## 2.2 The HBox Language as a Classical Attribute Grammar

### Example tree with dynamic dependencies

### Example static grammar instance

### Dynamic evaluator

## 2.3 Desugaring Loops and Other Modern Constructs

### Motivation: Productive Features with Simple Implementations

### Interfaces: Lightweight and Reusable Input/Output Specifications

### Traits: Reusing Cross-cutting Code

### Foreign Functions: Embedded Domain Specific Language

### Loops

## 2.4 Automatically Staged SIMD Memory Allocation for Rendering

### Problem

Dynamic memory allocation provides significant flexibility for a language, but it is unclear how to perform it on a GPU without significant performance penalties. This needed ended up leading to both performance and programmability issues in our design of a tessellation library that connects our GPU layout engine to our GPU rendering engine. Our insight is that the memory allocation may be staged using a variant of prefix sum node labeling. One pass gathers memory requests, a bulk allocation for the total amount is made, and then a scatter pass provides each node with a contiguous memory segment of it. We found manipulating memory addresses in this way to be error-prone, so we show how to use our synthesizer to automatically schedule use of the parallel memory allocator. Furthermore, we show how to syntactically hide the use of our allocation scheme through a macro that automatically expands into staged dynamic memory allocation and consumption calls.

For example, we found parallel dynamic memory allocation to simplify the transition between layout and rendering. All nodes that render a circle will call some form of drawCircle in Figure 6.6a. Depending on the size of the circle, which is computed as part of the layout traversals, a different amount of memory will be allocated. Once the memory is allocated,

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

(a) **Naıve drawing primitive .**

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

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

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

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

Figure 2.1: **Partitioning of a library function that uses dynamic memory allocation into parallelizable stages.**

vertices will be filled in with the correct position. The rendering engine will then connect the vertices with lines and paint them to the screen. The processing of converting the abstract shape into renderable vertices is known as tessellation. We want our system to tesselate the display objects for each node in parallel.

## Staged Parallel Memory Allocation

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

1. Parallel request (bottom-up tree traversal to gather )
2. Physical memory allocation
3. Parallel response (top-down tree traversal to scatter)
4. Computations that consume dynamic memory (normal parallel tree traversals)

The staging allows us to parallelize the request and response stages. We reuse the parallel tree traversals for them, as well as for the actual consumption. The actual allocation of physical memory in stage 2 is fast because it is a single call. Figure 6.8 shows the dynamic data dependencies and two parallel tree traversals for an instance of staged parallel memory allocation.

```
CBOX → BOX1 BOX2
{
        ...
        CBOX.render =
                drawCircle(CBOX.x, CBOX.y, CBOX.radius)
                + drawCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
}
```

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

```
CBOX → BOX1 BOX2
{
        ...
        CBOX.sizeSelf =
                allocCircle(CBOX.x, CBOX.y, CBOX.radius)
                + allocCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
        CBOX.size = CBOX.sizeSelf +BOX1.size + BOX2.size;
        BOX1.buffer = CBOX.buffer + CBOX.sizeSelf;
        BOX2.buffer = BOX1.buffer + BOX1.size;
        CBOX.render =
                fillCircle(CBOX.x, CBOX.y, CBOX.radius, CBOX.buffer)
                + fillCircle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5,
                        CBOX.buffer + allocCircle(CBOX.x, CBOX.y, CBOX.radius));
}
```

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

```
CBOX → BOX1 BOX2
{
        ...
        CBOX.render =
                @Circle(CBOX.x, CBOX.y, CBOX.radius)
                + @Circle(CBOX.x + 10, CBOX.y + 10, CBOX.radius * 0.5);
}
```

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

Figure 2.2: **Use of dynamic memory allocation in a grammar for rendering two circles.**

Library functions that requires dynamic memory allocation are manually rewritten into allocation request (Figure 6.6b) and memory consumption fragments (Figure **??**). The transformation was not onerous to perform on our library primitives and, in the future, might be automated.

Invocations of the original in the attribute grammar are rewritten to use the new primitives. For example, drawing two circles (Figure 6.7a) is split into calls for allocation requests, buffer pointer manipulation, and buffer usage (Figure 6.7b). The transformation increases memory consumption costs due to book keeping of allocation sizes.

The result of our staging is three logical parallel passes, which, in practice, is merged into two parallel passes over the tree. 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 needs is computed,

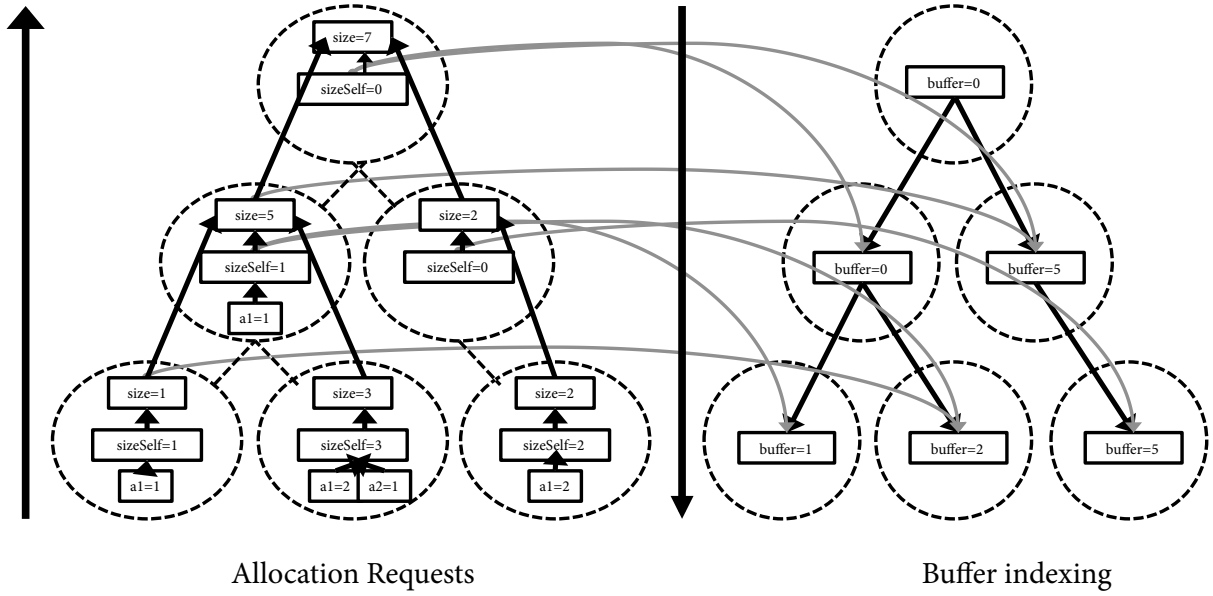


Figure 2.3: **Staged parallel memory allocation as two tree traversals.** First pass is parallel bottom-up traversal computing the sum of allocation requests and the second pass is a parallel top-down traversal computing buffer indices. Lines with arrows indicate dynamic data dependencies.

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 can be used for actual computations through normal parallel passes. Memory use can occur immediately upon computation of the buffer index, so the last two logical stages are merged in implementation.

## Automation with Automatic Scheduling and Macros

Manually manipulating the allocation requests and buffer pointers is error prone. We eliminated the problem 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 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 abstract 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 it is similar to code that assumes dynamic memory allocation primitives: the slight syntactic difference can be seen between Figure 6.7c and Figure 6.7a. Our macros (implemented in OMetaJS [[CITE]]) automatically expand into the form seen in Figure 6.7b.

Our use case only required one allocation stage, but multiple 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.

## 2.5 Evaluation: Mechanized Layout Features

### Staged SIMD Memory Allocation

We evaluate 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 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 data sets.

#### Flexibility

Our staged structuring and automation approach cannot express all dynamic memory usage patterns, so it is important to validate that it works on common patterns that occur in visualization. We found the three following patterns to be important:

- **Functional graphics.** Functional graphics primitives used in languages such as Scheme, O'Caml, and Haskell follow the form that we used for Circle. For example, many of our visualizations use simple variants such as 2D rectangles, 3D line, and arcs.
- **Linked view**. Multiple renderable objects can be associated with one node, which we can use for providing different views of the same data. Such functionality is common for statistical analysis software:

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

- **Zooming.** We can use the same multiple representation capability for a live zoomed out view ("picture-in-picture"):

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

- **Visibility toggles.** Our macros 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) ;
```

### 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 suggestive productivity measure is of how many lines of code the macro abstraction eliminates from our visualizations. We measured the impact on using it for 3 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 [[CITE]] and multiple 3D line graphs [[CITE]].

Table 2.1: 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 6.3 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 code that a human wrote.

As shown in Figure 6.7, the eliminated code is code that was introduced by staging the library calls. Porting unstaged functional graphics calls to the library, is in practice, an alpha renaming of function names. Using 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), so the productivity benefit is non-trivial.

#### Performance

### Rendering: Immediate Mode and Beyond

### Non-euclidean: Sunburst Diagram

### Charts: Line graphs

### Animation and Interaction: Treemap

### Flow-based: CSS Box Model

### Grid-based: HTML Tables

## 2.6 Related Work

- loose formalisms: browser impl (C++), d3 (JavaScript), latex formulas (ML)
- restricted formalisms: cassowary and hp, UREs
- AGs: html tables