety of schedule constructs.

The greedy heuristic is a generalization of the one used by Kastens.

Autotuning

Many others have examined autotuning parameter values for problems such as FFTs and stencils (Whaley et al., 2001; Datta et al., 2008). When used for tasks such as optimizing the cache block size, the challenge is in limiting the search space size. As explored in Chapter 5, we use traditional autotuning techniques to optimize the implementations of our patterns.

Our work autotunes over schedules rather than just parameters, which provides more opportunities for optimization but also requires additional reasoning. FFTW (Whaley et al., 2001) and PetaBricks (Ansel et al., 2009) select algorithms based on predefined configuration options. Programmers must also state what algorithms to try in those systems. The superoptimization project (Massalin, 1987) instead automates the process by generating functionally equivalent programs based on rewrite rules for a low-level instruction set. Superoptimization requires every intermediate rewrite step to be correct and generally suffers from weak low-level reasoning. Our synthesizer explores correct schedules through a higher-level and more scalable program analysis. Elixir (Prountzos et al., 2012) is more similar in that it infers dynamic task scheduling optimizations for a single traversal. We examine multiple traversals.

Chapter 5

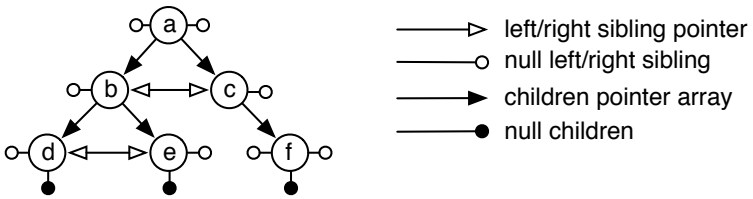
Optimizing Parallel Tree Traversals for MIMD and SIMD

Expressing parallelism is not enough: we need efficient runtime implementations to exploit them. In this chapter, we show how to exploit the parallelism identified within a tree traversal by optimizing for the architectural properties of two hardware platforms: MIMD (multicore) and SIMD (sub-word SIMD and GPU) hardware. Using existing techniques, we saw little-to-no speedups. Our solution was to optimize the schedule within a traversal and the data representation, and in different ways for different types of hardware. We innovated upon known techniques in two ways:

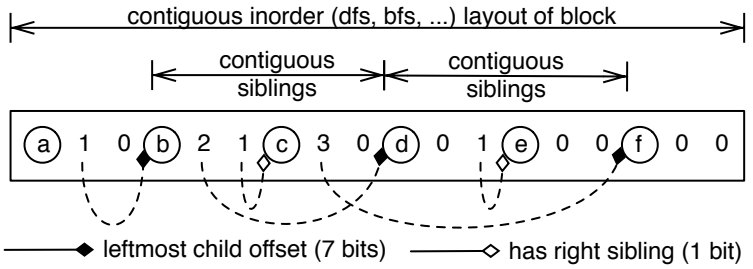
1. **Semi-static work stealing for MIMD:** MIMD traversals should be optimized for low overheads, load balancing, and locality. Existing techniques such as work stealing provide spatial locality and, with tiling, low overheads. However, dynamic load balancing within a traversal leads to poor temporal locality across traversals. The problem is that a processor a node is assigned to in one traversal may not be the same one in a subsequent traversal, and as the number of processors increases, the probability of assigning to a different one increases. Our solution dynamically load balances the first traversal and, due to similarities across traversals, successfully reuses it for all following ones.

2. **Clustering traversals for SIMD:** SIMD evaluation struggles when parallel tasks diverge in instruction selection. Visits to different types of tree nodes yield different instruction streams, so naıve vectorization fails on webpages with high visual variety, for example. Our insight is that similar nodes can be semi-statically identified. Thus, once a tree is available, our scheduler *clusters* nodes into self-similar groups and uses SIMD instructions for nodes in each one.

Our techniques are effective and general. They overcame bottlenecks preventing seeing any speedup from parallel evaluation for webpage layout and data visualization. Notably, they are generic to computations over trees, not just layout. Finally, as commodity hardware



(a) Naïve pointer-based tree representation



(b) Compressed tree encoding

Figure 5.1: **Two representations of the same tree: Naıve pointer-based and optimized**. The optimized version employs packing, breadth-first layout, and pointer compression via relative indexing.

typically supports MIMD evaluation across cores and SIMD within them, our techniques are complementary.

5.1 MIMD: Semi-static Work Stealing

We optimize the tree data representation and runtime schedule for MIMD evaluation. We did not see significant parallel speedups when either one was left out. Through a non-trivial amount of experimentation, we found an almost satisfactory combination of existing techniques. It includes popular ideas such as work stealing (Blumofe et al., 1995) for load-balanced runtime scheduling and tiling (Irigoin and Triolet, 1988) for data locality, and we report on how to combine them. However, we did not see more than 2X speedups until we added a novel technique to optimize for low run-time scheduling overheads and temporal data locality: semi-static work stealing. The remainder of this section explores our basic data representation and runtime scheduling techniques for MIMD parallelism.

Data Representation: Tuned and Compressed Tiles

Our data representation optimizes for spatial and temporal locality and, as will be used by the scheduler, low overheads for operating over multiple nodes. Many researchers have proposed individual techniques for similar needs, and it is unclear which to use for what hardware. For example, mobile devices typically have smaller caches than laptops, they

should exchange time for space. Our solution was to implement many techniques and build an autotuner that automatically choose an effective combination.

Our autotuner runs sample data on multiple configurations for a particular platform to decide which configuration to use. The most prominent options are:

- C++ collections or contiguous arrays
- tiling (Irigoin and Triolet, 1988) of subtrees
- depth-first or breadth-first ordering of nodes in a tile, with matching traversal order (Chilimbi et al., 1999)
- aligned data, or unaligned but more packed data
- pointer compression

Several of the techniques are parameterized, so our tuner performs a brute force search for parameter values such as the maximum size of a subtree tile. To make the search tractable, we prune by manually providing heuristics, such as for parameter ranges.

The individual optimizations target several objectives:

- **Compression** Compressing the tree better utilizes memory bandwidth and decreases the working set size. We use two basic techniques: structure packing and pointer compression. Packing combines several fields in the same word of memory, such as storing 32 boolean attributes in one 32 bit integer field. Similar to Lattner and Adve (2005), compression encodes node references as relative offsets (16–20 bits) rather than 32 bit of 64 bit pointers. Likewise, as there are typically few siblings, instead of a counter of number of children (or siblings), we use an isLastSibling bit. Figure 5.1 depicts a tree using pointers and one of our representations: in the example, the compressed form uses 96% fewer bits on a 64-bit architecture.
- **Temporal and Spatial Locality** The above compression optimizations improve locality by decreasing the distance between data. To further improve locality, we support rearranging the data in several ways .

  Tiling (Irigoin and Triolet, 1988) partitions the tree into subtrees and collocates nodes of the same subtree in memory. It improves spatial locality because a node only reads and writes to its neighbors. Likewise, we support breadth-first and depth-first node orderings within a subtree (and across subtrees). Such a representation matches the tree traversal order (Chilimbi et al., 1999) and therefore improves temporal locality.
- **Prefetching** We supports several options for prefetching to avoid waiting on data reads. First, the data access patterns with the data layout, so hardware prefetchers might automatically predict and prefetch data. Second, our compiler can automatically insert explicit prefetch instructions as part of the traversal. Finally, runahead

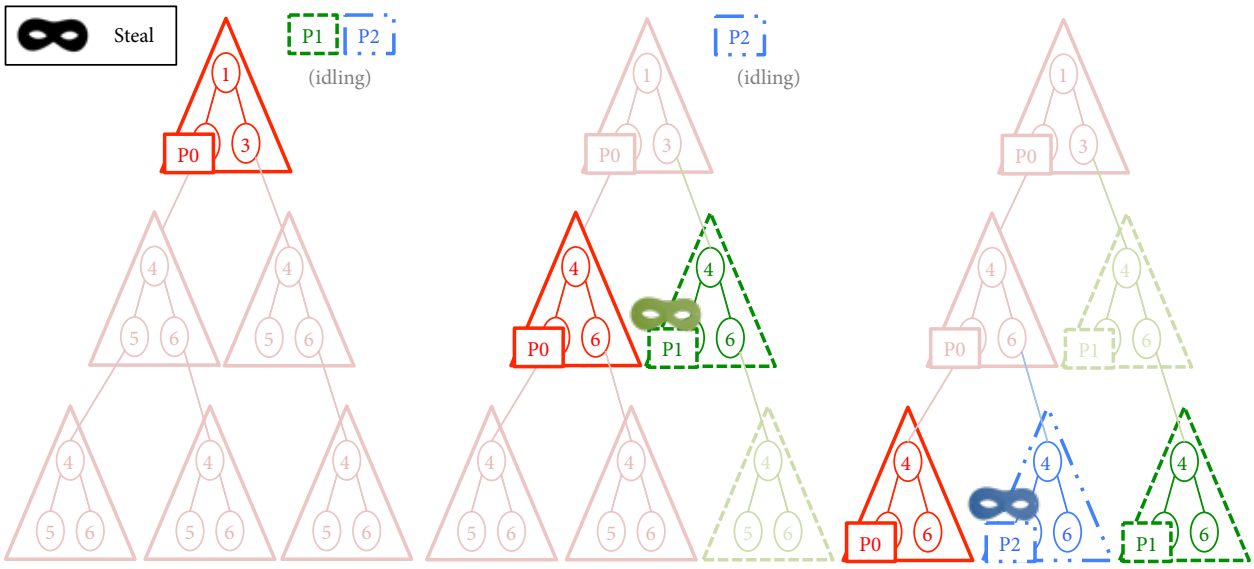


Figure 5.2: **Simulation of work stealing.** Top-down simulated tree traversal of a tiled tree by three processors in three steps.

processing pre-executes data access instructions. A helper thread traverses a subtree ahead of a corresponding evaluator thread, requesting node data while the evaluator is still computing an earlier thread. We only saw benefits of the first in practice, but leave the others as tunable.

- **Parallel scheduling.** Reasoning about individual nodes at runtime, such as for load balancing and synchronization, leads to high overheads. By scheduling tiles rather than nodes, we cut overheads. Because nodes correspond to tasks in our system, our approach is a form of *coarsening*. Furthermore, different synchronization strategies are possible for tiles, such as whether to use spin locks, so we autotune over the implementation options.

  We also support several scheduling options. First, we support third-party task schedulers, including Intel TBB (Reinders, 2007), Cilk (Blumofe et al., 1995), and those of Tesselation OS (Colmenares et al., 2013). Second, we built our own scheduler that uses a variant of work-stealing threads pinned to processors. It includes options such as whether to use hyper threads or not, and as we saw low speedups when using multiple sockets, options for how many threads to use. Our autotuner picks between scheduler implementations.

Figure 5.1 depicts several of the data representation optimizations: packing, pointer compression, and a breadth-first layout.

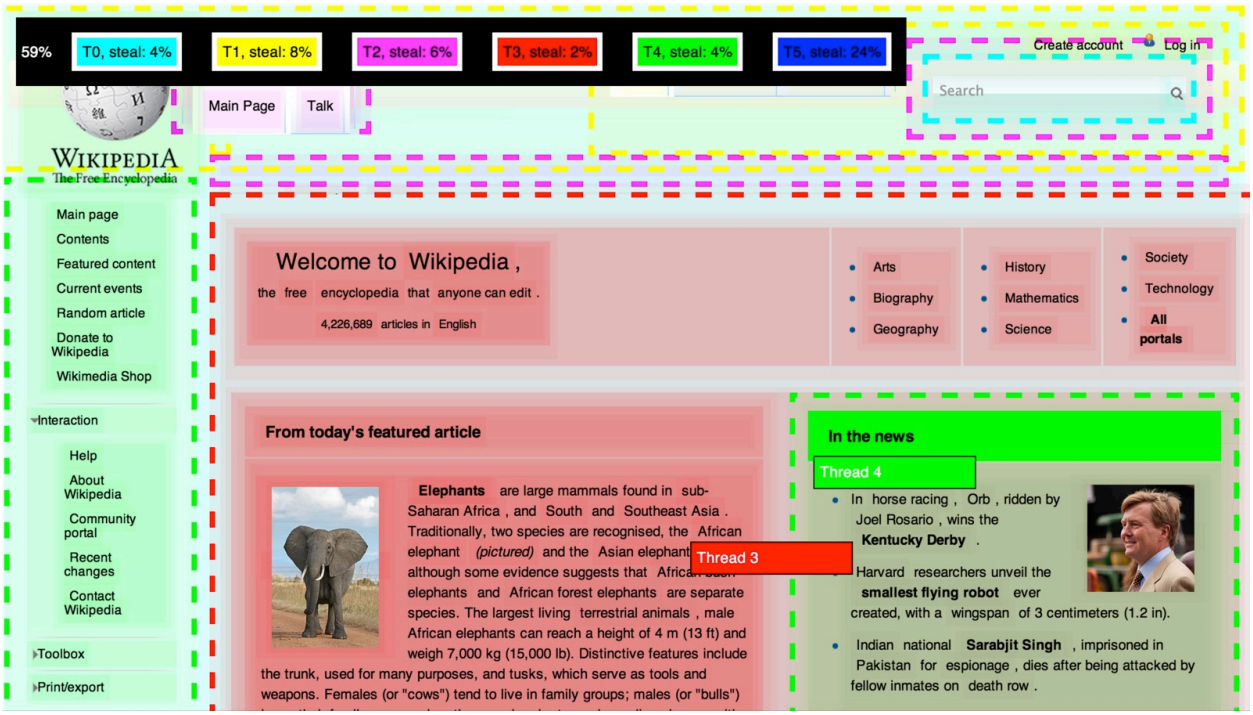


Figure 5.3: **Simulation of work stealing on Wikipedia.** Colors depict claiming processor and dotted boundaries indict subtree steals. Top-left boxes measure the percentage of steps an individual processor spent stealing rather than computing.

Scheduling: Semi-Static Work Stealing

We optimize our tree traversal task scheduler for low overheads, high temporal and spatial data locality, and load balancing. Webpages are relatively small and use many traversals, so we found that aggressively optimizing individual traversals to be an important implementation concern. Our approach is to combine static scheduling with dynamic work stealing. We did not see significant speedups with the base approaches on their own, but our combination led to 7X parallel speedups.

Work stealing

Work stealing was introduced as a dynamic scheduling algorithm that provides load balancing and spatial locality (Blumofe et al., 1995). Figure 5.2 depicts a trace of three processors performing work stealing. Each processor operates on an internal task queue, and whenever a processor exhausts its internal queue, it will *steal* from another processor's queue. In the case of a top-down tree traversal, acting upon an internal queue corresponds to a depth-first traversal of a subtree, and stealing corresponds to transferring ownership of an untraversed subtree. Figure 5.2 visualizes the spatial locality by using color to show that the same

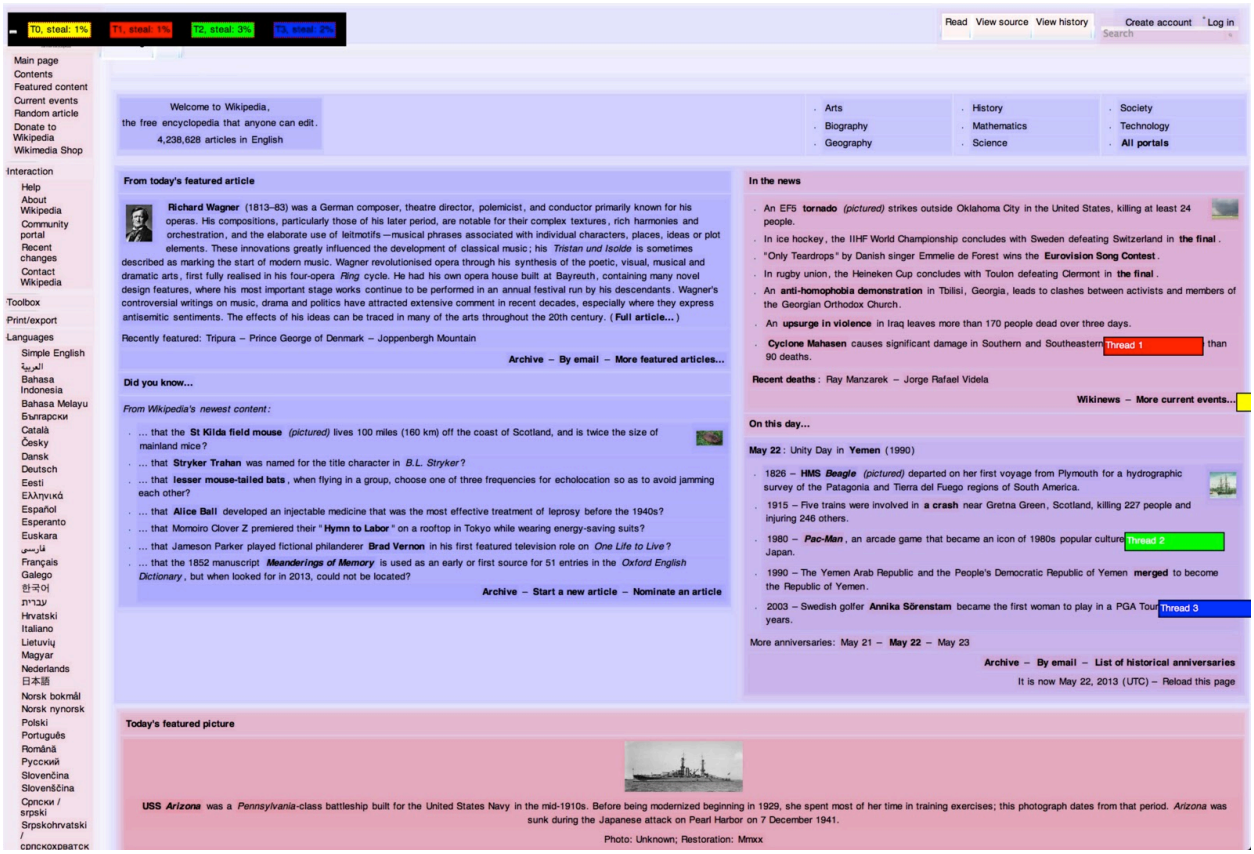


Figure 5.4: **Temporal cache misses for simulated work stealing over multiple traversals.** Simulation of four threads on Wikipedia. Blue shade represents a hit and red a miss. 67% of the nodes were misses. Top-left boxes the percent of steps different processors spent stealing.

processor generally claims adjacent nodes. Likewise, the figure demonstrates that there are relatively few scheduling overheads (steals are indicated by dotted borders).

Our work must address the problem that work stealing suffers from runtime overheads and poor temporal locality. To estimate the runtime overhead, we simulated work stealing for six processors on Wikipedia. Assuming uniform compute time per node, 5% of the nodes would trigger stealing. This cost is in addition to constant overhead to processing the internal per-processor task queues.

The problem with temporal locality is that a node will be assigned to different processors across multiple traversals. Figure 5.4 shows which nodes move across processors in a simulation of four processors performing a sequence of two traversals. Two thirds of the nodes are red, indicating substantial movement. Both the steal rate and temporal miss rate worsen as the number of processors increase.

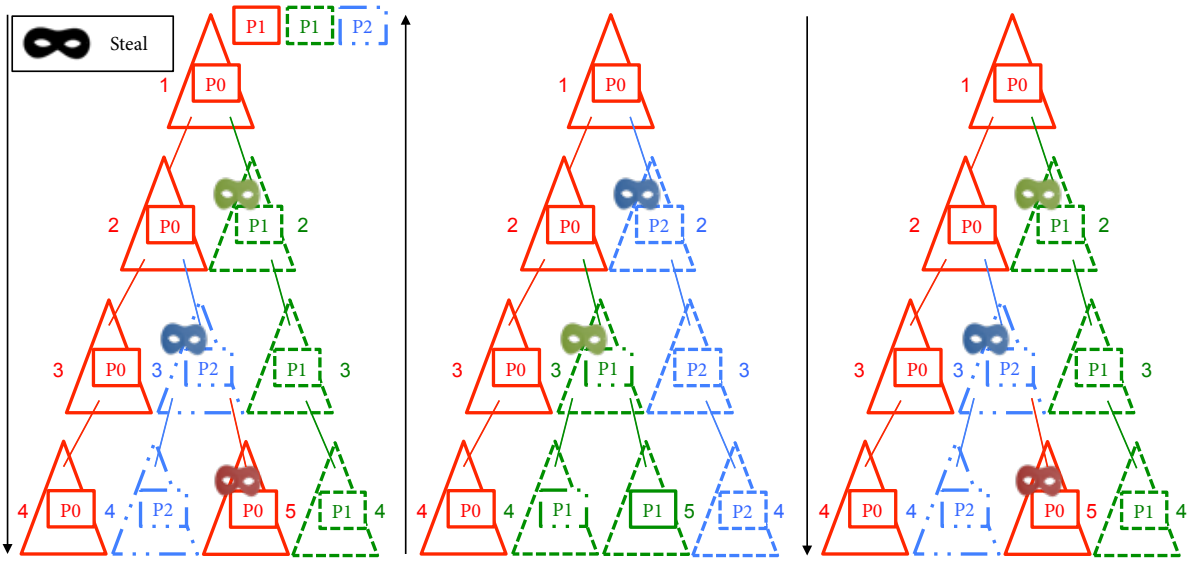


Figure 5.5: **Dynamic work stealing for three traversals.** Tiles are claimed by different processors in different traversals.

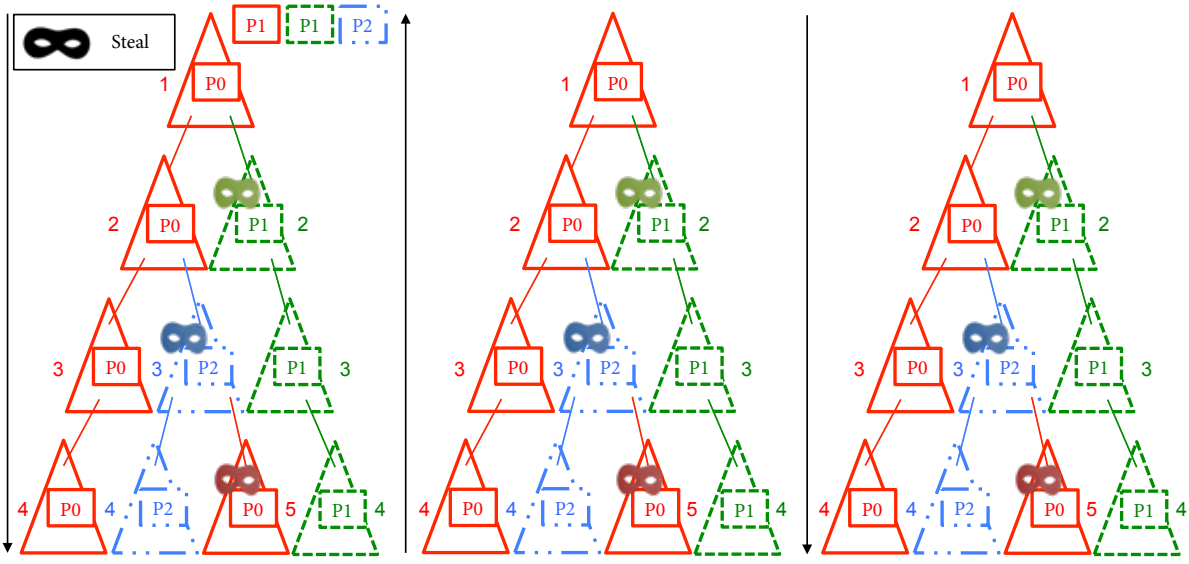


Figure 5.6: **Semi-static work stealing.** Dynamic schedule for first traversal is reused for subsequent ones.

Semi-Static Scheduling

Our scheduler optimizes the order in which every traversal visits nodes once the tree is loaded. The idea is to optimize the schedule for the first traversal and then reuse that schedule across the subsequent ones.

Simulation. For the first traversal, we use an approximation of work stealing to gain its load balancing and spatial locality benefits . Our variant lowers overheads in two ways. First, we coarsen the task size by scheduling tiles rather than individual nodes. Scheduling tiles lowers the number of runtime tasks that must be managed. Second, we use one thread to simulate the work stealing to determine the schedule rather than doing it dynamically. The simulation approximate the cost of a tile as the number of nodes within it and likewise penalizes steals. The simulation performs a linear walk through the metadata of the tiles without touching any actual nodes. We thus replace the overheads of managing concurrent task queues over multiple traversals with the overheads of an initial simulation over the metadata and, when following the schedule, tile locks.

Schedule Reuse. For subsequent traversals, we reuse the simulated schedule in order to also optimize for temporal locality. For example, two successive preorder traversals would use the same schedule because they share the same static dependencies across nodes. A follow-on postorder traversal would follow the schedule in reverse. A schedule localizes nodes to processors, so reusing one across traversals means a node will be accessed by the same processor. The spatial locality and load balancing benefits of work stealing apply to each traversal because, as few instructions execute per-node, the workloads are similar.

Our approach achieves low overheads, high temporal and spatial locality, and load-balanced evaluation. Temporal locality is enforced by reusing the same schedule across the traversals, and semi-static scheduling with a fast heuristic provides low overheads. Our work stealing heuristic provides spatial locality and an approximate form of load balancing.

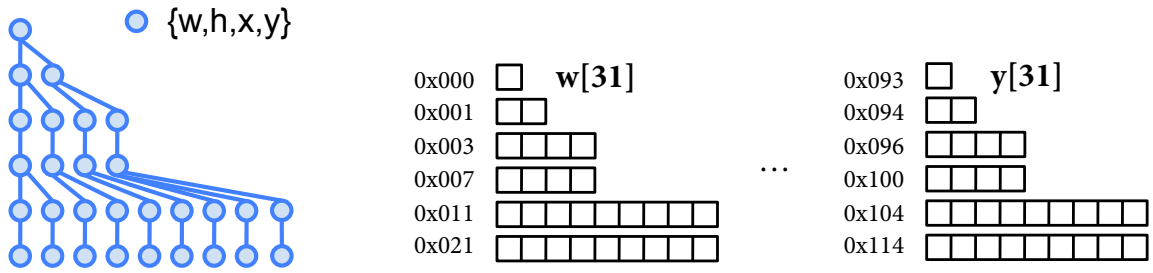
5.2 SIMD Background: Level-Synchronous Breadth-First Tree Traversal

We built our new SIMD optimizations upon the idea of implementing preorder and postorder tree traversals as level-synchronous breadth-first tree traversals. Reps first suggested the related concept of scan grammars (Reps, 1993), but did not implement it. We tried implementing them using the schedules and data representation of more recent data parallel languages such as NESL (Blelloch et al., 1994) and Data Parallel Haskell (Chakravarty et al., 2007), but we saw no speedups for document layout. Our new approach is discussed in the next section, but first, we present an overview of the level-synchronous breadth-first tree traversals in all these techniques.

The naıve tree traversal schedule is to sequentially iterate over one level of the tree at a time and traverse the nodes of a level in parallel. A parallel preorder traversal starts on

```
1  void parPre(void (*visit)(Prod &), List<List<Prod>> &levels) {
2    for (List<Prod> level in levels)
3          parallel_for (Prod p in level)
4                visit(p)
5  }
6  void parPost(void (*visit)(Prod &), List<List<Prod>> &levels) {
7    for (Array<Prod> level in levels.reverse())
8          parallel_for (Prod p in level)
9                visit(p)
10 }
```

(a) **Level-synchronous Breadth-First Traversal**



(b) **Logical Tree** (c) **Tree Representation**

Figure 5.7: **SIMD tree traversal as level-synchronous breadth-first iteration with corresponding structure-split data representation.**

the root node's level and then proceeds downwards. Likewise, a postorder traversal starts on the tree fringe and moves upwards (Figure 5.7).

The level synchronous approach features several benefits for SIMD evaluation. Unlike the MIMD depth-first traversals where different processors may be accessing different levels of the tree, breadth-first traversals guarantee level-by-level access. SIMD hardware can exploit the breadth-first order to coalesce memory accesses to adjacent nodes. Furthermore, in data visualizations, we empirically observed that most of the nodes on the same level will dispatch to the same layout instructions. Computing on one level at a time helps SIMD evaluation avoid a source of instruction divergence.

To eliminate a key source of divergent memory accesses, the level-synchronous traversal uses a corresponding data representation. Instead of storing adjacent nodes side-by-side, the layout stores node attributes in *column* order by converting the array-of-structures to a structure-of-arrays. The conversion collocates individual attributes, such as the width attribute of one node being stored next to the width attribute of the node's sibling (Figure 5.7c). The index of a node in a breadth-first traversal of the tree is used to perform a lookup in any of the attribute arrays. The benefit of this encoding is that during SIMD layout of several adjacent nodes, reads and writes are coalesced into bulk reads and writes. For example, if a layout pass adds a node's padding to its width, several contiguous paddings and several contiguous widths will be read, and the sum will be stored with a contiguous write. These optimizations are crucial because the penalty of non-coalesced access is high

and, for layout, relatively few computations occur for each read and write.

The implementation of the data representation addresses additional subtleties:

- **Level representation.** To eliminate traversal overhead, a summary provides the index of the first and last node on each level of a tree. Such a summary provides data range information for launching the parallel kernels that evaluate the nodes of a level as well as the information for how to proceed to the next level.

- **Edge representation.** A node may need multiple named lists of children, such as an HTML table with a header, footer, and an arbitrary number of rows. We encode the table's edges as three global arrays of offsets: header, footer, and first-row. To support iterating across rows, we also introduce a fourth array to encode whether a node is the last sibling. Thus, any named edge introduces a global array for the offset of the pointed-to node, and for iteration, a shared global array reporting whether a node at a particular index is the end of a list.

- **Memory compression.** Allocating an array the size of the tree for every type of node attribute wastes memory. We instead statically compute the maximum number of attributes required for any type of node, allocate an array for each one, and map the attributes of different types of nodes into different arrays. For example, if we extend `H-AG` with circle nodes that have attributes "r" and "angle", four arrays will be allocated. The HBox nodes require an array for each of the attributes "w", "h", "x", and "y" while the circle nodes only require two arrays. If a node's type is HBox, its entry in the first array will contain the 'w' attribute. If the node has type Circle, the node's entry in the first entry will contain the "r" attribute.

- **Tiling.** Local structural mutations to a tree such as adding or removing nodes should not force global modifications, such as moving all subsequent nodes in memory. As most SIMD hardware has limited vector lengths (e.g., 32 elements wide), we split our representation into blocks and limit modifications to them. For example, adding nodes may require allocation of a new block and reorganization of the old and new block. Likewise, after successive additions or deletions, the overall structure may need to be compacted. Such techniques are standard for file systems, garbage collectors, and databases.

In summary, our basic SIMD tree traversal schedule and data representation descend from the approach of NESL (Blelloch et al., 1994) and Data Parallel Haskell (Chakravarty et al., 2007). Previous work shows how to generically convert a tree of structures into a structure of arrays. Those approaches do not support statically unbounded nesting depth (i.e., tree depth), but we support arbitrary tree depths because our transformation is not as generic.

A key property of all of our systems, however, is that the structure of the tree is fixed prior to the traversals. In contrast, for example, parallel breadth-first traversals of graphs

will dynamically find the edges of a minimum spanning tree (Merrill et al., 2012). Such dynamic alternatives incur unnecessary overheads when performing a sequence of traversals and sacrifice memory coalescing opportunities. Layout is often a repetitive process, whether due to multiple tree traversals for one invocation or an animation incurring multiple invocations, so costs in creating an optimized data representation and schedule are worth paying.

5.3 Input-dependent Clustering for SIMD Evaluation

Once the tree is available, we automatically optimize the schedule for traversing a tree level in a way that avoids instruction divergence. Our insight is that we can cluster tasks (nodes) based on node attributes that influence control flow. Our runtime then matches the data layout to the new schedule and optimizes the clustering process to prevent the planning overhead to outweigh its benefit. Once nodes are in clusters, a traversal proceeds cluster-by-cluster and can use SIMD instructions within each one. The overall optimization can be thought of an extension to loop unswitching where the predicate is input-dependent; we add a prepass to achieve the necessary sorting invariant.

The Problem

The problem we address stems from layout being a computation where the instructions for each node are heavily input dependent. The intuition can be seen in comparing the visual regularity of a webpage against that of a data visualization. Different parts of a webpage look quite different from one another, which suggests sensitivity to values in the input tree. In contrast, the points of a chart generally look quite similar and thus does not use widely different instructions for different nodes. For `H-AG`, an HBox's width is the sum of its children widths, but a VBox's width is their maximum. For a random interleaving of HBox and VBox nodes, parallel visits diverge in instruction selection based on the node type.

We ran a simulation to demonstrate the performance cost of the divergence. If different types of nodes are uniformly distributed in random across a level, as the number of types of nodes go up, the probability that all of the nodes in a group share the same instructions drops exponentially. Figure 5.8 shows the simulated speedup for SIMD evaluation over a tree level of 1024 nodes on computer architectures with varying SIMD lengths. The x-axis of each chart represents the number of types and the y-axis is the speedup. As the number of choices increase, the benefit of the naıve breadth-first schedule (red line) decreases. The instruction divergence causes the speedup to be far from the ideal, which we estimated as a function of the SIMD length of the architecture (maximal parallel speedup, contributing the horizontal portion of the green lines) and the expected number of different types (mandatory divergences, contributing the diagonal portion).

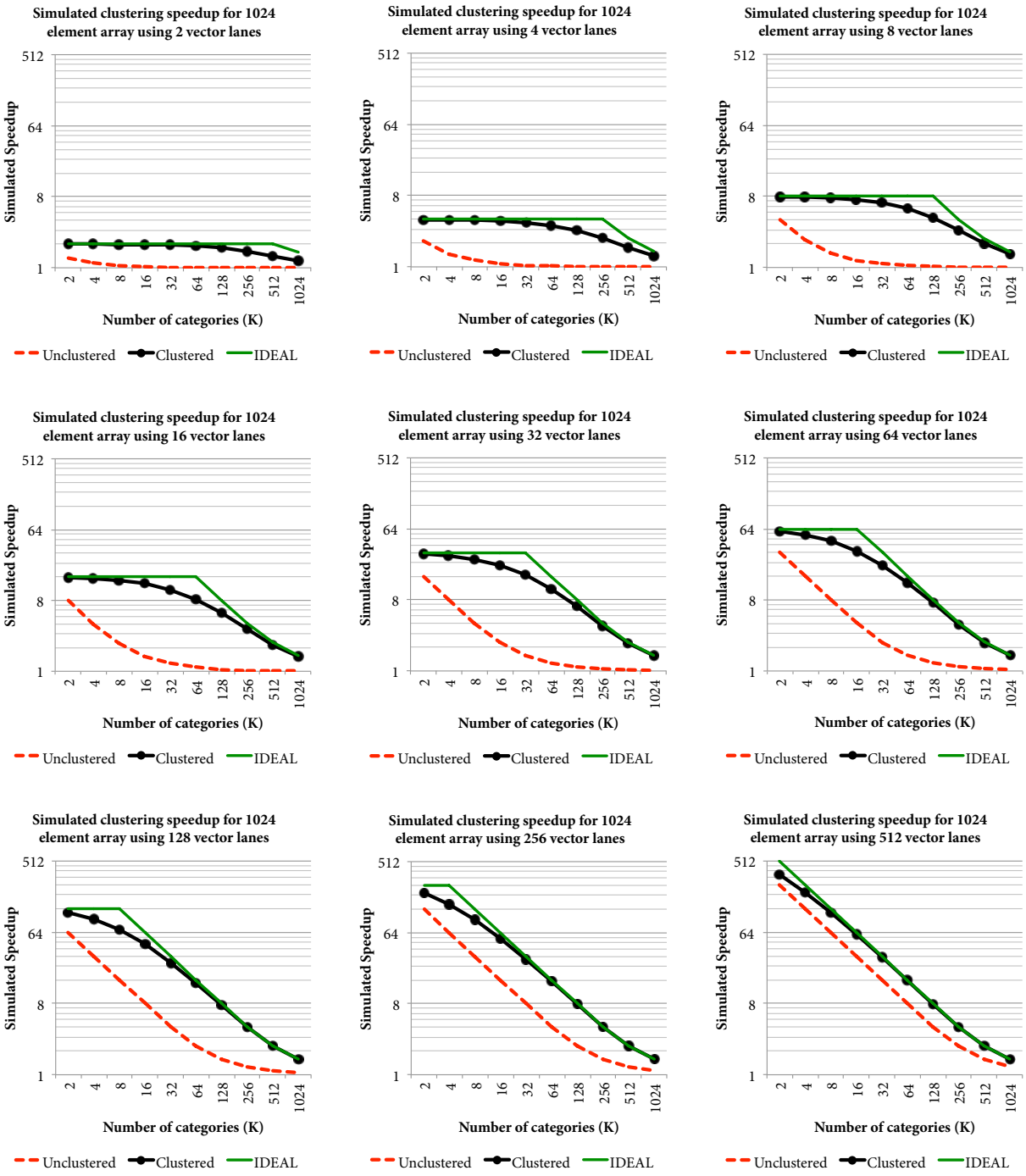


Figure 5.8: **Simulated vectorization speedup for different schedules.** Successive diagrams increase the number of vector lanes by a power of two.

```
1  void parPreClustered(void (*visit)(Prod &), List<List<Array<Prod>>> &levels) {
2    for (List<Prod> level in levels)
3          for (Array<Prod> cluster in level)
4                parallel_for (Prod p in cluster)
5                      visit(p)
6  }
```

Figure 5.9: **Clustered parallel preorder traversal.**

```
1  Prod firstProd = cluster[0]
2  parallel_for (prod in Cluster) {
3    switch (firstProd.type) {
4      case S → HBOX: break;
5      case HBOX → ϵ:
6        HBOX.w = input();
7        HBOX.h = input();
8        break;
9      case HBOX → HBOX1 HBOX2:
10       HBOX0.w = HBOX1.w + HBOX2.w;
11       HBOX0.h = MAX(HBOX1.h, HBOX2.h);
12       break;
13   }
14 }
```

(a) **Clustered dispatch.**

```
1  Prod firstProd = cluster[0]
2    switch (firstProd.type) {
3      case S → HBOX: break;
4      case HBOX → ϵ:
5        parallel_for (prod in Cluster) {
6            HBOX.w = input();
7            HBOX.h = input();
8        }
9        break;
10     case HBOX → HBOX1 HBOX2:
11       parallel_for (prod in Cluster) {
12         HBOX0.w = HBOX1.w + HBOX2.w;
13         HBOX0.h = MAX(HBOX1.h, HBOX2.h);
14       }
15       break;
16   }
17 }
```

(b) **Unswitched dispatch**.

Figure 5.10: **Loop transformations to exploit clustering for vectorization.**

Code Clustering

Our solution is to cluster nodes of a level based on the values of attributes that influence the flow of control. SIMD evaluation of the nodes in a cluster will be free of instruction divergence. Furthermore, by changing the data representation to match the clustered schedule, memory accesses will also be coalesced. We first focus on applying the clustering transformation to the code.

Figure 5.9 shows the clustered evaluation variant of the MIMD *parPre* traversal for `H-AG`. The traversal schedule is different because the order is based on the clustering rather than breadth-first index. Changing the order is safe because the original loop was parallel with no dependencies between elements. Computing over clusters guarantees that all calls to a visit dispatch function in the parallel inner loop (e.g., of *visit*1) will branch to the same case of the switch statement. GPU ("SIMT") architectures will automatically exploit this phenomena.

Subword-SIMD architectures need an additional code transformation. We performed a loop transformation that can be understood as a form of loop unswitching. Loop unswitching lifts a conditional out of a loop by duplicating the loop inside of both cases of the conditional. Clustering establishes the invariant that the dispatch used for the first element of a cluster

is the same for the rest, so the hoisted conditional need only check one item. Figure 5.10 demonstrates how to use the same exemplar for the dispatch.

Clustering is with respect to input attributes that influence control flow, which may include more than just the node type. For example, in our vectorization of the C3 layout engine (Burg and Schulte, 2011), we found that the engine author combined the logic of multiple box types into one visit function because the variants shared a lot of code. He recorded which code to use with multiple node flags and invoked them through if-then statements. Both the node type and various other node attributes influenced control flow, and therefore our clustering condition dependend on whether they were all equal. Using all of the attributes led to too fine of a clustering condition, so we manually tuned the choice of attributes.

Data Clustering

Because the data representation should be modified to match the clustering order, we colocate nodes of a cluster. The benefit is coalesced memory accesses, but overhead costs in performing the clustering should be considered.

Reordering data is expensive as all of the data is moved. In the case of our data visualization system, we avoided such costs because the data is preprocessed on our server. For webpage layout, the client performs clustering at runtime. We optimized the clustering enough such that the cost is outweighed by the subsequent performance improvements.

We optimized reordering with a simple parallel two-pass technique. The first pass traverses each level in parallel to compute the cluster for each node and tabulates the cluster sizes for each tree level. The second pass again traverses each level in parallel, and as each node is traversed, copies it into the next free slot of the appropriate cluster. Even finer-grained parallelization is possible, but this algorithm was sufficient for lowering reordering costs enough to be amortized.

Nested Clustering

We experimented with applying clustering to address different types of divergences encountered when computing over trees:

- **Branches.** For some cases of webpage layout, attributes of the parent node or children influence instruction selection, such as whether to include a child node in a width computation. We included these properties in the clustering condition in order to eliminate the corresponding instruction divergences.

- **Load imbalance in loops.** One node may have no children while another may have many. If the layout computation involves a loop, SIMD evaluation will perform the two loops in lock-step. Thus, as the nodes have different amounts of children, the SIMD lanes devoted to the smaller-length node will not be utilized: this is a load

balancing problem. The number of children can be included in the clustering condition to eliminate load imbalance.

- **Random memory access in loops.** A further issue with lock-step loops over child nodes is memory divergence. An unclustered breadth-first layout would provide strided memory access, such as for the first child of each node. However, if each level is clustered, the locations of a node's children may be random without further aid. We found a *nested* solution where *subtrees* are assigned to clusters. Beyond associating nodes of a level with a cluster, our algorithm also treats the nodes of a cluster as roots. It recursively expands subtrees in tandem to determine clusters that span multiple levels. The data layout follows the nested clustering, so parallel memory accesses to the children of nodes will be coalesced. Likewise, traversals of the next level of the tree will also achieve coalesced accesses.

Each of these clusterings introduce an intra-cluster invariant that we exploited for optimizing performance within that cluster. However, the clustering condition is more discriminating. Cluster sizes may decrease, which would decrease performance if cluster size shrinks below vector length size. Section 5.4 explores these options in practice.

5.4 Evaluation

We evaluate speedups from each of our techniques. For MIMD architectures, we demonstrated that semi-static work stealing achieves better load balancing and suffers from lower overheads than techniques such as dynamic work stealing. For SIMD architectures, we verified the benefit of using level-synchronous breadth-first traversals (Blelloch et al., 1994), and for the case of webpage layout, show that clustering nodes improves scalability. We evaluated both the sequential and parallel speedups due to our technique, and in the case of SIMD evaluation, measured the power efficiency.

MIMD Data Representation and Scheduling Optimizations

By statically exposing traversal structure (e.g., **parPre**) to our code generators, we observe sequential and parallel speedups. We separately evaluate the importance of the data representation optimizations from the scheduling ones on randomly generated 500-1000 node documents. The implementation under test is `hbox++` , which extends `H-AG` with additional node types such as vertical boxes and extra attributes such as padding. Finally, we examine the parallel benefit on webpages for our implementation of CSS.

We first evaluate the perform of our task scheduler (FTL in Table 5.1). Our comparison point is Intel's TBB (Reinders, 2007) dynamic task scheduler that performs work stealing (Blumofe et al., 1995), which was the most efficient third-party work stealing library that we tried. We included our data layout optimizations in all calculations because, without them, we saw no speedup. TBB causes slowdowns until achieving no cost (nor benefit)

	Total speedup Cores				**Parallel speedup** Cores		
Configuration	1	2	4	8	2	4	8
TBB, server	1.2x	0.6x	0.6x	1.2x	0.5x	0.5x	1.0x
FTL, server	1.4x	2.4x	5.2x	9.3x	1.8x	3.8x	6.9x
FTL, laptop	1.4x	2.1x			1.6x		
FTL, mobile	1.3x	2.2x			1.7x		

Table 5.1: **Speedups and strong scaling across different schedulers and hardware**. Baseline is a sequential traversal with no data layout optimizations. FTL is our multicore tree traversal library. Left columns show total speedup (including data layout optimizations by our code generator) and right columns show just parallel speedup. Server = Opteron 2356, laptop = Intel Core i7, mobile = Atom 330.

		Parallel speedup Cores		
Backend	Input	2	4	8
TBB	Wikipedia	1.5x	1.6x	1.2x
TBB	xkcd Blog	1.5x	1.8x	1.2x
FTL	Wikipedia	1.6x	2.8x	3.2x
FTL	xkcd Blog	1.5x	2.3x	3.1x

Table 5.2: **Parallel CSS layout engine**. Run on a 2356 Opteron.

at 8 cores. We hypothesize that it suffered from high overheads: switching to scheduling tiles by using our optimized data representation improved performance. Our semi-static working stealing scheduler, however, achieved a 6.9X speedup on 8 cores. We did not see significant further speedups for higher core counts, and hypothesize that it is due to the socket jump. We experimented with other schedulers, such as a parallel for-loop over tiles near the fringe of the tree, but the achieved 2X speedup is much lower than the 6.9X of our semi-static work stealer.

Optimizing the data representation was key to achieving parallel speedups. Doing so achieved 1.2X-1.4X speedups for sequential processing (Table 5.1). However, on 4 cores, it improved the cumulative speedup from 2.8X without data representation optimizations to 5.2X when using them. The difference is 1.9X: our data representation optimizations both complement and improve scheduling optimizations. Without them, parallel performance was poor.

Table 5.2 shows the parallel speedup on running our 9 pass layout engine for two popular web pages that render faithfully with it: Wikipedia and the XKCD blog. Note that the benchmarks do *not* include sequential speedups. The best performance of TBB was a 1.8X

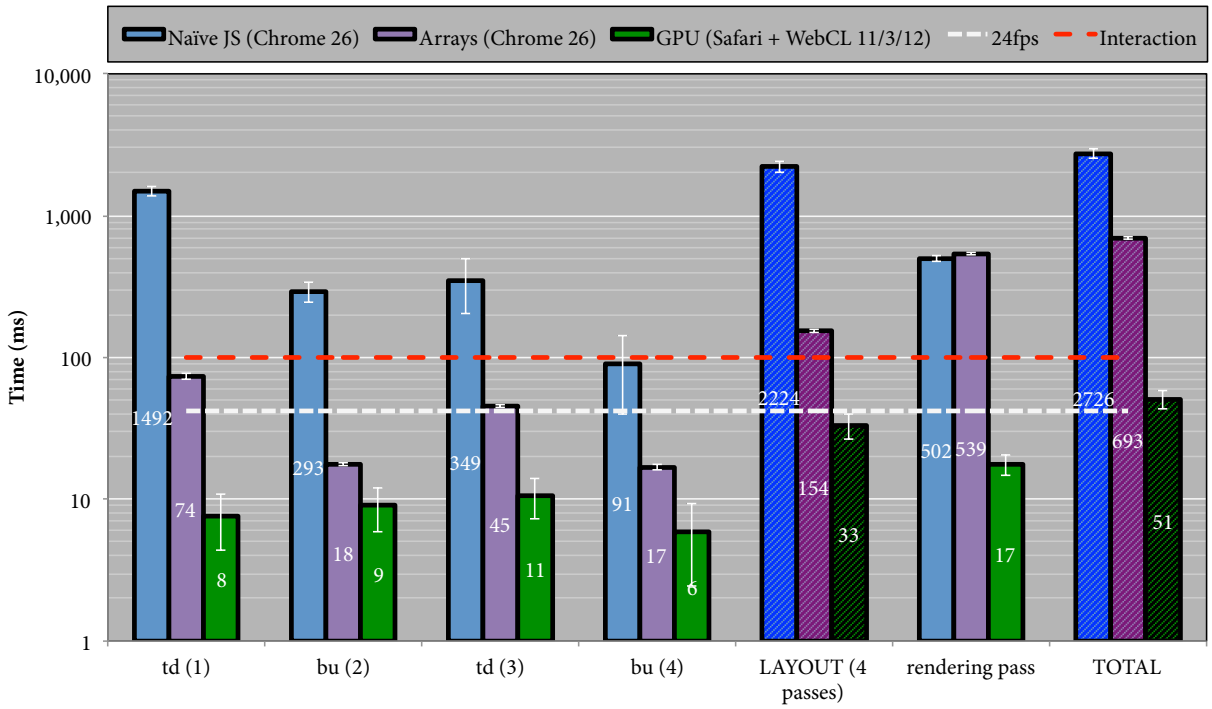


Figure 5.11: **Sequential and parallel benefits of breadth-first layout and staged allocation.** Allocation is merged into the 4th stage and buffer indexing and tessellation form the rendering pass. JavaScript variants use HTML5 canvas drawing primitives while WebCL does not include WebGL painting time (< 5ms). Thin vertical bars indicate standard deviation and horizontal bars show deadlines for animation and hand-eye interaction.

speedup on 4 cores, and TBB's speedup on 8 cores was 1.2X. In comparison, our scheduler achieved 2.8X on 2 cores and 3.2X on 8. We blame the lack of further benefits on overheads. Across our benchmarks, we generally saw speedups when sequential traversals took longer than a certain amount, but because so many traversals are used for CSS, enough of them are small enough that we do not expect strong scaling. Our intuition is that either a full layout engine is complicated enough that the sequential cost of each traversal will be higher than in our prototype, or even more aggressive data representation optimizations should be performed. As is, we have demonstrated significant 3X+ speedups on real workloads from just the parallelization.

Baseline SIMD Speedups (GPU)

We evaluate the sequential and parallel performance benefits of our baseline breadth-first layout. For an animation to achieve 24fps, the time spent to process a frame should not exceed 42ms, and for hand-eye interactions, 100ms (10fps). We examine the case of a 5 pass treemap that supports live filtering over 100,000 data points. The first 3 passes are purely

devoted to layout, the 4th pass includes layout computations and allocation requests, and the 5th pass propagates buffer indices and performs tessellation.

We compare 3 backends for our compiler: canonical JavaScript (a tree of nodes), JavaScript over our structure-split breadth-first tree layout (and with typed arrays), and WebCL for the GPU. The first two variants invoke HTML5 canvas drawing primitives, while the last invokes WebGL (GPU) painting primitives over vertex buffers computed in the rendering pass. The time for WebGL painting calls are not shown, but they take less than 5ms. Each variant is repeated 15 times on a 4 core 2012 2.66GHz Intel Core i7 with 8 GB memory and a 1024 MB NVIDIA GeForce GT 650M graphics card.

We first examine the significant sequential benefits. The first 4 groups of columns in Figure 5.11 shows the average time spent on different layout passes and the 6th on the pass for buffer index computation and tessellation. Changing the data representation and schedule enables a 14X sequential speedup on layout in the Chrome web browser. No speedup is observed in the rendering pass because the time is dominated by HTML5 canvas calls. We hypothesize part of the sequential benefit is related to our clustering optimization: all of the nodes in a level have the same type, so implicit optimizations such as branch prediction should perform better when moving away from the depth-first traversal. Finally, we note that while sequential layout time is a magnitude too slow for real-time animation, our parallel prototype is within 54ms for real-time interaction (ignoring rendering).

Parallel speedups are also significant. WebCL (GPU) evaluation of layout is 5X faster than sequential. The impact of compiling JavaScript versus C (WebCL) on the benchmark is unclear: JavaScript is generally a magnitude slower than native code, except the runtime WebCL compiler is not running at high optimization levels. The benefits for parallel computation of the buffer indices and tessellation is much more clear: the speedup is 31X.

To better understand the benefit of parallelization, we compared running the layout traversals using multicore versus GPU acceleration (Figure 5.12) for an early prototype of the layout traversals. Both use breadth-first traversals compiled with OpenCL, except differ on the hardware target. We see that a server-grade multiprocessor (32-core AMD Opteron 2356) can outperform a laptop GPU, but the comparison is unfair in terms of form-factor and implications for power ratings.

Ultimately, when the sequential and parallel optimizations are combined, we see an end-to-end speedup of 54X. It is high enough such that it enables real-time animation for our dataset, not just real-time user interaction.

SIMD Clustering

We evaluated several aspects of our clustering approach. First, we examined applicability to various layouts. Second, we evaluated the speed and power benefit. Clustering provides invariants that benefit more than just vectorization, so we distinguish sequential versus parallel speedups. Finally, there are different options in what clusters to form, so we measured ideal speedups (compression ratios) and observed ones.

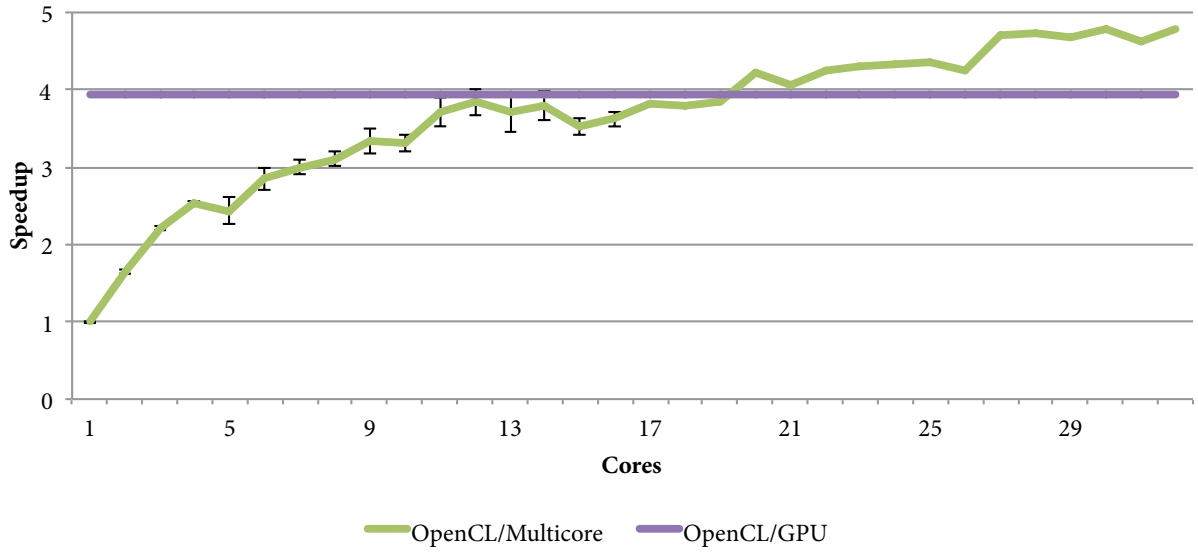


Figure 5.12: **Multicore versus GPU acceleration of layout.** Benchmark on an early version of the treemap visualization and does not include rendering pass.

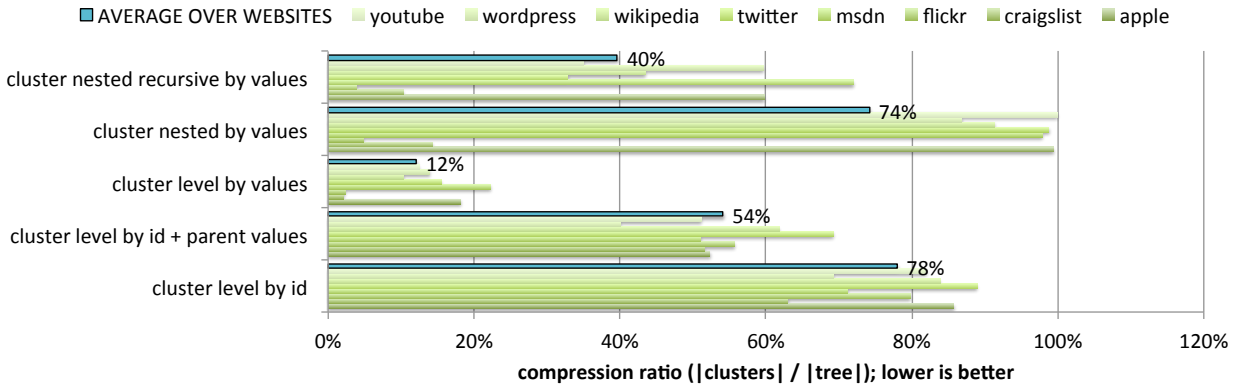


Figure 5.13: **Compression ratio for different CSS clusterings.** Bars depict compression ratio (number of clusters over number of nodes). Recursive clustering is for the reduce pattern, level-only for the map pattern. ID is an identifier set by the C3 browser for nodes sharing the same style parse information while value is by clustering on actual style field values.

Applicability

We examined idealized speedup for several workloads:

- **Synthetic.** For a controlled synthetic benchmark, we simulated the effect of increasing number of clusters on speedup for various SIMD architectures. Our simulation assumes perfect speedups for SIMD evaluation of nodes run together on a SIMD unit. The ideal speedup is a function of the minimum of the SIMD unit's length (for longer clusters, multiple SIMD invocations are mandatory) and the number of clusters (at least one SIMD step is necessary for each cluster). Figure 5.8 shows, for architectures of different vector length, that the simulated speedup from clustering (solid black line with circles) is close to the ideal speedup (solid green line).

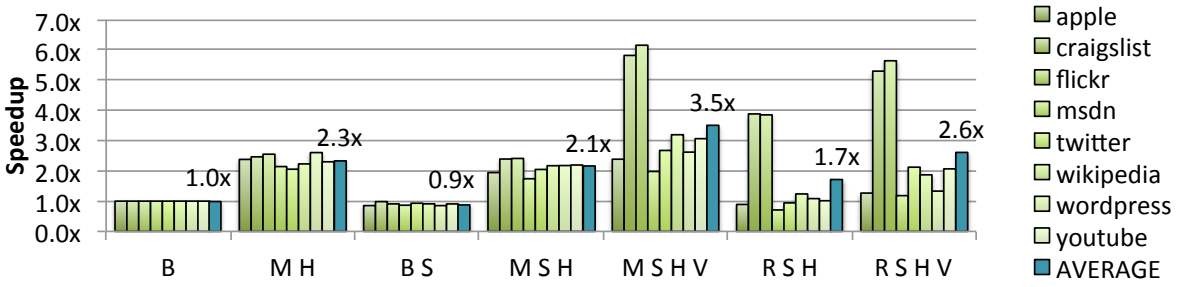
- **Data visualization.** For our data visualizations, we found that across the board all of the nodes of a level shared the same type. For example, our visualization for multiple line graphs puts the root node on the first level, the axis for each line graph on the second level, and all of the actual line segments on the third level.

- **CSS.** We analyzed potential speedup on webpages. Webpages are a challenging case because an individual webpage features high visual diversity, with popular sites using an average of 27KB of style data per page [1]. We picked 10 popular websites from the Alexa Top 100 US websites that rendered sufficiently correctly in the C3 web browser (Burg and Schulte, 2011).

  Figure 5.13 compares how well nodes of a webpage can be clustered. It reports the *compression ratio*, which divides the number of clusters by the number of nodes. Sequential execution would assign each node to its own cluster, so the ratio would be 1. In contrast, if the tree is actually a list of 100 elements, and the list can be split into 25 clusters, the ratio would be 25%. Assuming infinite-length vector processors and constant-time evaluation of a node, the compression ratio is the exact inverse of the speedup. A ratio of 1 leads to a 1X speedup, and a compression ratio of 25% leads to a 4X speedup.

  Clustering each level by attributes that influence control flow achieved a 12% compression ratio (Figure 5.13): an 8.3X idealized speedup. When we strengthened the clustering condition to enforce stronger invariants in the cluster, such as to consider properties of the parent node, the ratio quickly worsened. Thus, we see that our basic approach is promising for websites on modern subword-SIMD instruction sets, such as a 4-wide SSE (x86) and NEON (ARM), and the more recent 8-wide AVX (x86). Even longer vector lengths are still beneficial because some clusters were long. However, eliminating all divergences requires addressing control flows influenced by attributes of node neighbors, which leads to poor compression ratios. Thus, we emphasize that

[1] https://developers.google.com/speed/articles/web-metrics



B =breadth first, S = structure splitting, M = level clustering, R = nested clustering, H = hoisting, V = SSE 4.2

Figure 5.14: **Speedups from clustering on webpage layout.** Run on a 2.66GHz Intel Core i7 (GCC 4.5.3 with flags -O3 -combine -msse4.2) and does not preprocessing time.

8.3X is an upper bound on the idealized speedup: not all branches in a cluster are addressed.

Empirically, we see that clustering is applicable to CSS, and unnecessary in the case of our data visualizations.

Speedup

We evaluated the speedup benefits of clustering for webpage layout. We take care to distinguish sequential benefits from parallel benefits, and compare different clustering approaches. Our implementation was manual: we examined optimizing one pass of the C3 browser's CSS layout engine (Burg and Schulte, 2011) that is responsible for computing intrinsic dimensions. The C3 browser was written in C#, so we wrote our optimized traversal in C and pinned the memory for shared access between C# and C. We used a breadth-first tree representation and schedule for our baseline, but note that doing such a layout already provides a speedup over C3's unoptimized global layout.

For our experimental setup, we evaluated the same popular webpages above that rendered legibly with the experimental C3 browser. Benchmarks ran on a 2.66GHz Intel Core i7 (GCC 4.5.3 with flags -O3 -combine -msse4.2). We performed 1,000 trials, and to avoid warm data cache effects, iterated through different webpages.

We first examine sequential performance. Converting an array-of-structures to a structure-of-arrays causes a 10% slowdown (B S in Figure 5.14). However, clustering each level and hoisting computations shared throughout a cluster led to a 2.1X sequential benefit (M S H). Nested clustering provided more optimization opportunities, but the compression ratio worsened: it only achieved a 1.7X sequential speedup (R S H). Simple level clustering provides a significant sequential speedup.

Next, we examine the benefit of vectorization. SSE instructions provide 4-way SIMD parallelism. Vectorizing the nested clustering improves the speedup from 1.7X to 2.6X, and

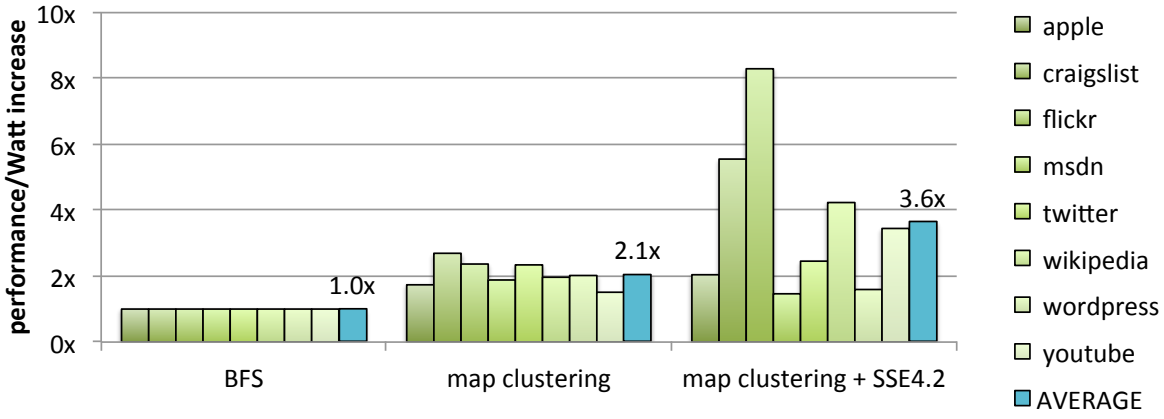


Figure 5.15: **Performance/Watt increase for clustered webpage layout.**

the level clustering from 2.1X to 3.5X. Thus, we see significant total speedups. The 1.7X relative speedup of vectorization, however, is still far from the 4X: level clustering suffers from randomly strided children, and the solution of nested clustering sacrifices the compression ratio.

Power

Much of our motivation for parallelization is better performance-per-Watt, so we evaluated power efficiency. To measure power, we sampled the power performance counters during layout. Each measurement looped over the same webpage over 1s due to the low resolution of the counter. Our setup introduces warm cache effects, but we argue it is still reasonable because a full layout engine would use multiple passes and therefore also have a warm cache across traversals.

In Figure 5.15, we show a 2.1X improvement in power efficiency for clustered sequential evaluation, which matches the 2.1X sequential speedup of Figure 5.14. Likewise, we report a 3.6X cumulative improvement in power efficiency when vectorization is included, which is close to the 3.5X speedup. Thus, both in sequential and parallel contexts, clustering improves performance per Watt. Furthermore, the similarity between speedup and power numbers supports the general reasoning in parallel computing of 'race-to-halt' as a strategy for improving power efficiency.

Overhead

Our final examination of clustering is of the overhead. Time spent clustering before layout must not outweigh the performance benefit; it is an instance of the planning problem in AI.

For the case of data visualization, we flatten the tree into arrays with a preprocessor on the server that runs off the critical path. Thus, our data visualizations experience no

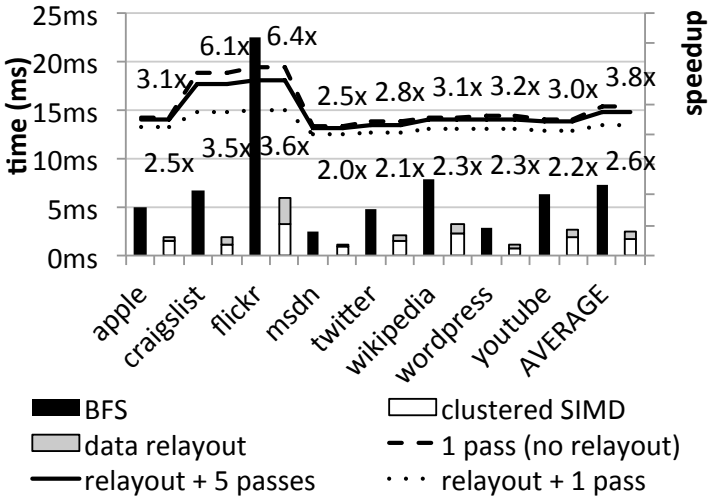


Figure 5.16: **Impact of data relayout time on total CSS speedup.** Bars depict layout pass times. Speedup lines show the impact of including clustering preprocessing time.

clustering cost.

For webpage layout, clustering is performed on the client when the webpage is received. We measured performing clustering with our two-pass algorithm. Figure 5.16 shows the overhead relative to one pass using the bars. The highest relative overhead was for the Flickr homepage, where it reached almost half the time of one pass. However, layout occurs in multiple passes. For a 5-pass layout engine where we model each pass as similar to the one we optimized, the overhead is amortized. The small gap between the solid and dashed lines in Figure 5.16 show there is little difference when we include the preprocessing overhead in the speedup calculation.

5.5 Related Work

Our MIMD and SIMD algorithms for implementing a known traversal schedule descend from a variety of techniques:

Static and Dynamic Task Scheduling

Kwok and Ahmad (1999) survey statically scheduling a directed acyclic graph (DAG) of tasks for MIMD execution. The problem is NP-complete when nodes contain irregularities such as non-uniform weights. Furthermore, many classic techniques assume shared-nothing message passing and no temporal locality. Our interest in static scheduling techniques stems more from the ability to eliminate dynamic scheduling overheads and perform locality optimizations.

Work stealing (Blumofe et al., 1995; Reinders, 2007) achieves load balancing because underutilized processors dynamically steal tasks from oversubscribed processors. Jourdan and Parigot (1991) apply this idea to implementing the parallel traversals for attribute grammars in the FNC-2 system and report multi-factor speedups. Our experiments saw few-to-no speedups from using the Cilk and TBB work stealers due to poor locality and high overheads. Instead, we semi-statically perform work stealing as a partitioning heuristic for lock-based tiled evaluation, and only then saw speedups. Our approach is more similar to Galois (Kulkarni et al., 2008), which dynamically tiles subgraphs during an iterative graph computation.

Optimizing Data Representation for Locality

Both our SIMD and MIMD algorithms optimize the data representation in order to improve temporal and spatial data locality. Fine-grained approaches such as the structure split coallocation of Chilimbi et al. (1999) and Ding and Kennedy (1999) improve spatial locality. Our SIMD algorithm achieved some of the effect by converting from structs to arrays. For MIMD, we optimized temporal locality by tiling (Irigoin and Triolet, 1988), as opposed to ignoring machine details (Frigo et al., 1999). We also performed thread pinning and tuned the tree layout. Concurrent work by Jo et al describes a similar technique for DAGs (Jo and Kulkarni, 2011): point blocking. They studied large graphs while we found incorporating further optimizations to be useful for small ones.

Nuzman et al. (2006) applied data layout transformations to vectorization. Many such optimizations are known, e.g., structure conversion. We show additional optimizations are possible by using clustering, such as load balancing and eliminating instruction divergence.

SIMD Tree Traversals

Indicative of the challenge of manual approaches, an early paper about manually vectorizing a Barnes-Hut n-body simulation (Barnes, 1990) was titled "Don't laugh, it runs!". As seen with the recent FAST (Kim et al., 2010) algorithm that uses SIMD instructions for the comparator in traversing a binary search tree, effective vectorization of tree computations is still a challenge. Not described in this work, we used an idea similar to our clustering to cluster hot paths down a binary search tree. For predictable workloads, we observed multi factor speedups over FAST. In contrast, FAST performs work-inefficient BFS traversals over every accessed B-node (tree tile).

Blelloch (Blelloch et al., 1994; Chatterjee et al., 1990)'s NESL language demonstrates one transformation for lifting arbitrary computations over bounded nested vectors (e.g., matrices) to use vector instructions. As extensions, **?** and Peyton Jones (2008) support recursive types (e.g., trees) and target multiple cores and GPUs. Catanzaro et al. (2011) presents an alternate transformation in Copperhead. Reps (1993) ran simulations to measure the connection between attribute grammar evaluation and such data parallel execution.

Implementing the idea and running case studies led us to additional techniques: clustering based on instruction sequence and grouping dynamic memory allocations.

Our clustering transformation can be generalized to loop transformation, which is well-studied (Bacon et al., 1994; Allen and Kennedy, 1987; Smith et al., 2000). It essentially introduces an additional loop that permutes and partitions the original parallel interval. Doing so enables guarantees within a subinterval such as lack of instruction divergence or load imbalance. Similar to recent GPU and SIMD schedulers (Zhang et al., 2011), it contains a runtime component. Merrill et al. (2012) examined the more hostile scenario of BFS graph traversal on a GPU where the minimum spanning tree is not known ahead of time and provides relevant optimizations such as in load balancing.

Autotuning

Our MIMD implementation resembles the ATLAS (Whaley et al., 2001) framework for linear algebra in that we autotune over parameters such as block size to optimize for a particular device. In contrast, cache oblivious algorithms for FFTW (Frigo and Johnson, 2005) asymptotically optimize for general architectures but may be less efficient on any individual device. Autotuning is actively being applied to further domains, such as work in stencils (Datta et al., 2008): we examine computations over trees.

Data structure selection is examined early on by Low (1978) for abstract data types in an ALGOL-60 variant. Recent work by Hawkins et al. (2011) examines exposing and implementing multiple pointer-based representation satisfying the same relational interface. Based on discussions with the author, supporting data layout optimizations used by our systems remains a challenge. Not included herein, we made a DSLs for more traditional layout autotuning (e.g., parameter space to explore for cache block size or which type of lock to use).

Chapter 6

Conclusion

This thesis explored the practical question of how to parallelize a layout language and introduced programming language constructs, compiler optimizations, and parallel algorithms for doing so. The core idea is to formalize layout languages as attribute grammars and automatically synthesize a schedule of parallel tree traversals to implement one. Schedule design is non-trivial, so we introduce schedule holes as a new abstraction for parallel programming and a fast synthesis algorithm to fill them in. Once a static schedule is determined, we achieved multifactor speedups by introducing new optimizations for runtime schedules and data representations on MIMD and SIMD architectures. In addition, we identified parallelism in CSS and common layout language features, and measured speedups of up to 54X on commodity hardware.

Bibliography

Alblas, H. (1991). Attribute evaluation methods. In *Proceedings on Attribute Grammars, Applications and Systems*, pages 48–113, London, UK, UK. Springer-Verlag.

Allen, R. and Kennedy, K. (1987). Automatic translation of Fortran programs to vector form. *TOPLAS*.

Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). Petabricks: A language and compiler for algorithmic choice. In *PLDI'09*.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys*.

Badros, G. J., Borning, A., and Stuckey, P. J. (2001). The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306.

Barnes, J. E. (1990). A modified tree code: Don't laugh; it runs. *J. Comput. Phys.*, 87(1):161–170.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J., and Zagha, M. (1994). Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21:4–14.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: an efficient multithreaded runtime system. In *PPOPP'95*, pages 207–216.

Bochmann, G. V. (1976). Semantic evaluation from left to right. *Commun. ACM*, 19(2):55–62.

Boehm, H.-j. and Zwaenepoel, W. (1987). Parallel attribute grammar evaluation. In *Proceedings of the 7th International Conference on Distributed Computing Systems, IEEE*, pages 347–354. Kim,IEEE Computer Society.

Bostock, M., Ogievetsky, V., and Heer, J. (2011). D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309.

Brown, H. (1988). Parallel processing and document layout. *Electron. Publ. Origin. Dissem. Des.*, 1(2):97–104.

Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., and Ball, T. (2011). Two for the price of one: a model for parallel and incremental computation. In *OOPSLA'11*, pages 427–444.

Burg, B. S. L. B. and Schulte, H. V. W. (2011). C3: an experimental, extensible, reconfigurable platform for html-based applications. In *2nd USENIX Conference on Web Application Development*, page 61.

Catanzaro, B., Garland, M., and Keutzer, K. (2011). Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 47–56, New York, NY, USA. ACM.

Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S., Keller, G., and Marlow, S. (2007). Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA. ACM.

Chatterjee, S., Blelloch, G., and Zagha, M. (1990). Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*.

Chilimbi, T. M., Hill, M. D., and Larus, J. R. (1999). Cache-conscious structure layout. In *PLDI*.

Colmenares, J. A., Eads, G., Hofmeyr, S., Bird, S., Moretó, M., Chou, D., Gluzman, B., Roman, E., Bartolini, D. B., Mor, N., Asanović, K., and Kubiatowicz, J. D. (2013). Tessellation: Refactoring the os around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 76:1–76:10, New York, NY, USA. ACM.

Colmerauer, A. (1990). An introduction to Prolog III. *CACM*, 33.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. (2008). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08*.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Demers, A., Reps, T., and Teitelbaum, T. (1981). Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 105–116, New York, NY, USA. ACM.

Ding, C. and Kennedy, K. (1999). Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*.

Eckstein, R., Loy, M., and Wood, D. (1998). *Java Swing*. O'Reilly & Associates, Inc.

Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *IEEE*, 93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–298, Washington, DC, USA. IEEE Computer Society.

Gruber, J. (2004). Daring fireball: Markdown.

Hawkins, P., Aiken, A., Fisher, K., Rinard, M., and Sagiv, M. (2011). Data representation synthesis. In *PLDI'11*.

Heckmann, R. and Wilhelm, R. (1997). A functional description of TEX's formula layout. *Journal of Functional Programming*, 7(5):451–485.

Irigoin, F. and Triolet, R. (1988). Supernode partitioning. In *POPL*.

Jo, Y. and Kulkarni, M. (2011). Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 463–482. ACM.

Jones, C. G., Liu, R., Meyerovich, L., Asanović, K., and Bodík, R. (2009). Parallelizing the web browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 7–7, Berkeley, CA, USA. USENIX Association.

Jourdan, M. (1991). A survey of parallel attribute evaluation methods. In *Proceedings on Attribute Grammars, Applications and Systems*, pages 234–255, London, UK. Springer-Verlag.

Jourdan, M. and Parigot, D. (1991). Internals and externals of the fnc-2 attribute grammar system. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 485–504. Springer Berlin Heidelberg.

Karp, R. M., Miller, R. E., and Winograd, S. (1967). The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590.

Kastens, U. (1980). Ordered attributed grammars. *Acta Informatica*, 13(3):229–256.

Keller, G. and Chakravarty, M. M. T. (1998). Flattening trees. In *Euro-Par*.

Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. (2010). FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*.

Klaiber, A. and Gokhale, M. (1992). Parallel evaluation of attribute grammars. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):206–220.

Klein, E. and Koskimies, K. (1990). Parallel one-pass compilation. In *Proceedings of the International Conference on Attribute Grammars and Their Applications*, WAGA, pages 76–90, New York, NY, USA. Springer-Verlag New York, Inc.

Knuth, D. (1990). The genesis of attribute grammars. In Deransart, P. and Jourdan, M., editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg.

Knuth, D. E. and Bibby, D. (1986). *The TeXbook*, volume 1993. Addison-Wesley Reading, MA, USA.

Koskimies, K. (1991). Object-orientation in attribute grammars. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 297–329. Springer Berlin Heidelberg.

Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., and Chew, L. P. (2008). Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 217–228, New York, NY, USA. ACM.

Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31:406–471.

Lattner, C. and Adve, V. S. (2005). Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 24–35, New York, NY, USA. ACM.

Lie, H. W. and Bos, B. (1997). *Cascading Style Sheets*. Addison Wesley Longman.

Lin, X. (2006). Active layout engine: Algorithms and applications in variable data printing. *Comput. Aided Des.*, 38(5):444–456.

Low, J. (1978). Automatic data structure selection: an example and overview. *CACM*, 21(5):376–385.

Mai, H., Tang, S., King, S. T., Cascaval, C., and Montesinos, P. (2012). A case for parallelizing web pages. In *HotPar'12*.

Massalin, H. (1987). Superoptimizer: a look at the smallest program. *SIGPLAN Not.*, 22:122–126.

Matsuzaki, K., Hu, Z., and Takeichi, M. (2006a). Parallel skeletons for manipulating general trees. *Parallel Computing*.

Matsuzaki, K., Hu, Z., and Takeichi, M. (2006b). Towards automatic parallelization of tree reductions in dynamic programming. In *SPAA*.

Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable GPU graph traversal. In *PPOPP '12*, pages 117–128.

Meyerovich, L. A. and Bodík, R. (2010). Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 711–720, New York, NY, USA. ACM.

Meyerovich, L. A., Torok, M. E., Atkinson, E., and Bodík, R. (2013). Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 187–196, New York, NY, USA. ACM.

Milner, R., Tofte, M., and Macqueen, D. (1997). *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

Noll, T. and Romanith, S. (1996). Parallel evaluation of lr-attributed grammars.

Nuzman, D., Rosen, I., and Zaks, A. (2006). Auto-vectorization of interleaved data for SIMD. In *PLDI*.

Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Patney, A. and Owens, J. D. (2008). Real-time reyes-style adaptive surface subdivision. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 143:1–143:8, New York, NY, USA. ACM.

Peyton Jones, S. (2008). Harnessing the multicores: Nested data parallelism in Haskell. In *APLAS*.

Prountzos, D., Manevich, R., and Pingali, K. (2012). Elixir: a system for synthesizing concurrent graph programs. In *OOPSLA '12*.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

Reps, T. (1993). Scan grammars: parallel attribute evaluation via data-parallelism. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 367–376, New York, NY, USA. ACM.

Reps, T. W., Marceau, C., and Teitelbaum, T. (1986). Remote attribute updating for language-based editors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 1–13, New York, NY, USA. ACM.

Saraiva, J. a. and Swierstra, D. (2003). Generating spreadsheet-like tools from strong attribute grammars. In *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 307–323, New York, NY, USA. Springer-Verlag New York, Inc.

Shiue, L.-J., Jones, I., and Peters, J. (2005). A real-time GPU subdivision kernel. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1010–1015. ACM.

Smith, J. E., Faanes, G., and Sugumar, R. (2000). Vector instruction set support for conditional operations. In *ISCA*.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S., and Saraswat, V. (2006). Combinatorial sketching for finite programs. In *ASPLOS-XII*, pages 404–415.

Sutherland, I. E. (1963). Sketchpad: a man-machine graphical communication system. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, AFIPS '63 (Spring), pages 329–346, New York, NY, USA. ACM.

Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1989). Higher order attribute grammars. *SIGPLAN Not.*, 24(7):131–145.

Warth, A. and Piumarta, I. (2007). Ometa: an object-oriented language for pattern matching. In *Proceedings of the 2007 Symposium on Dynamic Languages*, pages 11–19. ACM.

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. (2011). On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*.

Appendix A

Layout Grammars

A.1 Sunburst

The following attribute grammar demonstrates an animated radial layout. Subtrees expand and contract.

```
1  schedule {
2     "P = [(_,td,_,_,_),(_,bu,_,_,_),(_,td,_,_,_)]"
3  }
4  interface Node {
5    input open : float;
6    var show : float;

8    var r : float;
9    var parentTotR : float;
10   var alpha : float;
11   var sectorAng : float;
12   var maxR : float;

14   input bgcolor : color;

16   var rootCenterX : float;
17   var rootCenterY : float;
18 }

20 class Radial : Node {
21   children {
22     child : [ Node ];
23   }
24   attributes {
25     var numOpenChildren : int;
26   }

28   actions {

30     r := (maxR - parentTotR)/3 < 10 ? 10 : (maxR - parentTotR)/4;

32     loop child {
33       //subtreeWeight := fold 1 .. $-.subtreeWeight + child$i.subtreeWeight;

35       child.parentTotR := parentTotR + r;
```

```
37        child.rootCenterX := rootCenterX;
38        child.rootCenterY := rootCenterY;

40        child.show := show * child$i.open;


43        child.maxR := maxR;

45        child.sectorAng :=  $$.numOpenChildren > 0.01 ?
46          child$i.show * sectorAng / $$.numOpenChildren : 0;

48        child.alpha :=
49          fold
50             $$.numOpenChildren > 0.01?
51              alpha - (sectorAng / 2.0f) - (sectorAng/($$.numOpenChildren*2))
52              : 0
53          ..
54          $$.numOpenChildren > 0.01 ?
55            child$-.alpha + child$i.show * (sectorAng/$$.numOpenChildren)
56            : 0;

58        numOpenChildren := fold 0 .. $-.numOpenChildren + child$i.show;
59      }
60      @render @Arc(rootCenterX, rootCenterY, show * (parentTotR + r), alpha, sectorAng, (show
61    }
62  }


65  interface IRoot { }

67  class Root : IRoot {
68    children {
69      child : Node;
70    }

72    attributes {
73      input radius : float;
74      input centerRadius : float;
75      input centerAlpha : float;

77      input w : float;
78      input h : float;
79    }

81    actions {
82      child.alpha := 45;
83      child.maxR := radius;
84      child.parentTotR := 0;
85      child.sectorAng := 360.0f;

87      child.show := 1.0;

89      child.rootCenterX := centerRadius * cos(PI() * centerAlpha / 180);
90      child.rootCenterY := centerRadius * sin(PI() * centerAlpha / 180);
91    }
92  }
```

A.2 Table Layout

The following attribute grammar demonstrates the automatic table layout algorithm in HTML and CSS.

```
schedule {
   "P = [(_,td,_,_,_),(_,td,_,_,_),(_,bu,_,_,_),(_,td,_,_,_),
         (_,td,_,_,_),(_,bu,_,_,_),(_,td,_,_,_)]"
}

/*****
standard interfaces and traits
******/
interface Node {
  var canvas : int;
  var render : int;
  input height : ?int;
  var intrinsHeight : int;
  var computedHeight : int;
  var relRightX : int;
  var relX : int;
  var absX : int;
  var relBotY : int;
  var relY : int;
  var absY : int;

  input minWidth : ?int;
  input maxWidth : ?int;
  input percentWidth : ?int;
  input width : ?int;
  var intrinsPrefWidth : int;
  var intrinsMinWidth : int;
  var availableWidth : int;
  var computedWidth : int;

  var lineH : int;
}

trait shrinkToFitHeightWidth {
  actions {
    computedWidth :=
      !isEmptyInt(width) ?
        valueInt(width) :
      (!isEmptyInt(percentWidth) ?
        (0.01 * valueInt(percentWidth) * availableWidth) :
      min(
        max(
          max(intrinsMinWidth, !isEmptyInt(minWidth) ? valueInt(minWidth) : 0),
          availableWidth),
        min(intrinsPrefWidth, !isEmptyInt(maxWidth) ? valueInt(maxWidth) : intrinsPrefWidth
    computedHeight := isEmptyInt(height) ? intrinsHeight : valueInt(height);
  }
}

trait relToAbsChilds {
  actions {
    loop childs {
      childs.absY := absY + childs$i.relY;
      childs.absX := absX + childs$i.relX;
```

```
55      }
56    }
57  }

59  trait strokeBox {
60    actions {
61      render :=
62        canvas
63        + paintLine (absX, absY, absX+computedWidth, absY, borderc) //top
64        + paintLine (absX+computedWidth, absY, absX+computedWidth, absY+computedHeight, borde
65        + paintLine (absX+computedWidth, absY+computedHeight, absX, absY+computedHeight, borc
66        + paintLine (absX, absY+computedHeight, absX, absY, borderc); //left
67    }
68  }

70  trait countChilds {
71    attributes {
72      var numChilds : int;
73    }
74    actions {
75      loop childs {
76        numChilds := fold 0 .. $-.numChilds + 1;
77      }
78    }
79  }



83  /***************************
84  some typical fully-handled box nodes
85  ***************************/
86  interface Root {
87    input w : int = 100;
88    input h : int = 100;
89  }

91  class Top : Root {
92    children { child : Node }
93    actions {
94      child.relRightX := 0;
95      child.relX := 0;
96      child.absX := 0;
97      child.relBotY := 0;
98      child.relY := 0;
99      child.absY := 0;
100     child.canvas := paintStart(child.computedWidth, child.computedHeight);
101     child.availableWidth := w; //FIXME
102   }
103 }



107 trait Wrapping {
108   children { childs : [ Node ]; }
109   actions {
110     loop childs {
111       intrinsPrefWidth :=
112         fold
113           ($$.numChilds == 0 ? 10 : 5)
114           ..
```

```
115             self$-.intrinsPrefWidth + childs$i.intrinsPrefWidth + 5;
116         intrinsMinWidth :=
117           fold
118             ($$.numChilds == 0 ? 10 : 5)
119             ..
120             max(self$-.intrinsMinWidth, 5 + childs$i.intrinsMinWidth + 5);

122         childs.relRightX :=
123           fold
124             0 ..
125             (childs$-.relRightX + 5 + childs$i.computedWidth > computedWidth) ?
126               (5 + childs$i.computedWidth) : (childs$-.relRightX + 5 + childs$i.computedWidth
127         childs.relX := childs$i.relRightX - childs$i.computedWidth;

129         childs.lineH := fold 0
130             .. childs$i.relX == 5 ?
131               childs$i.computedHeight
132               : (childs$i.computedHeight > childs$-.lineH ?
133                  childs$i.computedHeight : childs$-.lineH);
134         childs.relY := fold 0
135             .. childs$-.relY + (childs$i.relX == 5 ? childs$-.lineH + 5 : 0);
136         childs.relBotY := fold 0 .. childs$i.relY + childs$i.computedHeight;
137         childs.canvas := fold render .. childs$-.canvas;
138         intrinsHeight := fold 10 .. childs$i.relY + childs$i.lineH + 5;
139         childs.availableWidth := computedWidth - 10;
140       }
141     }
142 }


145 /****
146 table stuff
147 *****/

149 interface CellI {

151     input colSpan : ?int;
152     input rowSpan : ?int;

154     //Node
155     var canvas : int;
156     var render : int;
157     input height : ?int;
158     var intrinsHeight : int;
159     var computedHeight : int;
160     var relX : int;
161     var absX : int;
162     var relBotY : int;
163     var relY : int;
164     var absY : int;


167     input width : ?int;
168     input minWidth : ?int;
169     input maxWidth : ?int;
170     input percentWidth : ?int;
171     var intrinsPrefWidth : int;
172     var intrinsMinWidth : int;
173     var availableWidth : int;
174     var computedWidth : int;
```

```
176       var cellNum : int;
177       var column : int;
178       var row : int;

180   }
181   interface RowI {
182     var intrinsColCount : int;
183     var colCount : int;
184     input height : ?int;
185     var intrinsHeight : int;
186     var computedHeight : int;
187     var relBotY : int;
188     var relY : int;
189     var absY : int;
190     var absX : int;

192     var rowNum : int;
193     var cells : int;
194     var colAssignment : int;

196     var canvas : int;
197     var render : int;

199     var computedWidth : int;
200     var tableContentWidth : int;

202   }


205   class Cell(shrinkToFitHeightWidth, relToAbsChilds, strokeBox, countChilds, Wrapping) : Cell
206     attributes {
207       input borderc : color = #777;
208     }
209   }

211   class Row : RowI {
212     attributes {
213       input borderc : color = #070;
214     }
215     children {
216       childs : [ CellI ];
217     }
218     phantom { //do not emit code for these
219       childs.relX;
220       childs.absX;
221       childs.availableWidth;
222     }
223     actions {
224       loop childs {
225         intrinsColCount :=
226           fold 0 .. $-.intrinsColCount +
227             (isEmptyInt(childs$i.colSpan) ? 1 : valueInt(childs$i.colSpan));
228         intrinsHeight :=
229           fold 10
230           ..
231           (isEmptyInt(childs$i.rowSpan)  valueInt(childs$i.rowSpan) == 1) ?
232           max($-.intrinsHeight, childs$i.computedHeight + 10)
233           : $-.intrinsHeight;
234         childs.relBotY := fold 0 .. 5 + childs$i.computedHeight;
```

```
235         childs.relY := fold 0 .. 5;
236         childs.absY := absY + childs$i.relY;
237       }
238       computedHeight := isEmptyInt(height) ? intrinsHeight : valueInt(height);

240       loop childs {
241         computedWidth := fold 0 .. $-.computedWidth + childs$i.intrinsMinWidth;
242       }

244       loop childs {
245         cells :=
246           fold
247             mtIntPairList()
248             ..
249             appendIntPairList(
250               $-.cells,
251               pairInt(
252                 isEmptyInt(childs$i.rowSpan) ? 1 : valueInt(childs$i.rowSpan),
253                 isEmptyInt(childs$i.colSpan) ? 1 : valueInt(childs$i.colSpan)));
254         childs.column := columnsGetCol(colAssignment, childs.cellNum);
255         childs.row := rowNum;
256         childs.cellNum := fold 0 .. childs$-.cellNum + 1;
257       }

259       render :=
260         canvas
261         + paintLine (absX, absY, absX+tableContentWidth, absY, borderc) //top
262         + paintLine (absX+tableContentWidth, absY, absX+tableContentWidth,
263                 absY+computedHeight, borderc)
264         + paintLine (absX+tableContentWidth, absY+computedHeight, absX,
265                 absY+computedHeight, borderc)
266         + paintLine (absX, absY+computedHeight, absX, absY, borderc); //left
267     }
268   }

270   interface ColI {
271     var colCount : int;
272     var colNum : int;

274     var intrinsMinWidth : int;
275     var intrinsPrefWidth : int;
276     var availableWidth : int;
277     var computedWidth : int;

279     var intrinsHeight : int;
280     var computedHeight : int;

282     var relRightX : int;
283     var relX : int;
284     var relY : int;
285     var absX : int;
286     var absY : int;
287     var canvas : int;
288     var render : int;

290     var cellsReady : int;
291     var tableContentHeight : int;

293     var borderc : color;
294   }
```

```
296 class Col(shrinkToFitHeightWidth, countChilds) : ColI {
297   attributes {
298     input width : ?int;
299     input percentWidth : ?int;
300     input minWidth : ?int;
301     input maxWidth : ?int;
302     input height : ?int;
303   }
304   phantom {
305     childs.column;
306     childs.row;
307     childs.cellNum;
308     childs.relBotY;
309     childs.relY;
310     childs.absY;
311   }
312   children {
313     childs : [ CellI ]; //  ../rows/childs[.column == self.colNum]
314   }
315   actions {
316     loop childs {
317       intrinsMinWidth :=
318         fold
319           10
320           ..
321           (isEmptyInt(childs$i.colSpan)  valueInt(childs$i.colSpan) == 1) ?
322               max($-.intrinsMinWidth, 10 + childs$i.intrinsMinWidth)
323             : $-.intrinsMinWidth;
324       intrinsPrefWidth :=
325         fold
326           10
327           ..
328           (isEmptyInt(childs$i.colSpan)  valueInt(childs$i.colSpan) == 1) ?
329               max($-.intrinsPrefWidth, 10 + childs$i.intrinsPrefWidth)
330             : $-.intrinsPrefWidth;

332       intrinsHeight := fold 5 + ($$.numChilds == 0 ? 0 : 0) ..
333         $-.intrinsHeight + childs$i.computedHeight + 5;
334       childs.availableWidth := computedWidth;

336       childs.relX := 5;
337       childs.absX := absX + childs$i.relX;

339       childs.canvas := fold render .. childs$-.canvas;
340     }

342     render :=
343       canvas
344       + paintLine (absX, absY, absX+computedWidth, absY, borderc) //top
345       + paintLine (absX+computedWidth, absY, absX+computedWidth,
346                 absY+tableContentHeight, borderc)
347       + paintLine (absX+computedWidth, absY+tableContentHeight, absX,
348                 absY+tableContentHeight, borderc)
349       + paintLine (absX, absY+tableContentHeight, absX, absY, borderc); //left

351   }
352 }
```

```
355 interface ColsI {
356   var intrinsPrefWidth : int;
357   var intrinsMinWidth : int;

359   var colCount : int;
360   var availableWidth : int;
361   var absX : int;
362   var absY : int;
363   var canvas : int;

365   var cellsReady : int;
366   var tableContentHeight : int;
367   var tableContentWidth : int;
368 }
369 class Cols : ColsI {
370   attributes {
371     input borderc : color = #F00;
372   }
373   children {
374     cols : [ ColI ];
375   }
376   actions {
377     //do not know until colCount, put in dep
378     loop cols {
379       cols.borderc := borderc;
380       cols.colCount := colCount;
381       intrinsPrefWidth :=
382         fold
383           10
384           ..
385           $-.intrinsPrefWidth + cols$i.intrinsPrefWidth;
386       intrinsMinWidth :=
387         fold
388           10
389           ..
390           $-.intrinsMinWidth + cols$i.intrinsMinWidth;
391       cols.colNum := fold 0 .. cols$-.colNum + 1;
392       cols.availableWidth := availableWidth;

394       cols.relRightX := fold 5 .. cols$-.relRightX + cols$i.computedWidth;
395       cols.relX := fold 0 .. cols$i.relRightX - cols$i.computedWidth;
396       cols.absX := absX + cols$i.relX;

398       cols.relY := 5;
399       cols.absY := absY + cols$i.relY;
400       cols.canvas := fold canvas .. cols$-.canvas;
401       cols.cellsReady := cellsReady;
402       cols.tableContentHeight := tableContentHeight;
403       tableContentWidth := fold 0 .. $-.tableContentWidth + cols$i.computedWidth;
404     }
405   }
406 }


409 class TableBox(shrinkToFitHeightWidth, strokeBox) : Node {
410   attributes {
411     input borderc : color = #DDD;
412     var colCount : int;
413     var cellsReady : int;
414   }
```

```
415     children {
416       rows : [ RowI ];
417       columns : ColsI;
418     }
419     actions {
420       loop rows {
421         colCount := fold 0 .. max($-.colCount, rows$i.intrinsColCount);
422         rows.colCount := $$.colCount;
423         intrinsHeight := fold 10 .. $-.intrinsHeight + rows$i.computedHeight;
424         rows.relBotY := fold 5 .. rows$-.relBotY + rows$i.computedHeight;
425         rows.relY := fold 0 .. rows$i.relBotY - rows$i.computedHeight;
426         rows.absY := absY + rows$i.relY;
427         rows.absX := absX + 5;
428         rows.colAssignment :=
429           fold
430             emptyColumnList(colCount)
431             ..
432             columnsAppendRow(rows$-.colAssignment, rows$i.cells, rows$i.rowNum);
433         rows.rowNum := fold 0 .. rows$-.rowNum + 1;
434         rows.canvas := fold render .. rows$-.canvas;
435         cellsReady := rows$$.colAssignment;
436         rows.tableContentWidth := columns.tableContentWidth;
437       }

439       intrinsPrefWidth := columns.intrinsPrefWidth;
440       intrinsMinWidth := columns.intrinsMinWidth;
441       columns.colCount := colCount;
442       columns.cellsReady := cellsReady ? true : true;
443       columns.availableWidth := computedWidth;
444       columns.absX := absX;
445       columns.absY := absY;
446       columns.canvas := render;
447       columns.tableContentHeight := intrinsHeight - 10;
448     }
449   }
```

A.3 Multiple Time Series

The following attribute grammar demonstrates 3D layout of multiple time series data. Interactive controls enable examining different subintervals.

```
1  schedule {
2     "P = [(_,td,_,_,_),(_,bu,_,_,_),(_,td,_,_,_)]"
3  }
4  interface IRoot {

6    input xOffset : int;
7    input yOffset : int;

9      input tweenMin : float;
10     input tweenMax : float;

12     input radius : float;
13     input minRadius : float;
14     input height : float;
15     input rotation : int;
```

```
17    ///////

19      var depth : int;

21  }

23  interface SecondI {
24    var xOffset : int;
25    var yOffset : int;
26    var tweenMin : float;
27    var tweenMax : float;
28    var radius : float;
29    var minRadius : float;
30    var height : float;
31    var rotation : int;


34    var w : float;
35    var x : float;
36    var rx : float;

38      var depth : int;
39  }


42  interface Node {
43    var xOffset : int;
44    var yOffset : int;
45    var tweenMin : float;
46    var tweenMax : float;
47    var radius : float;
48    var minRadius : float;
49    var height : float;
50    var rotation : int;

52    var x : float;
53    var rx : float;

55      var depth : int;
56      var levelLength : int;
57      var idx : int;
58  }

60  interface Leaf {
61    var xOffset : int;
62    var yOffset : int;
63    var tweenMin : float;
64    var tweenMax : float;
65    var tween : float;
66    var radius : float;
67    var minRadius : float;
68    var height : float;
69    var rotation : int;

71    input val : float;


74    var valPrev : float;
75    var valCopy : float;
76    var angle : float;
```

```
77    var increment : float;

79    var isFirst : int;
80    var isLast : int;

82    //////

84      var depth : int;
85      var idx : int;
86  }


89  trait propagateChilds {
90    actions {
91      loop childs {
92        childs.xOffset := xOffset;
93        childs.yOffset := yOffset;
94        childs.tweenMin := tweenMin;
95        childs.tweenMax := tweenMax;
96        childs.radius := radius;
97        childs.minRadius := minRadius;
98        childs.height := height;
99        childs.rotation := rotation;
100     }
101   }
102 }
103 trait propagateChild {
104   actions {
105     child.xOffset := xOffset;
106     child.yOffset := yOffset;
107     child.tweenMin := tweenMin;
108     child.tweenMax := tweenMax;
109     child.radius := radius;
110     child.minRadius := minRadius;
111     child.height := height;
112     child.rotation := rotation;
113   }
114 }

116 trait propagateIntermediate{
117     actions{
118         loop childs {
119             childs.depth := depth + 1;
120         }
121     }
122 }

124 class Root(propagateChild) : IRoot {
125   attributes {}
126   children {
127     child : SecondI ;
128   }
129   actions {

131     child.rx := 5 + child.w;
132     child.x := child.rx - child.w;
133     child.depth := depth + 1;
134     depth := 1;
135   }
136 }
```

```
138 class Second (propagateIntermediate,propagateChilds) : SecondI {
139   attributes {
140     var len : int;
141   }
142   children {
143     childs : [ Node ];
144   }
145   actions {
146     loop childs {
147       len := fold 0 .. $-.len + 1;
148       childs.levelLength := $$.len;
149     childs.idx := fold 0 .. childs$-.idx + 1;
150     }

152   //background
153   @render @RectangleZ(xOffset - radius, yOffset - radius,
154     4 * radius, 4 * radius, -0.1f, rgb(0,0,0));
155   w := 10;
156     loop childs {
157       childs.rx := fold x .. childs$-.rx + 5 + 10;
158       childs.x := childs$i.rx - 10;
159     }
160   }
161 }

163 // [a,b] of i for [l%,h%] out of n
164 // i < l% * n  OR  i > h% * n:
165 //   off
166 // else:
167 //   on, tween = (i - l%n) / (h%n - l%n)
168 class Generator (propagateIntermediate,propagateChilds) : Node {
169   attributes {
170     var numSpikes : int;
171     var increment : float;
172     var angle : float;
173   }
174   children {
175     childs : [ Leaf ];
176   }
177   actions {
178     numSpikes := childs$$.idx;
179     increment := 1.0f / ((tweenMax - tweenMin) * numSpikes);
180     angle := 2 * PI() * ((rotation / 360.0f) + (idx + 0.0f) / levelLength);
181     loop childs {
182       childs.idx := fold 0 .. childs$-.idx + 1;
183       childs.increment := increment;

185       childs.tween :=
186           ((childs$i.idx < tweenMin * numSpikes)  (childs$i.idx > tweenMax * numSpikes))
187         ? -1.0f
188         : ((childs.idx - tweenMin * childs$$.idx)
189           / (tweenMax * numSpikes - tweenMin * numSpikes));

191       childs.isFirst :=
192         ((childs$i.idx >= tweenMin * numSpikes)
193           && ((childs$i.idx - 1) < tweenMin * numSpikes)) ? 1 : 0;
194       childs.isLast :=
195         ((childs$i.idx <= tweenMax * numSpikes)
196           && ((childs$i.idx + 1) > tweenMax * numSpikes)) ? 1 : 0;
```

```
198         childs.angle := 0.01f + angle;
199         childs.valCopy := fold 0 .. childs$i.val;
200         childs.valPrev := fold 0 .. childs$-.valCopy;
201       }
202     }
203   }
204   class Spike : Leaf {
205     attributes {
206       var isOn : int;
207       var enableShadow : int;
208     }
209     actions {
210       enableShadow := 0;
211       isOn := tween >= 0.0f ? 1 : 0;

213       @render (isOn != 1) ? 0 :  //segment
214         @Line3D(xOffset + 1.5 + radius + minRadius * cos(angle) + (radius *
215             ((isLast == 1) ? (1.0f + increment/1.0f) : (tween + increment/1.0f))
216             ) * cos(angle),
217           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius *
218             ((isLast == 1) ? (1.0f + increment/1.0f) : (tween + increment/1.0f))
219             ) * sin(angle),
220           height * val,
221           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius *
222             ((isFirst == 1) ? increment/1.0f : tween)
223             ) * cos(angle),
224           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius *
225             ((isFirst == 1) ? increment/1.0f : tween)
226             ) * sin(angle),
227           height * valPrev,
228           0.15f, rgba(0, 204, 255, 102));
229       @render (isOn != 1  enableShadow != 1) ? 0 : //shadow
230         @Line3D(
231           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius *
232             ((isLast == 1) ? (1.0f + increment/1.0f) : (tween + increment/1.0f))
233             ) * cos(angle),
234           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius *
235             ((isLast == 1) ? (1.0f + increment/1.0f) : (tween + increment/1.0f))
236             ) * sin(angle),
237           0.0f,
238           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius *
239               ((isFirst == 1) ? increment/1.0f : tween)
240             ) * cos(angle),
241           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius *
242               ((isFirst == 1) ? increment/1.0f : tween)
243             ) * sin(angle),
244           0.0f,
245           0.1f, rgb(255, 255, 255));
246       @render (isOn != 1  isLast != 1) ? 0 : //vertical ending
247         @Line3D(
248           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius * (1.0f + increment/1.0f)
    ) * cos(angle),
249           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius * (1.0f + increment/1.0f)
    ) * sin(angle),
250           height * val,
251           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius * (1.0f + increment/1.0f)
    ) * cos(angle),
252           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius * (1.0f + increment/1.0f)
    ) * sin(angle),
```

```
253           0.0f,
254           0.1f, rgba(255, 255, 255, 125));
255       @render (isOn != 1  isFirst != 1) ? 0 : // vertical beginning
256         @Line3D(
257           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius *
258             increment/1.0f
259             ) * cos(angle),
260           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius *
261             increment/1.0f
262             ) * sin(angle),
263           height * valPrev,
264           xOffset + 1.5 + radius + minRadius * cos(angle) + (radius *
265             increment/1.0f
266             ) * cos(angle),
267           yOffset + 1.5 + radius + minRadius * sin(angle) + (radius *
268             increment/1.0f
269             ) * sin(angle),
270           0.0f,
271           0.1f, rgba(0, 204, 255, 102));
272     }
273   }
```

A.4 Tree Map

The following attribute grammar demonstrates an interactive treemap. Interactive controls enable toggling which value to use for each node and filtering which nodes to show by performing value comparisons.

```
1  schedule {
2     "P = [(_,td,_,_,_),(_,bu,_,_,_),(_,td,_,_,_),(_,bu,_,_,_),(_,td,_,_,_)]"
3  }
4  interface IRoot {
5    input width : float;
6    input height : float;

8    // Only show polling placee with turnout from (minTurnout, maxTurnout]
9    input minTurnout : float;
10   input maxTurnout : float;

12   // Change color of fraudulent nodes to fraudColor
13   input showFraud : float;

15   // [0.0-1.0] When at 1.0, fraudulent nodes have color correspond to
16   // projected non-fradulent votes, instead of their actual value
17   input showProjected : float;

19   var votesUR : float;

21   // If true, width as data resizes stays fixed (but height may vary);
22   //if false, width varies along with height.
23   input fixWidth : int;

25   // Tween value. When at 0, only shows Javascript simulation nodes;
26   //when at 1, all nodes shown as normal.
27   input showJavascript : float;

29   // Here in the top so we can easily read it in host code
```

```
30    var totalMag : float;
31  }

33  interface Node{
34    var totalMag : float;

36    var minTurnout : float;
37    var maxTurnout : float;

39    var votesUR : int;

41    var showFraud : float;
42    var showProjected : float;

44    var fixWidth : int;

46    var showJavascript : float;

48    var w : float;
49    var h : float;
50    var x : float;
51    var rx : float;
52    var y : float;
53    var by : float;
54  }

56  trait tweenMagnitude{
57    actions{
58      loop childs{
59        totalMag := fold 0 .. $-.totalMag + childs$i.totalMag;
60        childs.minTurnout := minTurnout;
61        childs.maxTurnout := maxTurnout;

63        childs.showFraud := showFraud;
64        childs.showProjected := showProjected;

66        childs.fixWidth := fixWidth;

68        childs.showJavascript := showJavascript;

70        votesUR := fold 0 .. $-.votesUR + childs$i.votesUR;
71      }
72    }
73  }

75  class Root : IRoot {
76    children { childs : Node; }
77    attributes{
78    }
79    actions{
80      childs.w := (fixWidth != 0) ? width : width * (totalMag / 63895164);
81      // Make height a function of the current totalMag and our pre-computed
82      // default totalMag
83      childs.h := height * (totalMag / 63895164);
84      childs.rx := width;
85      childs.by := height;

87      childs.minTurnout := minTurnout;
88      childs.maxTurnout := maxTurnout;
```

```
90        childs.showFraud := showFraud;
91        childs.showProjected := showProjected;

93        childs.fixWidth := fixWidth;
94        childs.showJavascript := showJavascript;

96        totalMag := childs.totalMag;
97        votesUR := childs.votesUR / totalMag;
98      }
99    }

101   class CountryContainer(tweenMagnitude) : Node{
102     children {childs : [Node];}
103     attributes{
104     }
105     actions{
106       x := rx - w;
107       y := by - h;

109       @render fixWidth != 0 ? @RectangleOutline(x, y, w, h, rgb(0,0,0)) : 0;

111       loop childs{
112         childs.w := (childs$i.totalMag / totalMag) * w;
113         childs.h := h;
114         childs.rx := fold x .. childs$-.rx + childs$i.w;
115         childs.by := y + h;
116       }
117     }
118   }

120   class Region(tweenMagnitude) : Node{
121     children {childs : [Node];}
122     attributes{
123     }
124     actions{
125       x := rx - w;
126       y := by - h;

128       loop childs{
129         childs.w := w;
130         childs.h := (childs$i.totalMag / totalMag) * h;
131         childs.rx := x + w;
132         childs.by := fold y .. childs$-.by + childs$i.h;
133       }
134     }
135   }

137   class District(tweenMagnitude) : Node{
138     children {childs : [Node];}
139     attributes{
140     }
141     actions{
142       x := rx - w;
143       y := by - h;

145       @render fixWidth != 0 ? @RectangleOutline(x, y, w, h, rgb(0,0,0)) : 0;

147       loop childs{
148         childs.w := (childs$i.totalMag / totalMag) * w;
149         childs.h := h;
```

```
150         childs.rx := fold x .. childs$-.rx + childs$i.w;
151         childs.by := y + h;
152       }
153     }
154   }

156   class VSquare(tweenMagnitude) : Node{
157     children {childs : [Node];}
158     attributes{
159     }
160     actions{
161       x := rx - w;
162       y := by - h;

164       loop childs{
165         childs.w := w;
166         childs.h := (childs$i.totalMag / totalMag) * h;
167         childs.rx := x + w;
168         childs.by := fold y .. childs$-.by + childs$i.h;
169       }
170     }
171   }

173   class HSquare(tweenMagnitude) : Node{
174     children {childs : [Node];}
175     attributes{
176     }
177     actions{
178       x := rx - w;
179       y := by - h;

181       loop childs{
182         childs.w := (childs$i.totalMag / totalMag) * w;
183         childs.h := h;
184         childs.rx := fold x .. childs$-.rx + childs$i.w;
185         childs.by := y + h;
186       }
187     }
188   }

190   class PollingPlace : Node{
191     attributes{
192       // Total number of ballots cast in this place
193       input totalVotes : int;
194       // Number of ballots cast for UR
195       input totalVotesUR : int;
196       // Percent of votes for UR
197       input urVotes : float;
198       // Average of district's percent of votes for UR (e.g., projected UR vote %)
199       input urVotesProjected : float;
200       // Percent of registered voters who cast a ballot in this place
201       input turnout : float;

203       // Default color (e.g., color when % of votes for UR is 0%)
204       input defColor : color;
205       // UR colors (e.g., color when % of votes for UR is 100%)
206       input urColor : color; // Red
207       // Color to turn fraudulent nodes
208       input fraudColor : color;
```

```
210       // Bool-like int to let us know if this node should be rendered in our JS simulation
211       input inJavascript : int;

213       var calcRegularColor : color;
214       var calcFraudColor: color;
215       var calcProjectedColor : color;
216       var calcVotesColor : color;

218       var magnitude : float;
219     }

221     actions{
222       calcProjectedColor := lerpColor(defColor, urColor, urVotesProjected);
223       calcRegularColor := lerpColor(defColor, urColor, urVotes);
224       calcVotesColor := turnout > 0.83f ?
225         lerpColor(calcRegularColor, calcProjectedColor, showProjected)
226         : calcRegularColor;
227       calcFraudColor := (turnout > 0.83f) ? fraudColor : calcVotesColor;

229       @render @Rectangle(x, y, w, h,
230         255*256*256*256 + lerpColor(calcVotesColor, calcFraudColor, showFraud));

232       x := rx - w;
233       y := by - h;

235       magnitude := (turnout > minTurnout && turnout <= maxTurnout) ? totalVotes : 0;
236       totalMag := (inJavascript != 0) ? magnitude : showJavascript * magnitude;

238       // How many votes for UR does this node contribute?
239       // Turn to 0 if we're not showing this bin
240       // If this is a suspect polling place, interpolate between real and
241       // projected values based off showProjected.
242       votesUR := (turnout > minTurnout && turnout <= maxTurnout) ?
243         ((turnout > 0.83f) ?
244           ((totalVotesUR * (1 - showProjected))
245             + ((totalVotes * urVotesProjected) * showProjected))
246           : totalVotesUR)
247         : 0;
248     }

250 }
```

A.5 Box Model

The following attribute grammar demonstrates specifying a subset of the CSS box model. It is sufficient to render static content from Wikipedia and Wordpress blog.

```
1  schedule{
2    "BUS = [W,X,Y,Z], member(blockimg,BUS), member(normalblock, BUS),
3     member(flowblock,BUS), member(root,BUS),
4     P = [(_,td,_,_,_),(_,td,_,_,_),(_,td,_,_,_),(_,bu,_,_,_),
5          (_,td,_,_,_),(_,td,_,_,_),(_,bu,_,_,_),(_,buSubInorder,_,_,((BUS,_),_)),
6          (_,td,_,_,_)]"
7  }

9  interface Block{
10   var canvas : int;
```

```
11    var render : int;

13    var absX : int;
14    var absY : int;
15    var computedX : int;
16    var computedY : int;

18    var availableWidth : int;
19    var containHeight : int;
20    var computedWidth : int;
21    input width : taggedInt = {1,0};
22    var computedHeight : int;
23    input height : taggedInt = {1,0};

25    var intrinsPrefWidth : int;
26    var intrinsMinWidth : int;
27    var intrinsHeight : int;

29    var inhFontSize : int;
30    input intrinsFontSize : taggedInt = {2,0};

32    var inhColor : color;
33    input color : ? color;

35    input position : string = "static";
36    input left : taggedInt = {1,0};
37    input right : taggedInt = {1,0};
38    input top : taggedInt = {1,0};
39    input bottom : taggedInt = {1,0};

41    //box model
42    input borderc : ?color;
43    input borderw : int;
44    input borders : string = "none";
45    input bgc : ?color;

47    input marginTop : taggedInt = {302,0};
48    input marginBottom : taggedInt = {302,0};
49    input marginLeft : taggedInt = {302,0};
50    input marginRight : taggedInt = {302,0};
51    var mt : int;
52    var mb : int;
53    var ml : int;
54    var mr : int;
55    input paddingTop : taggedInt = {302,0};
56    input paddingBottom : taggedInt = {302,0};
57    input paddingLeft : taggedInt = {302,0};
58    input paddingRight : taggedInt = {302,0};
59    var pt : int;
60    var pb : int;
61    var pl : int;
62    var pr : int;

64    var childNum : int;
65  }


68  interface FlowRoot{
69    var canvas : int;
70    var render : int;
```

```
72    var relRightX : int;
73    var relX : int;
74    var relRightY : int;
75    var relY : int;
76    var oldLineH : int;
77    var maxLineH : int;

79    var firstChildWidth : int;
80    var rightPadding : int;

82    var minX : int;
83    var minY : int;
84    var maxWidth : int;
85    var containHeight : int;

87    var intrinsPrefWidth : int;
88    var intrinsMinWidth : int;
89    var intrinsHeight : int;

91    var inhFontSize : int;
92    input intrinsFontSize : taggedInt = {2,0};


95    var inhColor : color;
96    input color : ?color = "inherit";

98    input position : string = "static";
99    //box model
100   input marginTop : taggedInt = {302,0};
101   input marginBottom : taggedInt = {302,0};
102   input marginLeft : taggedInt = {302,0};
103   input marginRight : taggedInt = {302,0};
104   var mt : int;
105   var mb : int;
106   var ml : int;
107   var mr : int;
108   input paddingTop : taggedInt = {302,0};
109   input paddingBottom : taggedInt = {302,0};
110   input paddingLeft : taggedInt = {302,0};
111   input paddingRight : taggedInt = {302,0};
112   var pt : int;
113   var pb : int;
114   var pl : int;
115   var pr : int;
116 }

118 interface Inline{
119   var canvas : int;
120   var render : int;

122   var relRightX : int;
123   var relX : int;
124   var relRightY : int;
125   var relY : int;
126   var oldLineH : int;
127   var maxLineH : int;

129   var firstChildWidth : int;
130   var rightPadding : int;
```

```
132     var minX : int;
133     var minY : int;
134     var maxWidth : int;
135     var containHeight : int;

137     var intrinsPrefWidth : int;
138     var intrinsMinWidth : int;
139     var intrinsHeight : int;

141     var inhFontSize : int;
142     input intrinsFontSize : taggedInt = {2,0};

144     var inhColor : color;
145     input color : ?color = "inherit";

147     //Relative positioning
148     var offsetX : int;
149     var offsetY : int;
150     var inhOffsetX : int;
151     var inhOffsetY : int;

153     input position : string = "static";
154     input left : taggedInt = {1,0};
155     input right : taggedInt = {1,0};
156     input top : taggedInt = {1,0};
157     input bottom : taggedInt = {1,0};


160     //box model
161     input marginTop : taggedInt = {302,0};
162     input marginBottom : taggedInt = {302,0};
163     input marginLeft : taggedInt = {302,0};
164     input marginRight : taggedInt = {302,0};
165     var mt : int;
166     var mb : int;
167     var ml : int;
168     var mr : int;
169     input paddingTop : taggedInt = {302,0};
170     input paddingBottom : taggedInt = {302,0};
171     input paddingLeft : taggedInt = {302,0};
172     input paddingRight : taggedInt = {302,0};
173     var pt : int;
174     var pb : int;
175     var pl : int;
176     var pr : int;

178     var childNum : int;
179   }

181   interface Positioned{
182     var canvas : int;
183     var render : int;

185     var computedX : int;
186     var computedY : int;

188     var posX : int;
189     var posY : int;
190     var posWidth : int;
```

```
191     var posHeight : int;

193     var computedWidth : int;
194     input width : taggedInt = {1,0};
195     var computedHeight : int;
196     input height : taggedInt = {1,0};

198     var intrinsPrefWidth : int;
199     var intrinsMinWidth : int;
200     var intrinsHeight : int;

202     var inhFontSize : int;
203     input intrinsFontSize : taggedInt = {2,0};

205     var inhColor : color;
206     input color : ?color = "none";

208     input left : taggedInt = {1,0};
209     input right : taggedInt = {1,0};
210     input top : taggedInt = {1,0};
211     input bottom : taggedInt = {1,0};
212     input position : string;
213     //box model
214     input marginTop : taggedInt = {302,0};
215     input marginBottom : taggedInt = {302,0};
216     input marginLeft : taggedInt = {302,0};
217     input marginRight : taggedInt = {302,0};
218     var mt : int;
219     var mb : int;
220     var ml : int;
221     var mr : int;
222     input paddingTop : taggedInt = {302,0};
223     input paddingBottom : taggedInt = {302,0};
224     input paddingLeft : taggedInt = {302,0};
225     input paddingRight : taggedInt = {302,0};
226     var pt : int;
227     var pb : int;
228     var pl : int;
229     var pr : int;

231   }

233   trait countChilds {
234     attributes {
235     var numChilds : int;
236     }
237     actions {
238     loop childs {
239       numChilds := fold 0 .. self$-.numChilds + 1;
240       childs.childNum := fold 0 .. childs$-.childNum + 1;
241     }
242     }
243   }
244   trait inlineWidthIntrinsics{
245     attributes{
246       var sumMarginsPadding : int;
247     }
248     actions{
249       sumMarginsPadding :=
250         (getTag(marginLeft) == CONST_AUTO() ?
```

```
251           0 : getValue(marginLeft,usedFontSize,0)) +
252         (getTag(marginRight) == CONST_AUTO() ?
253           0 : getValue(marginRight,usedFontSize,0)) +
254         (getTag(paddingLeft) == CONST_AUTO() ?
255           0 : getValue(paddingLeft,usedFontSize,0)) +
256         (getTag(paddingRight) == CONST_AUTO() ?
257           0 : getValue(paddingRight,usedFontSize,0));
258     }
259   }

261   trait strokeBox {
262     actions {
263     render :=
264       canvas + (validColor(bgc) ?
265         paintRect(absX + ml,absY + mt,
266           computedWidth + pl + pr, computedHeight + pt + pb,getColor(bgc)) : 0) +
267       (borders != "solid" ? 0 :
268         paintLine(absX + ml, absY + mt,
269           absX+ml+pr+pl+computedWidth ,absY + mt, borderw,getColor(borderc)) +
270         paintLine(absX + ml + pr + pl + computedWidth, absY + mt,
271           absX + ml + pl + pr + computedWidth, absY + mt + pt + pb + computedHeight,
272           borderw, getColor(borderc)) +
273         paintLine(absX + ml + pr + pl + computedWidth, absY + mt + pt + pb + computedHeight,
274           absX + ml, absY + mt + pt + pb + computedHeight, borderw, getColor(borderc)) +
275         paintLine(absX + ml, absY + mt + pt + pb + computedHeight, absX + ml,
276           absY + mt, borderw, getColor(borderc))
277       );
278     }
279   }


282   trait blockPosCont{
283     actions{
284       loop posChilds{
285         posChilds.posX := computedX;
286         posChilds.posY := computedY;
287         posChilds.posWidth := computedWidth;
288         posChilds.posHeight := computedHeight;

290         posChilds.inhFontSize := usedFontSize;
291         posChilds.inhColor := usedColor;
292         posChilds.canvas := fold render .. posChilds$-.canvas;
293       }
294     }
295   }

297   trait inlinePosCont{
298     actions{
299        loop posChilds{
300         posChilds.posX := minX + relX + offsetX;
301         posChilds.posY := minY + relY + offsetY;
302         posChilds.posWidth := intrinsPrefWidth;
303         posChilds.posHeight := intrinsHeight;

305         posChilds.inhFontSize := usedFontSize;
306         posChilds.inhColor := usedColor;
307         posChilds.canvas := fold render .. posChilds$-.canvas;
308       }
309     }
310   }
```

```
312 trait widthIntrinsics{
313   attributes{
314     var sumMarginsPadding : int;
315     var selfIntrinsWidth : int;
316   }
317   actions{
318     sumMarginsPadding :=
319       (getTag(marginLeft) == CONST_AUTO() ?
320         0 : getValue(marginLeft,usedFontSize,0)) +
321           (getTag(marginRight) == CONST_AUTO() ?
322             0 : getValue(marginRight,usedFontSize,0)) +
323           (getTag(paddingLeft) == CONST_AUTO() ?
324             0 : getValue(paddingLeft,usedFontSize,0)) +
325           (getTag(paddingRight) == CONST_AUTO() ?
326             0 : getValue(paddingRight,usedFontSize,0));

328     selfIntrinsWidth := (getTag(width) == CONST_AUTO() ?
329       0 : getValue(width,usedFontSize,0));
330   }

332 }

334 trait fontStyle{
335   attributes{
336     var usedFontSize : int;
337     var usedColor : color;
338   }
339   actions{
340     usedColor := validColor(color) ? getColor(color) : inhColor;
341     usedFontSize := validFontSize(intrinsFontSize) ?
342       getFontSize(intrinsFontSize, inhFontSize) : inhFontSize;

344     loop childs{
345       childs.inhColor := usedColor;
346       childs.inhFontSize := usedFontSize;
347     }
348   }
349 }

351 trait blockWidth{
352   attributes { var tmpComputedHeight : int; }
353   actions{
354     computedWidth := getTag(width) != CONST_AUTO() ?
355       getValue(width,usedFontSize,availableWidth) :
356         max(intrinsMinWidth,availableWidth) - ml - mr - pl - pr;

358     tmpComputedHeight := getTag(height) != CONST_AUTO() ?
359       getValue(height,usedFontSize,containHeight) : intrinsHeight;
360     computedHeight := isNaN(tmpComputedHeight) ? intrinsHeight : tmpComputedHeight;

362     pt := getValue(paddingTop,usedFontSize,availableWidth);
363     pb := getValue(paddingBottom,usedFontSize,availableWidth);
364     pl := getValue(paddingLeft,usedFontSize,availableWidth);
365     pr := getValue(paddingRight,usedFontSize,availableWidth);

367     mt := getTag(marginTop) != CONST_AUTO() ?
368       getValue(marginTop, usedFontSize, availableWidth) : 0;
369     mb := getTag(marginBottom) != CONST_AUTO() ?
370       getValue(marginBottom, usedFontSize, availableWidth) : 0;
```

```
372       ml := (getTag(marginLeft) != CONST_AUTO()) ?
373         getValue(marginLeft,usedFontSize,availableWidth) :
374         (getTag(width) == CONST_AUTO() ?
375           0 : (getTag(marginRight) == CONST_AUTO() ?
376             (availableWidth - pr - pl - getValue(width,usedFontSize, availableWidth))/2
377         : (availableWidth - pr - pl
378           - getValue(width,usedFontSize,availableWidth)
379           - getVal(marginRight,usedFontSize,availableWidth))));

381       mr := (getTag(marginRight) != CONST_AUTO()) &&
382         (getTag(width) == CONST_AUTO()  getTag(marginLeft) == CONST_AUTO()) ?
383           getValue(marginRight,usedFontSize,availableWidth) :
384         (getTag(width) == CONST_AUTO() ?
385           0 : (getTag(marginLeft) == CONST_AUTO() ?
386             (availableWidth - pr - pl
387               - getValue(width,usedFontSize, availableWidth))/2
388         : (availableWidth - pr - pl
389           - getValue(width,usedFontSize,availableWidth)
390           - getValue(marginLeft,usedFontSize,availableWidth))));
391     }
392   }



396   trait blockRelPos{
397     actions{
398       computedX := absX + (position == "relative" ?
399         (getTag(left) == CONST_AUTO() ?
400           (getTag(right) == CONST_AUTO() ? 0
401           : -getValue(right,usedFontSize,availableWidth))
402       : getValue(left,usedFontSize,availableWidth))  : 0);
403       computedY := absY + (position == "relative" ?
404         (getTag(top) == CONST_AUTO() ?
405           (getTag(bottom) == CONST_AUTO() ? 0 :
406             -getValue(bottom,usedFontSize,availableWidth))
407           : getValue(top,usedFontSize,availableWidth)) : 0);
408     }
409   }

411   trait inlineRelPos{
412     actions{
413       offsetX := inhOffsetX + (position == "relative" ?
414         (getTag(left) == CONST_AUTO() ?
415           (getTag(right) == CONST_AUTO() ? 0
416             : -getValue(right,usedFontSize,maxWidth))
417         : getValue(left,usedFontSize,maxWidth))  : 0);
418       offsetY := inhOffsetY + (position == "relative" ?
419         (getTag(top) == CONST_AUTO() ?
420           (getTag(bottom) == CONST_AUTO() ? 0
421             : -getValue(bottom,usedFontSize,maxWidth))
422         : getValue(top,usedFontSize,maxWidth))  : 0);

424       loop childs{
425         childs.inhOffsetX := offsetX;
426         childs.inhOffsetY := offsetY;
427       }
428     }
429   }
```

```
431 trait inlineblockWidth{
432   attributes{ var tmpComputedHeight : int; }
433   actions{
434     computedWidth := getTag(width) != CONST_AUTO() ?
435       getValue(width,usedFontSize,maxWidth)
436       : min(max(intrinsMinWidth,maxWidth),intrinsPrefWidth)
437         - ml - mr - pl - pr;
438     tmpComputedHeight := getTag(height != CONST_AUTO()) ?
439       getValue(height,usedFontSize,containHeight)  : intrinsHeight;
440     computedHeight := isNaN(tmpComputedHeight) ?
441       intrinsHeight : tmpComputedHeight;

443     pt := getValue(paddingTop,usedFontSize,maxWidth);
444     pb := getValue(paddingBottom,usedFontSize,maxWidth);
445     pl := getValue(paddingLeft,usedFontSize,maxWidth);
446     pr := getValue(paddingRight,usedFontSize,maxWidth);

448     mt := getTag(marginTop) != CONST_AUTO() ?
449       getValue(marginTop, usedFontSize, maxWidth) : 0;
450     mr := getTag(marginLeft) != CONST_AUTO() ?
451       getValue(marginLeft, usedFontSize, maxWidth) : 0;
452     ml := getTag(marginRight) != CONST_AUTO() ?
453       getValue(marginRight, usedFontSize, maxWidth) : 0;
454     mb := getTag(marginBottom) != CONST_AUTO() ?
455       getValue(marginBottom, usedFontSize, maxWidth) : 0;

457   }
458 }

460 trait inlineMargins{
461   actions{
462     pt := 0;
463     pb := 0;
464     pl := getValue(paddingLeft,usedFontSize,maxWidth);
465     pr := getValue(paddingRight,usedFontSize,maxWidth);

467     mt := 0;
468     mb := 0;
469     mr := getTag(marginLeft) != CONST_AUTO() ?
470       getValue(marginLeft, usedFontSize, maxWidth) : 0;
471     ml := getTag(marginRight) != CONST_AUTO() ?
472       getValue(marginRight, usedFontSize, maxWidth) : 0;
473   }
474 }

476 trait Stacking{
477   actions{
478     loop childs{
479       childs.absX := computedX + ml+pl;
480       childs.absY := fold computedY + mt+pt ..
481         childs$-.absY + (childs$i.childNum == 1 ?
482           0 : (childs$-.computedHeight + childs$-.pt
483             + childs$-.pb + childs$-.mt + childs$-.mb));
484       childs.availableWidth := computedWidth;
485       childs.containHeight := getTag(height) != CONST_AUTO() ?
486         getValue(height, intrinsFontSize, containHeight) : CONST_NAN();

488       intrinsMinWidth := fold selfIntrinsWidth + sumMarginsPadding ..
489         max(self$-.intrinsMinWidth,
490           sumMarginsPadding + childs$i.intrinsMinWidth);
```

```
491         intrinsPrefWidth := fold selfIntrinsWidth + sumMarginsPadding ..
492           selfIntrinsWidth == 0 ?
493             max($-.intrinsPrefWidth,
494               sumMarginsPadding + childs$i.intrinsPrefWidth)
495             : $-.intrinsPrefWidth;
496         intrinsHeight := fold 0 ..
497           $-.intrinsHeight + childs$i.computedHeight + childs$i.mt
498           + childs$i.mb + childs$i.pt + childs$i.pb;
499       }
500     }
501   }

503   trait WrappingLeaf{
504     actions{
505       relRightX := relX + computedWidth + pl + pr + ml + mr;
506       relRightY := relY;
507       firstChildWidth := computedWidth + rightPadding + pl + pr + ml + mr;
508       maxLineH := max(oldLineH, computedHeight + pt + pb);
509     }
510   }

512   trait Wrapping {
513     actions {
514     loop childs {
515       childs.minX := minX;
516       childs.minY := minY;
517       childs.maxWidth := maxWidth;
518       childs.containHeight := containHeight;

520       intrinsPrefWidth :=
521         fold ($$.numChilds == 0 ? 0 : -5) + sumMarginsPadding
522         ..
523         self$-.intrinsPrefWidth + childs$i.intrinsPrefWidth + 5;
524       intrinsMinWidth := fold sumMarginsPadding ..
525         max(self$-.intrinsMinWidth,
526           sumMarginsPadding + childs$i.intrinsMinWidth);
527       intrinsHeight := fold 0 ..
528         max(self$-.intrinsHeight,
529           childs$i.intrinsHeight + childs$i.mt + childs$i.mb
530             + childs$i.pt + childs$i.pb);


533       firstChildWidth := fold ml+pl ..
534         (childs$i.childNum == 1) ?
535           childs$i.firstChildWidth + ml+mr+pl+pr : $-.firstChildWidth;

537       childs.rightPadding := fold 0 ..
538         (childs$i.childNum == $$.numChilds) ?
539           rightPadding + mr+pr : 0;

541       childs.relX := fold 0 ..
542         ((childs$i.childNum == 1 ? relX + ml+pl : childs$-.relRightX + 5)
543         + childs$i.firstChildWidth > maxWidth) ?
544          0 : (childs$i.childNum == 1 ?
545            relX + ml+pl : childs$-.relRightX + 5);

547       childs.relY := fold 0 ..
548         (childs$i.childNum == 1 ? relY : childs$-.relRightY)
549         + (childs$i.relX == 0 ?
550             (childs$i.childNum == 1 ?
```

```
551               oldLineH : childs$-.maxLineH + 5) : 0);

553       childs.oldLineH := (childs$i.childNum == 1) ?
554         oldLineH : ((childs$i.relX == 0) ? 0 : childs$-.maxLineH);

556       relRightX := fold relX + intrinsPrefWidth .. childs$i.relRightX + mr+pr;
557       relRightY := fold relY .. childs$i.relRightY;
558       maxLineH := fold max(oldLineH, 0) .. childs$i.maxLineH;
559     }
560     }
561   }

563   interface Top{}
564   class Root : Top{
565     children{ child : Block; }
566     actions{
567       child.absX := child.computedHeight ? 0 : 0;
568       child.absY := child.computedHeight ? 0 : 0;
569       child.availableWidth := getPageWidth() - 15;
570       child.canvas :=
571         paintStart(child.computedWidth + child.pr + child.pl + child.mr + child.ml,
572           child.computedHeight + child.mt + child.mb + child.pt + child.pb);
573       child.childNum := 1;
574       child.inhColor := "black";
575       child.inhFontSize := 20;
576       child.containHeight :=  getPageHeight();
577     }
578   }

580   //Misc

582   class BlockImg(blockRelPos) : Block{
583     attributes {
584        input imagehandle : int;
585        var usedFontSize : int;
586        var usedColor : color;
587     }
588     actions{
589       usedColor := validColor(color) ? getColor(color) : inhColor;
590       usedFontSize := validFontSize(intrinsFontSize) ?
591         getFontSize(intrinsFontSize, inhFontSize) : inhFontSize;

593       render := canvas + paintImg(computedX,computedY,imagehandle);
594       intrinsHeight := getImageHeight(imagehandle);
595       intrinsMinWidth := getImageWidth(imagehandle);
596       intrinsPrefWidth := getImageWidth(imagehandle);
597       computedWidth := intrinsPrefWidth;
598       computedHeight := intrinsHeight;

600       pt := 0;
601       pb := 0;
602       pl := 0;
603       pr := 0;

605       mt := 0;
606       mb := 0;
607       mr := 0;
608       ml := 0;
609     }
610   }
```

```
612 class InlineImg(WrappingLeaf) : Inline{
613   attributes {
614      input imagehandle : int;
615      var usedFontSize : int;
616      var usedColor : color;
617      var computedHeight : int;
618      var computedWidth : int;
619   }
620   actions{
621     usedColor := validColor(color) ? getColor(color) : inhColor;
622     usedFontSize := validFontSize(intrinsFontSize) ?
623       getFontSize(intrinsFontSize, inhFontSize) : inhFontSize;

625     render := canvas
626       + paintImg(minX + relX + offsetX, minY + relY + offsetY,imagehandle);
627     intrinsHeight := getImageHeight(imagehandle);
628     intrinsMinWidth := getImageWidth(imagehandle);
629     intrinsPrefWidth := getImageWidth(imagehandle);
630     computedWidth := intrinsPrefWidth;
631     computedHeight := intrinsHeight;

633     offsetX := inhOffsetX + (position == "relative" ?
634       (getTag(left) == CONST_AUTO() ?
635         (getTag(right) == CONST_AUTO() ? 0
636           : -getValue(right,usedFontSize,maxWidth))
637         : getValue(left,usedFontSize,maxWidth))
638       : 0);
639     offsetY := inhOffsetY + (position == "relative" ?
640       (getTag(top) == CONST_AUTO() ?
641         (getTag(bottom) == CONST_AUTO() ?
642           0 : -getValue(bottom,usedFontSize,maxWidth))
643         : getValue(top,usedFontSize,maxWidth))
644       : 0);

646     pt := 0;
647     pb := 0;
648     pl := 0;
649     pr := 0;

651     mt := 0;
652     mb := 0;
653     mr := 0;
654     ml := 0;
655   }

657 }

659 //Blocks

661 class FlowBlock(blockWidth, strokeBox, widthIntrinsics, blockRelPos) : Block{
662   children{child : FlowRoot;}
663   attributes{
664     var usedFontSize : int;
665     var usedColor : color;
666   }
667   actions{
668     child.canvas := render;

670     child.relX := 0;
```

```
671       child.relY := 0;
672       child.oldLineH := 0;

674       child.rightPadding := 0;
675       child.minX := computedX + ml + pl;
676       child.minY := computedY + mt + pt;
677       child.maxWidth := computedWidth;
678       child.containHeight := getTag(height) != CONST_AUTO() ?
679         getValue(height,usedFontSize,containHeight) : CONST_NAN();

681       intrinsMinWidth := max(selfIntrinsWidth + sumMarginsPadding,
682         child.intrinsMinWidth + sumMarginsPadding);
683       intrinsPrefWidth := selfIntrinsWidth == 0 ?
684         child.intrinsPrefWidth + sumMarginsPadding :
685           selfIntrinsWidth + sumMarginsPadding;
686       intrinsHeight := child.relRightY + child.maxLineH - child.relY
687         + child.mt + child.mb + child.pt + child.pb;

689       usedColor := validColor(color) ? getColor(color) : inhColor;
690       usedFontSize := validFontSize(intrinsFontSize) ?
691         getFontSize(intrinsFontSize, inhFontSize) : inhFontSize;

693       child.inhColor := usedColor;
694       child.inhFontSize := usedFontSize;
695     }
696   }

698   class NormalBlock(Stacking, blockWidth, strokeBox, fontStyle, countChilds,
699       widthIntrinsics, blockRelPos, blockPosCont) : Block{
700     children{ childs : [Block]; posChilds : [Positioned]; }
701     actions{
702       loop childs{
703         childs.canvas := fold render .. childs$-.canvas;
704       }
705     }
706   }

708   //Inlines
709   class FlowRootC (Wrapping, inlineWidthIntrinsics, fontStyle,
710       inlineMargins, countChilds, inlinePosCont) : FlowRoot{
711     children{ childs : [Inline]; posChilds : [Positioned]; }
712     attributes{
713       var offsetX : int;
714       var offsetY : int;
715     }
716     actions{
717       render := canvas;
718       offsetX := 0;
719       offsetY := 0;
720       loop childs{
721         childs.canvas := render;
722         childs.inhOffsetX := 0;
723         childs.inhOffsetY := 0;
724       }
725     }
726   }

728   class NormalInline(Wrapping, fontStyle, inlineMargins, countChilds,
729       inlineWidthIntrinsics, inlineRelPos, inlinePosCont) : Inline{
730     children{ childs : [Inline]; posChilds : [Positioned]; }
```

```
731     actions{
732       render := canvas;
733       loop childs{
734         childs.canvas := fold render .. childs$-.canvas;
735       }
736     }
737   }

739   class InlineBlock(WrappingLeaf, Stacking, fontStyle, inlineblockWidth,
740       countChilds, widthIntrinsics, blockPosCont, strokeBox) : Inline{
741     children{ childs : [Block]; posChilds : [Positioned]; }
742     attributes{
743       var computedWidth : int;
744       var absX : int;
745       var absY : int;
746       var computedX : int;
747       var computedY : int;
748       var computedHeight : int;
749       input width : taggedInt = {1,0};
750       input height : taggedInt = {1,0};

752       input borderc : ?color;
753       input borderw : int;
754       input borders : string = "none";
755       input bgc : ?color;
756     }
757     actions{
758       absX := minX + relX + offsetX;
759       absY := minY + relY + offsetY;
760       computedX := absX;
761       computedY := absY;

763       offsetX := inhOffsetX + (position == "relative" ?
764         (getTag(left) == CONST_AUTO() ?
765           (getTag(right) == CONST_AUTO() ?
766             0 : -getValue(right,usedFontSize,maxWidth))
767           : getValue(left,usedFontSize,maxWidth))
768         : 0);
769       offsetY := inhOffsetY + (position == "relative" ?
770         (getTag(top) == CONST_AUTO() ?
771           (getTag(bottom) == CONST_AUTO() ?
772             0 : -getValue(bottom,usedFontSize,maxWidth))
773           : getValue(top,usedFontSize,maxWidth))
774         : 0);

776       loop childs{
777         childs.canvas := fold render .. childs$-.canvas;
778       }
779     }
780   }

782   class TextBox(inlineMargins) : Inline {
783     attributes {
784     input text : string;
785     input fontFamily : string = "Arial";
786     var lineHeight : int;
787     var lineSpacing : int;
788     var numberLines : int;
789     var overflow : bool;
790     var renderFontSize : int;
```

```
791     var usedFontSize : int;
792     var renderColor : color;

794     var splitText : int;
795     var metrics : int;

797     }
798     actions {
799     metrics := 0;

801     offsetX := inhOffsetX + (position == "relative" ?
802       (getTag(left) == CONST_AUTO() ?
803         (getTag(right) == CONST_AUTO() ?
804           0 : -getValue(right,usedFontSize,maxWidth))
805         : getValue(left,usedFontSize,maxWidth))
806         : 0);
807     offsetY := inhOffsetY + (position == "relative" ?
808       (getTag(top) == CONST_AUTO() ?
809         (getTag(bottom) == CONST_AUTO() ?
810           0 : -getValue(bottom,usedFontSize,maxWidth))
811         : getValue(top,usedFontSize,maxWidth))
812       : 0);


815     renderFontSize := validFontSize(intrinsFontSize) ?
816       getFontSize(intrinsFontSize,inhFontSize) : inhFontSize;
817     renderColor := validColor(color) ? getColor(color) : inhColor;
818     usedFontSize := renderFontSize;

820     overflow := false;
821     lineSpacing := 5;
822     render := canvas
823       + paintParagraph(splitText, fontFamily, renderFontSize,
824           minX + offsetX, minY + offsetY, relX, relY, maxWidth,
825           false, lineHeight, renderColor, lineSpacing);

827     splitText := splitText(relX,maxWidth,text,fontFamily,renderFontSize);

829     relRightX := (numberLines == 1) ?
830       (relX + intrinsPrefWidth) : getLeftoverWidth(splitText);
831     relRightY := (numberLines - 1) * (lineHeight + lineSpacing) + relY;
832     maxLineH := max(oldLineH, lineHeight);

834     lineHeight := getLineHeight(text, fontFamily, renderFontSize, metrics);
835     intrinsPrefWidth := getSumWordW(text, fontFamily, renderFontSize, metrics);
836     intrinsMinWidth := getMaxWordW(text, fontFamily, renderFontSize, metrics);
837     intrinsHeight := lineHeight;
838     firstChildWidth := getFirstWordW(text,fontFamily,renderFontSize);
839     numberLines := getNumberLines(splitText);
840     }
841   }
842   //Positioned elements
843   class PositionedBlock(Stacking, fontStyle, countChilds, widthIntrinsics) : Positioned {
844     children{ childs : [Block]; }
845     attributes{
846       var posWidthTmp : int;
847       var posHeightTmp : int;
848       var posRelX : int;
849       var posRelY : int;
850       var containHeight : int;
```

```
851     }
852     actions{
853       render := canvas;

855       containHeight := posHeight - mt - mb - pt - pb;
856       loop childs{
857         childs.canvas := fold render .. childs$-.canvas;
858       }
859       computedHeight := intrinsHeight;

861       pt := getValue(paddingTop, usedFontSize, posWidthTmp);
862       pb := getValue(paddingBottom, usedFontSize, posWidthTmp);
863       pl := getValue(paddingLeft, usedFontSize, posWidthTmp);
864       pr := getValue(paddingRight, usedFontSize, posWidthTmp);

866       ml := (getTag(marginLeft) != CONST_AUTO()) ?
867         getValue(marginLeft, usedFontSize, posWidthTmp) :(
868           (getTag(width) != CONST_AUTO() && getTag(left) != CONST_AUTO()
869             && getTag(right) != CONST_AUTO()) ?
870             (getTag(marginRight) == CONST_AUTO() ?
871           (posWidthTmp - pr - pl
872             - getValue(left, usedFontSize, posWidthTmp)
873             - getValue(right, usedFontSize, posWidthTmp)
874             - getValue(width, usedFontSize, posWidthTmp))/2 :
875           (posWidthTmp - pr - pl
876             - getValue(left, usedFontSize, posWidthTmp)
877             - getValue(right, usedFontSize, posWidthTmp)
878             - getValue((marginRight), usedFontSize, posWidthTmp)
879             - getValue(width, usedFontSize, posWidthTmp))) : 0);

881       mr := (getTag(marginRight) != CONST_AUTO()) ?
882         getValue(marginRight, usedFontSize, posWidthTmp) : (
883           (getTag(width) != CONST_AUTO() && getTag(left) != CONST_AUTO()
884             && getTag(right) != CONST_AUTO()) ?
885         (getTag(marginLeft) == CONST_AUTO() ?
886           (posWidthTmp - pr - pl
887             - getValue(left, usedFontSize, posWidthTmp)
888             - getValue(right, usedFontSize, posWidthTmp)
889             - getValue(width, usedFontSize, posWidthTmp))/2 :
890         (posWidthTmp - pr - pl
891           - getValue(left, usedFontSize, posWidthTmp)
892           - getValue(right, usedFontSize, posWidthTmp)
893           - getValue(marginLeft, usedFontSize, posWidthTmp)
894           - getValue(width, usedFontSize, posWidthTmp))) : 0);

896       mt := (getTag(marginTop) != CONST_AUTO()) ?
897         getValue(marginTop, usedFontSize, posWidth) :
898           ((getTag(height) == CONST_AUTO()  getTag(top) == CONST_AUTO()
899                getTag(bottom) == CONST_AUTO()) ? 0 :
900             ((getTag(marginBottom) == CONST_AUTO()) ?
901             (posWidthTmp - pr - pl
902               - getValue(top, usedFontSize, posWidthTmp)
903               - getValue(bottom, usedFontSize, posWidthTmp)
904               - getValue(width, usedFontSize, posWidthTmp))/2 :
905             (posWidthTmp - pr - pl
906               - getValue(top, usedFontSize, posWidthTmp)
907               - getValue(bottom, usedFontSize, posWidthTmp)
908               - getValue(width, usedFontSize, posWidthTmp)
909               - getValue(marginBottom, usedFontSize, posWidthTmp))));
```

```
911       mb := (getTag(marginTop) != CONST_AUTO()) ?
912         getValue(marginBottom, usedFontSize, posWidth) :
913           ((getTag(height) == CONST_AUTO()  getTag(top) == CONST_AUTO()
914             getTag(bottom) == CONST_AUTO()) ? 0 :
915           ((getTag(marginTop) == CONST_AUTO()) ?
916           (posWidthTmp - pr - pl
917             - getValue(top, usedFontSize, posWidthTmp)
918             - getValue(bottom, usedFontSize, posWidthTmp)
919             - getValue(width, usedFontSize, posWidthTmp))/2 :
920           (posWidthTmp - pr - pl
921             - getValue(top, usedFontSize, posWidthTmp)
922             - getValue(bottom, usedFontSize, posWidthTmp)
923             - getValue(width, usedFontSize, posWidthTmp)
924             - getValue(marginTop, usedFontSize, posWidthTmp))));


927       posWidthTmp := position == "absolute" ? posWidth : getPageWidth() - 15;
928       posHeightTmp := position == "absolute" ? posHeight : getPageHeight();

930       computedWidth := getTag(width) != CONST_AUTO() ?
931         getValue(width, usedFontSize, posWidthTmp) : (
932         (getTag(left) == CONST_AUTO()  getTag(right) == CONST_AUTO()
933           getTag(marginLeft) == CONST_AUTO()
934           getTag(marginRight) == CONST_AUTO()) ?
935         min(max(intrinsMinWidth, posWidthTmp), intrinsPrefWidth)
936         : (posWidthTmp
937             - getValue(left, usedFontSize, posWidthTmp)
938             - getValue(right, usedFontSize, posWidthTmp)
939             - ml - mr - pl - pr));

941       computedHeight := getTag(height) != CONST_AUTO() ?
942         getValue(height, usedFontSize, posHeightTmp) : (
943           (getTag(top) == CONST_AUTO()  getTag(bottom) == CONST_AUTO()
944             getTag(marginTop) == CONST_AUTO()
945             getTag(marginBottom) == CONST_AUTO()) ?
946           intrinsHeight :
947           (posHeightTmp
948             - getValue(left, usedFontSize, posHeightTmp)
949             - getValue(right, usedFontSize, posHeightTmp)
950             - ml - mr - pl - pr));

952       posRelX := (getTag(left) != CONST_AUTO() ) ?
953         getValue(left, usedFontSize, posWidthTmp) :
954           (getTag(right) == CONST_AUTO() ?
955             0 :
956             (posWidthTmp - pl - pr - ml - mr
957               - getValue(right, usedFontSize, posWidthTmp)
958               - computedWidth));
959       posRelY := (getTag(top) != CONST_AUTO() ) ?
960         getValue(top, usedFontSize, posWidthTmp) :
961           (getTag(bottom) == CONST_AUTO() ?
962             0 :
963             (posWidthTmp - pl - pr - ml - mr
964               - getValue(bottom, usedFontSize, posWidthTmp)
965               - computedWidth));

967       computedX := (position == "absolute" ? posX : 0) + posRelX;
968       computedY := (position == "absolute" ? posY : 0) + posRelY;
969     }
970   }
```