A Kernel Semantics of Cascading Style Sheets

Leo Meyerovich,[*] Ras Bodik
University of California, Berkeley
{lmeyerov, siuman, siuon, bodik}@cs.berkeley.edu

ABSTRACT

We define the semantics of layout for a kernel of the CSS language ('ECSS') with attribute grammars. We investigate four aspects:

- CSS 2.1 specification semantics
- Idealized fixedpoint semantics
- Efficient sequential semantics and implementation
- Minimal-dependency semantics (to simplify decomposition for parallel evaluation strategies)

1. INTRODUCTION

Cascading style sheets, the language used to style web pages, is one of the most commonly used languages. Unfortunately, it is informally defined, which has led to significantly varying semantics among all major engine providers. In addition, while recent efforts have emphasized improving the performance of scripts running on pages, processing layouts actually dominates computation time, and interaction with the layout system is even a dominant bottleneck for typical scripts. We are interested in optimizing this layer, but, to do so, need a better understanding of its inherent structure. In particular, we are interested in isolating dependencies in order to explore parallel evaluation strategies. Thus, to facilitate both correct and efficient implementations, we formalize a functional semantics of an essential kernel of CSS (ecss) as a pipeline of tree and attribute grammars.

2. BACKGROUND

discuss div, span, and fictious blurb and quote and translate into canonical elements block, inlineline, floatleft, and inlineblock

A webpage to be displayed is computed from two data structures: a *content tree* of information to be displayed and style information for individual pieces of content, and a style template that contains information for styling multiple pieces of information. For example, a paragraph would be an intermediate node in the content tree, with descendents of sentences, words, or images – some of which may have extra padding or have some sort of extra emphasis. The style template might specify that any image nested within at least two tables will have extra padding. We are interested in the core functionality of the cascading style sheet system (CSS) most commonly used for styling web pages in this way.

Initial processing of layout occurs in two basic phases. First, the style sheet template is applied to the content tree. Next, the size and position of every node in the tree is computed. We elide over pre-processing stages like parsing and post-processing stages like actual rendering (known as painting) as they are significantly better defined and understood as well as relatively insignificant, computationally.

3. TEMPLATING

3.1 Templating Input

A style sheet template contains tuples of a path predicate and a set of properties, referred to as a rule. Every node in the content tree may match several rules, where a rule matches if its corresponding predicate is satisfied by the node. Every rule may specify several properties, like an element's color or padding, and may be in conflict with other rules: once rules have been matched to a node, they must be composed to resolve the final set of properties.

3.2 Rule Matching

TODO formal definition + popular kernel

3.3 Property Resolution

TODO

4. LAYOUT

Content nodes (eg., HTML elements) are classified as one of a few basic display types with some parameters. We focus on a subset of these that sufficiently exercises the layout process. Specified attributes are referred to as *computed* values as they are determined by earlier preprocessing stages. They may need to be solved, such as when they are percentages: the solved value is referred to as *used* values, as they can be almost directly plugged into a rendering engine.

4.1 Skeleton

The tree grammar in figure 4.1 represents the typed node structure of our simplified input, eliding input attributes

[*]This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

WWW2009, April 20-24, 2009, Madrid, Spain.
.

$$\begin{aligned}CHILD_{inline} &::= INLINE_{inline} \mid INLINE_{block} \mid FLOAT_{left}\\ CHILD_{block} &::= BLOCK \mid FLOAT_{left}\\ CHILD &::= CHILD_{inline} \mid CHILD_{block}\end{aligned}$$

$$\begin{aligned}BLOCK &::= {}^{\triangle}(CHILD^*)\\ INLINE_{inline} &::= {}^{\triangle}(CHILD^*)\\ INLINE_{block} &::= {}^{\triangle}(CHILD^*) \mid \mathrm{C}^+\\ FLOAT_{left} &::= {}^{\triangle}(CHILD^*)\end{aligned}$$

Figure 1: Input structure, modulo computed style attributes.

(). ${}^{\triangle}(\ldots)$ represents a new node with children, and * and $|$ are the usual Kleene star and disjunction symbols. C^+ represents a text string without spaces.

4.2 Input Attributes

$$\begin{aligned}BLOCK &::=_{attrib} [\mathrm{w} = DIM_a,\ \mathrm{h} = DIM_a]\\ INLINE_{block} &::=_{attrib} [\mathrm{w} = DIM_a,\ \mathrm{h} = DIM_a]\\ FLOAT_{left} &::=_{attrib} [\mathrm{w} = DIM_a,\ \mathrm{h} = DIM_a]\\ \mathrm{C}^+ &::=_{attrib} [\mathrm{w} = DIM,\ \mathrm{h} = DIM]\\ DIM_a &::=_{attrib} DIM \mid \widehat{\mathrm{auto}}\\ DIM &::=_{attrib} \widehat{\mathbb{PCNT}\ \%} \mid \widehat{\mathbb{R}\ \mathrm{px}}\\ &\quad \text{where } \mathbb{PCNT} = [0,1] \subset \mathbb{R}\end{aligned}$$

Figure 2: Computed attributes annotating the input tree to be sized and positioned.

Some nodes have input attributes that are explicitly or implicitly set in an upstream rule matching and property resolution phase. $INLINE$ elements are excluded. Set attributes, also known as *computed* values, are denoted with a $\widehat{\mathrm{hat}}$ (figure 4.2).

TODO: better way to formalize structure of attributes

In full CSS, there are other attributes commonly impacting layout, such as margin, border, and padding sizes, but they have little impact on functional dependencies and are essentially a constant multiplier on space and time.

4.3 Essential Layout Normal Form

$$\begin{aligned}&INLINE_{inline} :\\ &\qquad \mid {}^{\triangle}(x = (CHILD^*_{inline}))\\ &\qquad\quad \rightarrow\ {}^{\triangle}(x)\\ &\qquad \mid {}^{\triangle}((x = CHILD_{block} \mid y = CHILD_{inline})^*)\\ &\qquad\quad \rightarrow BLOCK[w = \widehat{\mathrm{auto}}, h = \widehat{\mathrm{auto}}]((\\ &\qquad\qquad\quad x \mid BLOCK[w = \widehat{\mathrm{auto}}, h = \widehat{\mathrm{auto}}](y))^*)\end{aligned}$$

Figure 3: Normalization of $INLINE_{inline}$ nodes.

To simplify layout, a normalization pass lifts $BLOCK$ nodes above $INLINE_{inline}$ ones (figure 4.3, creating new anonymous $BLOCK$s. It repeatedly applies the following bottom-up tree rewrite rules to fixpoint, replacing pattern matches of the left hand side with instantiated versions of the right. Note that the placement of the xs and ys is order-preserving.

It is unclear how much time this transformation takes in practice, though it likely takes little: due to its bottom-up nature, it is bounded by the size of the tree. If it does become a bottleneck, a task parallel strategy based on an upwards moving wave starting at the fringe would be natural. For example, given 5 cores, 5 large subtrees could be isolated as long as the input is not 4 long paths (which is unlikely).

The other part of the normalization step guarentees that children of block elements are homogenous. The patterns of the following rules short-circuit, and are again applied bottom-up to fixpoint:

$$\begin{aligned}&INLINE_{block},\ BLOCK,\ FLOAT_{left} :\\ &\qquad \mid {}^{\triangle}(x = CHILD^*_{block} \mid x =\ CHILD^*_{inline})\\ &\qquad\quad \rightarrow\ {}^{\triangle}(x)\\ &\qquad \mid {}^{\triangle}((x\ = CHILD_{block} \mid y\ =\ CHILD_{inline})^*)\\ &\qquad\quad \rightarrow\ {}^{\triangle}((x \mid BLOCK[w = \widehat{\mathrm{auto}}, h = \widehat{\mathrm{auto}}](y))^*)\end{aligned}$$

Figure 4: Normalization rules for blocks.

All of the rules are the same, modulo the type of the parent element, and the first half of every rule has no effect, only existing to guard the second case. Again, we assume the right-hand side preserves the order of the left in terms of instances of x and y.

The normal form, again eliding attributes, is shown in figure 4.3.

$$\begin{aligned}CHILD_{inline} &::= INLINE_{inline} \mid INLINE_{block}\\ &\quad \mid FLOAT_{left}\\ CHILD_{block} &::= BLOCK \mid FLOAT_{left}\\ CHILDREN_{homog} &::= CHILD^*_{block} \mid CHILD^*_{inline}\end{aligned}$$

$$\begin{aligned}BLOCK &::= {}^{\triangle}(CHILDREN_{homog})\\ INLINE_{inline} &::= {}^{\triangle}(CHILD^*_{inline})\\ INLINE_{block} &::= {}^{\triangle}(CHILDREN_{homog}) \mid \mathrm{C}^*\\ FLOAT_{left} &::= {}^{\triangle}(CHILDREN_{homog})\end{aligned}$$

Figure 5: Tree skeleton of normal form.

4.4 Attribute Grammars: Widths

We define layout solving in terms of an attribute grammar. A (x, y) coordinate pair relative to the nearest enclosing block or float and an absolute (w, h) dimension pair is assigned to every $BLOCK$, $FLOAT_{left}$, and $INLINE_{block}$ node. We first discuss solving a system without $FLOAT_{left}$

elements and simplied $INLINE_{block}$ elements, exploring potential evaluation strategies. After, we consider the dependencies induced by stronger $INLINE_{block}$ and $FLOAT_{left}$ support, ultimately suggesting how to optimistically account for them.

4.4.1 Simple Widths

$BLOCK$ elements stack vertically, while $INLINE_{inline}$ and $INLINE_{block}$ elements stack horizontally until a vertical boundary is reached, at which point they continue at the next line. Width sizing, at this point, is similar to that of typical *grid* or *box* style user interface systems: widths can be computed top down. Heights are computed bottom up, so boxes may contain all of the elements within them, but we consider this later.

Computed widths – those with $\widehat{\text{hats}}$ – are passed downwards to calculate *used* widths. An attribute $\overrightarrow{\text{computed by}}$ using such *inherited* attributes is depicted by a right arrow, hinting at the direction of data flow. Intuitively, an inherited attribute may only depend on $\widehat{\text{computed}}$ data or other $\overrightarrow{\text{inherited terms}}$. In figure 4.4.1, we describe layouts without $FLOAT_{left}$ elements nor $INLINE_{block}[w = \widehat{\text{auto}}]$ elements.

An ambiguity in the above definition is the used parent width of the root element: it is defined to be the width to the viewport. For clarity, we skip special casing it, assuming there is a parent viewport element.

Calculating widths in a tree thus constructed is simple: as information only flows downwards, once propagation has reached a node, recursion into its children can be done independently.

4.4.2 Shrink-to-Fit Inline Blocks

$INLINE_{block}$ elements with a computed width attribute of $\widehat{\text{auto}}$ use a shrink-to-fit policy to compute their used width. Three layout strategies are computed and compared based on the following: the minimum width, the preferred width, and the available width. Informally:

- **Minimum width:** Minimal width resulting from attempting all possible combinations of line breaks in descendents.
- **Preferred width:** Width resulting from only explicit line breaks in descendents.
- **Available width:** Width available from the parent.

The final *used* width is:

$$\min(\max(\mathit{minimum}, \mathit{available}), \mathit{preferred}) \tag{1}$$

With respect to widths, the CSS specifications are (intentionally) not well-defined for the current grammar and set of attributes. Consider the following sequence:

$$\begin{aligned}&{}_{a}BLOCK[w = \widehat{200\text{px}}](\\ &\quad {}_{b}INLINE_{block}[w = \widehat{\text{auto}}](\\ &\quad\quad {}_{c}BLOCK[w = \widehat{50\%}]))\end{aligned}$$

Element b is dependent upon the preferred size of c, but c is dependent upon the end size of b. We take the arbitrary (but hopefully intuitive and thus acceptable) strategy of breaking such dependencies by passing the closest block ancestor's used width (defaulting to the viewport if there is none).

Our original semantics attempted to solve for widths in two passes: once downwards, passing inherited widths (iw in the figure), and then once (topologically) upwards, resolving the final used widths (uw). However, we struggled to support the following example:

$$\begin{aligned}&{}_{a}INLINE_{block}[w = \widehat{\text{auto}}](\\ &\quad {}_{b}BLOCK[w = \widehat{\text{auto}}](\\ &\quad\quad {}_{c}BLOCK[w = \widehat{\text{auto}}](\\ &\quad\quad\quad {}_{d}BLOCK[w = \widehat{\text{auto}}](\ldots))))\end{aligned}$$

We first consider the basic constraints for this diagram. The width of a will shrink to fit its contents while trying to not blow outside of its parent. Elements b (and, transitively, c and d) will expand to fill the space provided by a. The children of d provide minimum and preferred width suggestions for d; b and c typically mimic these suggestions. This dictates the final size of a.

Operationally, we see multiple flows. Every $BLOCK$ width set to *auto* expands to match its parent: this represents a downwards pass to compute final widths. However, as demonstrated by d's children dictating the size of a, minimum and preferred widths must flow upwards before such $BLOCK$s are expanded, representing an earlier upwards pass. To compute some minimum and preferred widths of an element, if the element's width is a percentage, we must also know the width of the parent. This introduces a potential fixpoint computation: the percentage width is undefined in the CSS specification; to cleanly decomposes width computation into passes, we perform a pre-pass that computates percentages based on the closest non-shrink-to-fit width, and then a post-pass that uses the actual percentage. This appears to be similar to what Firefox and WebKit already perform.

Thus, we use two grammars. The first calculate initial percentages in a downwards pass by propagating non-shrink-to-fit widths, and then computes minimum and preferred widths in an upwards pass. In the second grammar, final widths are calculated moving downwards, recomputing percentages based on their final parent widths. These two grammars may be compacted into one because elements with percentage widths whose parent is shrink-to-fit are undefined and preferred and minimum widths are undefined, but the resultant grammar would likely not conform to intuitive behavior nor existing implementations.

Semantically, the width calculation can be represented with two grammars, but, for performance reasons, a lot of work can be lazily avoided or at least shifted into the first grammar. Most widths can be computed as soon as the preferred and minimum widths are known; the second grammar could likely only recompute over a subtree of the input. Furthermore, even though the current grammar computers minimum and preferred widths on all elements, this information is only necessary for some elements involved in shrink-to-fit computations, and thus might be lazily computed.

4.4.3 Floats

Floating elements provide a lot of flexibility to CSS, though

$$\begin{aligned}
CHILD_{inline} &\rightarrow INLINE_{inline}\\
CHILD_{block} &\rightarrow BLOCK\\
CHILDREN_{homog} &\rightarrow CHILD^{*}_{block} \mid CHILD^{*}_{inline}\\
(n = BLOCK[w = \widehat{i}])\{\overrightarrow{n.uw} = \overrightarrow{w_{bw}}(\overrightarrow{n.parent.uw}, \widehat{i})\} &\rightarrow {}^{\triangle}(CHILDREN_{homog})\\
(n = INLINE_{inline}[\,])\{\overrightarrow{n.uw} = \overrightarrow{n.parent.uw}\} &\rightarrow {}^{\triangle}(CHILD^{*}_{inline})\\
(n = INLINE_{block}[w = \widehat{i}])\{\overrightarrow{n.uw} = \overrightarrow{w_{ibw}}(\overrightarrow{n.parent.uw}, \widehat{i}, \widehat{s})\} &\rightarrow {}^{\triangle}(CHILDREN_{homog}) \mid \widehat{s} = \mathrm{C}^{+}
\end{aligned}$$

$$\begin{aligned}
&\overrightarrow{w_{bw}}(\cdot,\ \cdot):\\
&\quad \mid (\overrightarrow{w_{parent}},\ \widehat{n\,\%}) =\ w_{parent} * n\\
&\quad \mid (\overrightarrow{w_{parent}},\ \widehat{n\,\mathrm{px}}) =\ n\\
&\quad \mid (\overrightarrow{w_{parent}},\ \widehat{\mathrm{auto}}) =\ w_{parent}\\
&\overrightarrow{w_{ibw}}(\cdot,\ \cdot,\ \cdot):\\
&\quad \mid (\overrightarrow{w_{parent}},\ \widehat{n\,\%},\ \widehat{str}) =\ w_{parent} * n\\
&\quad \mid (\overrightarrow{w_{parent}},\ \widehat{n\,\mathrm{px}},\ \widehat{str}) =\ n\\
&\quad \mid (\overrightarrow{w_{parent}},\ \widehat{\mathrm{auto}},\ \widehat{str}) =\ str.length
\end{aligned}$$

Figure 6: Basic width calculation based on inherited attributes, ignoring $FLOAT_{left}$ and $INLINE_{block}[w = \widehat{auto}]$ elements.

[some examples]

Figure 9: Examples of some floats..

at the expense of specification and implementation complexity. They horizontally stack like $INLINE$ elements and similarly must not overlap with each other. Due to the normal form, they either occur interleaved with all $BLOCK$ or all $INLINE$ elements. Blocks interleaved with them still stack vertically, edge to edge: the $FLOAT$ elements push content within the $BLOCK$s. When interleaved with $INLINE$ elements, they are also positioned in lockstep with them, except pushed to the left or the right of the current line, and may span multiple lines. When pushed to the left, $INLINE$ and other $FLOAT$ elements after one cannot be both to its left and before the bottom of it. An analogous property holds for elements floating to the right, and figure 4.4.3 illustrates some interesting cases.

We do not model $FLOAT_{right}$ elements as they are analogous to $FLOAT_{left}$ ones. As with $INLINE_{block}$s, an $\widehat{\mathrm{auto}}$ width attribute corresponds to a shrink-to-fit policy. However, the heights of elements impacts the positioning of floats, which, coupled with nesting, makes exact computation of preferred widths and minimum widths height-dependent. Interestingly, as the CSS specification left the definition of the calculation of these properties undefined, we can again break height-dependencies in a reasonable manner while conforming to existing specification implementations (namely the popular Webkit engine used in Apple's Safari and Google's Chrome browsers)[1]. No browser that we are aware of computes the exact (undefined but intuitive) preferred and minimum width calculation, so we make a dependency-breaking choice that seems reasonable.

Consider the following structure:

$$\begin{aligned}
&{}_{a}INLINE_{block}[w = \widehat{\mathrm{auto}}](\\
&\quad {}_{b}FLOAT_{left}[w = \widehat{200\mathrm{px}};\ h = \widehat{100\mathrm{px}}]()\\
&\quad {}_{c}BLOCK[w = \widehat{\mathrm{auto}}](\\
&\quad\quad {}_{d}INLINE_{inline}[\,](\ldots))\\
&\quad {}_{e}BLOCK[w = \widehat{\mathrm{auto}}](\\
&\quad\quad {}_{f}INLINE_{inline}[\,](\ldots)))
\end{aligned}$$

Element b should occur on the same visual (horizontal) line as d, so the width of a should include both. However, it is unclear whether b will extend vertically after element c, and thus add the restriction that a's width should include the sum of b's width and f's. Thus, there is an induced dependency on heights, suggesting a fixpoint computation. No (known) browser actually performs this. An alternative permitted by the CSS specification closer to what seems to be performed by the Webkit engine is to consider $FLOAT$ elements indepndently of others, thus assuming that they do not overlap with non-$FLOAT$ siblings. This breaks the height dependency.

There is one more subtle case we encountered. Consider the following:

[1]Firefox/Gecko performs a similar calculation, but we believe it is because items following floats are typically placed on the next line, not because of width calculations.

$$
\begin{aligned}
&CHILD_{inline} \rightarrow INLINE_{inline} \mid INLINE_{block}\\
&CHILD_{block} \rightarrow BLOCK\\
&(n = BLOCK[w = \widehat{cw}])\ \{\overrightarrow{n.iw} = \overrightarrow{iw_{b,ib}}(\widehat{cw},\ \overrightarrow{n.parent.iw});\ \overrightarrow{n.cw} = \overrightarrow{cw_b}(\overrightarrow{n.parent.iw},\ \overrightarrow{n.parent.cw},\ \widehat{cw})\}\\
&\quad \mid \rightarrow\ ^{\triangle}(x = CHILD^*_{block})\ \{\overleftarrow{n.m} = (\max_{x_i \in x}(\overleftarrow{x_i.m})) \bowtie \overrightarrow{n.cw};\ \overleftarrow{n.p} = (\max_{x_i \in x}(\overleftarrow{x_i.p})) \bowtie \overrightarrow{n.cw}\}\\
&\quad \mid \rightarrow\ ^{\triangle}(x = CHILD^*_{inline})\ \{\overleftarrow{n.m} = (\max_{x_i \in x}(\overleftarrow{x_i.m})) \bowtie \overrightarrow{n.cw};\ \overleftarrow{n.p} = \sum_{x_i \in x}(\overleftarrow{x_i.p}) \bowtie \overrightarrow{n.cw}\}\\
&(n = INLINE_{block}[w = \widehat{cw}])\ \{\overrightarrow{n.iw} = \overrightarrow{iw_{b,ib}}(\widehat{cw},\ \overrightarrow{n.parent.iw});\ \overrightarrow{n.cw} = \overrightarrow{cw_{ib}}(\overrightarrow{n.parent.iw},\ \widehat{cw})\}\\
&\quad \mid \rightarrow\ ^{\triangle}(x = CHILD^*_{block})\ \{\overleftarrow{n.m} = (\max_{x_i \in x}(\overleftarrow{x_i.m})) \bowtie \overrightarrow{n.cw};\ \overleftarrow{n.p} = (\max_{x_i \in x}(\overleftarrow{x_i.p})) \bowtie \overrightarrow{n.cw}\}\\
&\quad \mid \rightarrow\ ^{\triangle}(x = CHILD^*_{inline})\ \{\overleftarrow{n.m} = (\max_{x_i \in x}(\overleftarrow{x_i.m})) \bowtie \overrightarrow{n.cw};\ \overleftarrow{n.p} = \sum_{x_i \in x}(\overleftarrow{x_i.p}) \bowtie \overrightarrow{n.cw}\}\\
&(n = INLINE_{inline})[\,]\{\overrightarrow{n.iw} = \overrightarrow{n.parent.iw}\} \rightarrow\ ^{\triangle}(x = CHILD^*_{inline})\ \{\overleftarrow{n.m} = \max_{x_i \in x}(\overleftarrow{x_i.m});\ \overleftarrow{n.p} = \sum_{x_i \in x}(\overleftarrow{x_i.p})\}
\end{aligned}
$$

$$
\begin{aligned}
&\overrightarrow{iw_{b,ib}}(\cdot,\ \cdot):\\
&\quad \mid (\overrightarrow{iw_{parent}},\ \widehat{n\,\%}) =\ iw_{parent} * n\\
&\quad \mid (\overrightarrow{iw_{parent}},\ \widehat{n\,\text{px}}) =\ n\\
&\quad \mid (\overrightarrow{iw_{parent}},\ \widehat{\text{auto}}) = iw_{parent}\\
&\overrightarrow{cw_b}(\cdot,\ \cdot,\ \cdot):\\
&\quad \mid (\_,\ \_,\ \widehat{p\,\text{px}}) = p\\
&\quad \mid (\overrightarrow{iw_{parent}},\ \_,\ \widehat{p\,\%}) = iw_{parent} * p\\
&\quad \mid (\_,\ \widehat{\text{auto}},\ \widehat{\text{auto}}) = \text{auto}\\
&\quad \mid (\_,\ \widehat{p},\ \widehat{\text{auto}}) = p\\
&\overrightarrow{cw_{ib}}(\cdot,\ \cdot):\\
&\quad \mid (\_,\ \widehat{p\,\text{px}}) = p\\
&\quad \mid (iw_{parent},\ \widehat{p\,\%}) = iw_{parent} * p\\
&\quad \mid (\_,\ \widehat{\text{auto}}) = \text{auto}\\
&\bowtie(\cdot,\ \cdot):\\
&\quad \mid (inner,\ \overrightarrow{\text{auto}}) = inner\\
&\quad \mid (inner,\ \overrightarrow{set}) = set
\end{aligned}
$$

Figure 7: Inital percentage, minimum, and preferred width calculations, assuming no $FLOAT_{left}$ elements.

$$
\begin{aligned}
&CHILD_{inline} \rightarrow INLINE_{inline} \mid INLINE_{block}\\
&CHILD_{block} \rightarrow BLOCK\\
&CHILDREN_{homog} \rightarrow CHILD^*_{block} \mid CHILD^*_{inline}\\
&(n = BLOCK[w = \widehat{cw}])\ \{\overrightarrow{n.uw} = w_b(\overrightarrow{n.parent.uw},\ \widehat{cw})\} \rightarrow\ ^{\triangle}(CHILDREN_{homog})\\
&(n = INLINE_{block}[w = \widehat{cw};\ m = \widehat{m};\ p = \widehat{p}])\ \{\overrightarrow{n.uw} = \overrightarrow{w_{ib}}(\overrightarrow{n.parent.uw},\ \widehat{cw},\ \widehat{m},\ \widehat{p})\} \rightarrow\ ^{\triangle}(CHILDREN_{homog})\\
&(n = INLINE_{inline}[\,])\ \{\overrightarrow{n.uw} = \overrightarrow{n.parent.uw}\} \rightarrow\ ^{\triangle}(CHILD_{inline})\\
&\overrightarrow{w_{ib}}(\cdot,\ \cdot,\ \cdot,\ \cdot):\\
&\quad \mid (\_,\ \widehat{n\,\text{px}},\ \_,\ \_) = n\\
&\quad \mid (\overrightarrow{uw_{parent}},\ \widehat{n\%},\ \_,\ \_) = n * uw_{parent}\\
&\quad \mid (\overrightarrow{uw_{parent}}, \widehat{\text{auto}}, \widehat{m}, \widehat{p}) = \min(\max(m, uw_{parent}),\ p)\\
&\overrightarrow{w_b} = \overrightarrow{iw_{b,ib}}
\end{aligned}
$$

Figure 8: Final used width calculation, including percentage recalculations, and assuming no $FLOAT_{left}$ elements.

$$
\begin{aligned}
&BLOCK[w = \widehat{200\text{px}}](\\
&\quad {}_aINLINE_{block}[w = \widehat{\text{auto}}](\\
&\quad\quad {}_bFLOAT_{left}[w = \widehat{100\text{px}};\ h = \widehat{\text{auto}}](\ldots)\\
&\quad\quad {}_cFLOAT_{left}[w = \widehat{50\text{px}};\ h = \widehat{\text{auto}}](\ldots)\\
&\quad\quad BLOCK[w = \widehat{100\%}](\\
&\quad\quad\quad {}_dINLINE_{block}[w = \widehat{75\text{px}}](\ldots))))
\end{aligned}
$$

Element c will be adjacent to b's right edge, but is unclear whether c's bottom edge is before or after that of b's. In the first case, a's width would be 175px. In the second case, a's width would be 150px. Thus, using perfect width and height calculation, there would be a dependency upon the heights of b and c to determine the width of a. However, a valid choice under the specification would be to define a preferred width that assumes inline content directly after floating content starts on the line after the floating content ends; this breaks the height dependency and is similar to the choice taken in Webkit.

Figure 4.4.3 formalizes the width calculations, fully comprising the second stage of layout computations.

4.5 Attribute Grammar: Heights and Positions

Height and position calculations are intertwined. As items are placed in an element, they increase the height of that element. This impacts the position of siblings of the element, and so on. Thus, we show how to compute the position of an element relative to its enclosing $BLOCK$ or $INLINE_{block}$ in lockstep with its height. Absolute positions may be computed later; partitioning the calculation introduces room for optimistic evaluation when computing heights and relative positions, and potentially SIMDization of conversion to absolute coordinates.

4.5.1 Basic Heights and Positions

Sizing and laying out elements, given the width sizes and ignoring floating elements, is fairly simple. Blocks stack vertically, and inline elements stack horizontally, starting new lines when a boundary is reached. A slight subtlety is that these inline elements are trees of elements to be positioned, not lists of them. Leaf nodes of a view of this tree, such as pictures or text, are placed in the nearest block ancestor, and the path from the root of the tree to the element specifies nested styling information, such as multiple borders.

Informally, a $BLOCK$ or $INLINE_{block}$ element may have a child tree consisting of nested $INLINE_{inline}$ and $INLINE_{block}$ elements, where every $INLINE_{block}$ element may again have $BLOCK$ elements below it. The subset of the tree of $INLINE_{block}$ elements directly reachable from the root will be positioned relative to the $BLOCK$ or $INLINE_{block}$.

We introduce a hidden variable representing a position cursor. Every inline element may push the cursor for its containing block to the right on the current line or to the beginning of the next line. Along with the cursor position, we also propagate the bounding width of the block alongside the cursor in order to determine when to make line breaks. The parent of a node annotates where the cursor is upon entry ($x.ic$), and the node returns to the parent where the cursor is upon exit ($x.ec$), where both ic and ex are singly assigned variables. The full cursor signature is:

$$C :: \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \qquad (2)$$

The parameters correspond to the top left (x, y) position in which to place the next item that fits on the line, the height of the height element on the current line since the last line break, and the width of the current allocated width for placing $INLINE$ elements before line breaking. The δ_c function takes a cursor and a rectangle, determining how to move the cursor to represent placing the rectagle, and returning the position of the rectangle and the new cursor ready for the next rectangle.

No fixpoint or undefined computations are allowed in the specification when computing heights, unlike the case for widths. When an element's height is a function of its parent's, namely by being a percentage, and the parent's height is dependent upon the element, the element's height is changed from a percentage to *auto* (the natural height of its children). The induced variable $\overrightarrow{x.ch}$ represents the computed value to use when finally solving for heights. Using structural induction, we can prove that, for the node constructors under consideration, only two passes (down for percentages and then up for $\widehat{\text{auto}}$s) are needed to solve for heights and positions.

4.5.2 Floats

As informally discussed earlier, floating elements move to one side of their *current* line until either another floating element or the parent's boundary is reached. If there is not enough room to support one, it is wrapped to the next line, at which point it is again slid to one direction. There are many subtleties to supporting them; browsers generally vary in their implementations, and this is permitted by a ambiguous specification terminolgy: "a floating box must be placed as high as *possible*", "a higher position is *preferred* over one that...", etc. We interpret these as providing local objective functions to be greedily optimized under the implicit sequential flow, which is close to what browsers typically do.

A key concept to floats is that they exist in order to break the typical visual containment implied by lexical nesting. Consider the following:

$$
\begin{aligned}
&BLOCK[w = \widehat{\text{auto}}](\\
&\quad {}_aBLOCK[w = \widehat{\text{auto}}](\\
&\quad\quad {}_bFLOAT_{left}[w = \widehat{200\text{px}};\ h = \widehat{100\text{px}}](\ldots)\\
&\quad\quad INLINE_{inline}[\,](\ldots))\\
&\quad {}_cBLOCK[w = \widehat{\text{auto}}](\\
&\quad\quad INLINE_{inline}[\,](\ldots)))
\end{aligned}
$$

Blocks a and c might be two paragraphs of text, where a might also contain a picture within b that pushes into c. $BLOCK$ elements do not consider $FLOAT$ elements when computing heights, supporting this feature. However, both $FLOAT$ and $INLINE_{block}$ elements *do* include $FLOAT$ elements during height calculations. Somewhat breaking dependencies, a $FLOAT$ element is only considered up to the next flow root ancestor, which, in our model, is denoted by $FLOAT$ elements. They follow the sequential flow: floats may extend below their containing block, but not to the

$$CHILD_{inline} \rightarrow INLINE_{inline} \mid INLINE_{block}$$
$$CHILD_{block} \rightarrow BLOCK$$
$$(n = BLOCK[w = \widehat{cw}])\ \{\overrightarrow{n.iw} = \overrightarrow{iw_{b,ib}}(\widehat{cw},\ \overrightarrow{n.parent.iw});\ \overrightarrow{n.cw} = \overrightarrow{cw_b}(\overrightarrow{n.parent.iw},\ \overrightarrow{n.parent.cw},\ \widehat{cw})\}$$
$$\mid \rightarrow\ ^{\triangle}(f_i = FLOAT^*_{left} \mid b_i = BLOCK^*)$$
$$\{\overleftarrow{n.m} = (\max_{f_{elt}\in f_i\in f,\ b_{elt}\in b_i\in b}(\overleftarrow{f_{elt}.m},\ \overleftarrow{b_{elt}.m})) \bowtie \overrightarrow{n.cw};$$
$$\overleftarrow{n.p} = (\max_{f_i\in f,\ b_i\in b}(\max(\sum_{f_{elt}\in f_i} \overleftarrow{f_{elt}.p},\ \max_{b_{elt}\in b_i}(\overleftarrow{b_{elt}.p})))) \bowtie \overrightarrow{n.cw}\}$$
$$\mid \rightarrow\ ^{\triangle}(x = CHILD^*_{inline})\ \{\overleftarrow{n.m} = (\max_{x_i\in x}(\overleftarrow{x_i.m})) \bowtie \overrightarrow{n.cw};\ \overleftarrow{n.p} = \sum_{x_i\in x}(\overleftarrow{x_i.p}) \bowtie \overrightarrow{n.cw}\}$$
$$(n = (INLINE_{block} \mid FLOAT_{left}))[w = \widehat{cw}])\ \{\overrightarrow{n.iw} = \overrightarrow{iw_{b,ib}}(\widehat{cw},\ \overrightarrow{n.parent.iw});\ \overrightarrow{n.cw} = \overrightarrow{cw_{ib}}(\overrightarrow{n.parent.iw},\ \widehat{cw})\}$$
$$\mid \rightarrow\ ^{\triangle}(f_i = FLOAT^*_{left} \mid b_i = BLOCK^*)$$
$$\{\overleftarrow{n.m} = (\max_{f_{elt}\in f_i\in f,\ b_{elt}\in b_i\in b}(\overleftarrow{f_{elt}.m},\ \overleftarrow{b_{elt}.m})) \bowtie \overrightarrow{n.cw};$$
$$\overleftarrow{n.p} = (\max_{f_i\in f,\ b_i\in b}(\max(\sum_{f_{elt}\in f_i} \overleftarrow{f_{elt}.p},\ \max_{b_{elt}\in b_i}(\overleftarrow{b_{elt}.p})))) \bowtie \overrightarrow{n.cw}\}$$
$$\mid \rightarrow\ ^{\triangle}(x = CHILD^*_{inline})\ \{\overleftarrow{n.m} = (\max_{x_i\in x}(\overleftarrow{x_i.m})) \bowtie \overrightarrow{n.cw};\ \overleftarrow{n.p} = \sum_{x_i\in x}(\overleftarrow{x_i.p}) \bowtie \overrightarrow{n.cw}\}$$
$$(n = INLINE_{inline})[\,]\{\overrightarrow{n.iw} = \overrightarrow{n.parent.iw}\} \rightarrow\ ^{\triangle}(x = CHILD^*_{inline})\ \{\overleftarrow{n.m} = \max_{x_i\in x}(\overleftarrow{x_i.m});\ \overleftarrow{n.p} = \sum_{x_i\in x}(\overleftarrow{x_i.p})\}$$

$$\overrightarrow{iw_{b,ib}}(\cdot,\ \cdot):$$
$$\mid (\overrightarrow{iw_{parent}},\ \widehat{n\,\%}) =\ iw_{parent} * n$$
$$\mid (\overrightarrow{iw_{parent}},\ \widehat{n\,\mathrm{px}}) =\ n$$
$$\mid (\overrightarrow{iw_{parent}},\ \widehat{\mathrm{auto}}) = iw_{parent}$$
$$\overrightarrow{cw_b}(\cdot,\ \cdot,\ \cdot):$$
$$\mid (\_,\ \_,\ \widehat{p\,\mathrm{px}}) = p$$
$$\mid (\overrightarrow{iw_{parent}},\ \_,\ \widehat{p\,\%}) = iw_{parent} * p$$
$$\mid (\_,\ \widehat{\mathrm{auto}},\ \widehat{\mathrm{auto}}) = \mathrm{auto}$$
$$\mid (\_,\ \widehat{p},\ \widehat{\mathrm{auto}}) = p$$
$$\overrightarrow{cw_{ib}}(\cdot,\ \cdot):$$
$$\mid (\_,\ \widehat{p\,\mathrm{px}}) = p$$
$$\mid (iw_{parent},\ \widehat{p\,\%}) = iw_{parent} * p$$
$$\mid (\_,\ \widehat{\mathrm{auto}}) = \mathrm{auto}$$
$$\bowtie (\cdot,\ \cdot):$$
$$\mid (inner,\ \overrightarrow{\mathrm{auto}}) = inner$$
$$\mid (inner,\ \overrightarrow{set}) = set$$

Figure 10: Inital percentage, minimum, and preferred width calculations.

$$CHILD_{inline} \rightarrow INLINE_{inline} \mid INLINE_{block}$$
$$CHILD_{block} \rightarrow BLOCK$$
$$CHILDREN_{homog} \rightarrow CHILD^*_{block} \mid CHILD^*_{inline}$$
$$(n = BLOCK[w = \widehat{cw}])\ \{\overrightarrow{n.uw} = w_b(\overrightarrow{n.parent.uw},\ \widehat{cw})\} \rightarrow\ ^{\triangle}(CHILDREN_{homog})$$
$$(n = (INLINE_{block} \mid FLOAT_{left})[w = \widehat{cw};\ m = \widehat{m};\ p = \widehat{p}])\ \{\overrightarrow{n.uw} = \overrightarrow{w_{ib}}(\overrightarrow{n.parent.uw},\ \widehat{cw},\ \widehat{m},\ \widehat{p})\} \rightarrow\ ^{\triangle}(CHILDREN_{homog})$$
$$(n = INLINE_{inline}[\,])\ \{\overrightarrow{n.uw} = \overrightarrow{n.parent.uw}\} \rightarrow\ ^{\triangle}(CHILD_{inline})$$
$$\overrightarrow{w_{ib}}(\cdot,\ \cdot,\ \cdot,\ \cdot):$$
$$\mid (\_,\ \widehat{n\,\mathrm{px}},\ \_,\ \_) = n$$
$$\mid (\overrightarrow{uw_{parent}},\ \widehat{n\%},\ \_,\ \_) = n * uw_{parent}$$
$$\mid (\overrightarrow{uw_{parent}}, \widehat{\mathrm{auto}}, \widehat{m}, \widehat{p}) = \min(\max(m, uw_{parent}),\ p)$$
$$\overrightarrow{w_b} = \overrightarrow{iw_{b,ib}}$$

Figure 11: Final used width calculation, including percentage recalculations.

$$CHILD_{inline} \rightarrow INLINE_{inline} \mid INLINE_{block}$$

$$CHILD_{block} \rightarrow BLOCK$$

$$\begin{aligned}
&(n = BLOCK[h = \widehat{i}])\ \{\overrightarrow{n.ch} = \overrightarrow{ch_{b,ib}}(\overrightarrow{n.parent.ch}, \widehat{i})\} :\\
&\qquad \mid \{h_{acc} \hookleftarrow 0\} \rightarrow\ ^{\triangle}((\{x.x = 0\}\ x = CHILD_{block}\ \{x.y = h_{acc};\ h_{acc} \hookleftarrow h_{acc} + x.uh\}\ )^{*})\\
&\qquad\quad \{n.uh = bh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ h_{acc})\}\\
&\qquad \mid \{c_{acc} \hookleftarrow (0, 0, 0, \widehat{n.uw})\} \rightarrow\ ^{\triangle}((\{x.c_i = c_{acc}\}\ x = CHILD_{inline}\ \{c_{acc} \hookleftarrow x.c_o\})^{*})\\
&\qquad\quad \{n.uh = bh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ c_{acc}[1] + c_{acc}[2])\}
\end{aligned}$$

$$\begin{aligned}
&(n = INLINE_{inline}[\,])\ \{c_{acc} \hookleftarrow n.c_i;\ \overrightarrow{n.ch} = \overrightarrow{n.parent.ch}\} \rightarrow\\
&\qquad ^{\triangle}((\{x.c_i = c_{acc}\}\ x = CHILD_{inline}\ \{c_{acc} \hookleftarrow x.c_o\})^{*})\ \{n.c_o = c_{acc}\}
\end{aligned}$$

$$\begin{aligned}
&(n = INLINE_{block}[h = \widehat{i}])\ \{\overrightarrow{n.ch} = \overrightarrow{ch_{b,ib}}(\overrightarrow{n.parent.ch}, \widehat{i})\} :\\
&\qquad (\ \mid \{h_{acc} \hookleftarrow 0\} \rightarrow\ ^{\triangle}((\{x.x = 0\}\ x = CHILD_{block}\ \{x.y = h_{acc};\ h_{acc} \hookleftarrow h_{acc} + x.uh\}\ )^{*})\\
&\qquad\quad \{n.uh = bh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.parent.ch},\ h_{acc})\}\\
&\qquad \mid \{c_{acc} \hookleftarrow (0, 0, 0, \widehat{n.uw})\} \rightarrow\ ^{\triangle}((\{x.c_i = c_{acc}\}\ x = CHILD_{inline}\ \{c_{acc} \hookleftarrow x.c_o\})^{*})\\
&\qquad\quad \{n.uh = bh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.parent.ch}, c_{acc}[1] + c_{acc}[2])\}\\
&\qquad \mid \{n.uh = LH\}\ \mathrm{C}^{+})\\
&\qquad \{(n.c_o,\ n.x,\ n.y) = \delta_c(n.c_i, n.w, n.uh)\}
\end{aligned}$$

$$\begin{aligned}
&\overrightarrow{ch_{b,ib}}(\cdot,\ \cdot) :\\
&\qquad \mid (\overrightarrow{ch},\ \widehat{\text{auto}}) = \text{auto}\\
&\qquad \mid (\overrightarrow{\text{auto}},\ \widehat{p\,\%}) = \text{auto}\\
&\qquad \mid (\overrightarrow{n},\ \widehat{p\,\%}) = n * p\ \text{px}\\
&\qquad \mid (\overrightarrow{ch},\ \widehat{p\,\text{px}}) = p\ \text{px}\\
&bh(\cdot,\ \cdot,\ \cdot) :\\
&\qquad \mid (h_{parent},\ \overrightarrow{n\,\%},\ h_{children}) =\ h_{parent} * n\\
&\qquad \mid (h_{parent},\ \overrightarrow{n\,\text{px}},\ h_{children}) =\ n\\
&\qquad \mid (h_{parent},\ \overrightarrow{\text{auto}},\ h_{children}) =\ h_{children}\\
&\delta_c((\cdot,\ \cdot,\ \cdot,\ \cdot),\ \cdot,\ \cdot) :\\
&\qquad \mid ((x,\ y,\ my,\ w),\ w_{ib},\ h_{ib})\ \text{where}\ (x > 0\ \wedge\ x + w_{ib} > w)\\
&\qquad\quad =\ ((w_{ib},\ y + my,\ h_{ib},\ w),\ 0,\ y + my)\\
&\qquad \mid ((x,\ y,\ my,\ w),\ w_{ib},\ h_{ib})\ \text{where}\ \neg(x > 0\ \wedge\ x + w_{ib} > w)\\
&\qquad\quad =\ ((x + w_{ib},\ y,\ \max(my, h_{ib}),\ w),\ x,\ y)
\end{aligned}$$

Figure 12: Basic height and relative position calculations based on used widths, ignoring $FLOAT_{left}$ elements.

left, right, nor above. Finally, while we could not ascertain exactly why relative to the specification, in practice, they must also go *below* earlier sibling blocks.

The essential idea to incorporating floats is to pass around the space occupied by previous floats in the same context under the current flow root. Just as the cursor was threaded and incremented for $INLINE_{block}$ elements, a space manager is threaded through all elements. Before, the cursor was reset when a new $BLOCK$ or $INLINE_{block}$ was encountered; the space manager is reset whenever a new $FLOAT_{left}$ element is encountered. In addition, as $FLOAT$ and $INLINE_{block}$ elements include heights from $FLOAT$s one nesting deep, we need to also reify this information.

Finally, before, we positioned elements relative to the nearest block. Now, as we must consider the space context, which is relative to the nearest enclosing $FLOAT$, we must somehow incorporate space described under an alternate coordinate system. Instead of positioning elements relative to the nearest enclosing $FLOAT$, we push the burden to the consideration of space shapes, continuing our support of modularity in optimistic settings. Thus, when the space is threaded to an element, so is an offset.

Interesting stuff about float heights:

- Generally, s_o represents space taken by floats from within an element. However, this is not true of $INLINE_{block}$ elements, which returns the known space taken floats from floats within the elements or preceeding siblings from the same line. This does not include all preceeding siblings; for example, as floats might be pushed to the next line, and later siblings may increase the line height, these are not known and thus not yet included.
-

TODO:

- Do inline-blocks include floats in auto-heights?
- Floats do not clear lines

Proofs

- **Single-assignment of node attributes**. Most follow attribute grammar recursion directly. Exception is y coordinates of inline-blocks and floats; they are set either following a δ_{ib} or TODO. However, they are passed around when not set, and removed from passing around when set.
- Type of constraints?

$$(n = BLOCK[h = \widehat{i}])\ \{\overrightarrow{n.ch} = \overrightarrow{ch_{b,ib}}(\overrightarrow{n.parent.ch}, \widehat{i}); n.x = 0;\ n.y = n.c_i[1];\ s_{acc} \hookleftarrow (\emptyset,\ 0);\ c_{acc} \hookleftarrow (0,\ 0,\ 0,\ \widehat{n.uw},\ [\,],\ [\,])\} :$$

$$(\ |\ \rightarrow\ ^{\triangle}((((\{f.s_i = s_{acc} \uplus n.s_i;\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{b.c_i = c_{acc};\ b.s_i = \searrow (s_{acc} \uplus n.s_i,\ c_{acc})\}\ b = BLOCK$$

$$\{s_{acc} \hookleftarrow\ s_{acc} \uplus \nwarrow (b.s_o,\ c_{acc});\ c_{acc} \hookleftarrow \delta_b(b.h,\ c_{acc})\}))^*)$$

$$\{n.uh = bh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ c_{acc}[1]);\ n.s_o = s_{acc}\}$$

$$|\ \rightarrow\ ^{\triangle}((((\{f.s_i = s_{acc} \uplus n.s_i;\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{x.s_i = s_{acc} \uplus n.s_i;\ x.c_i = c_{acc}\}\ x = (INLINE_{inline}\ |\ INLINE_{block})$$

$$\{s_{acc} \hookleftarrow s_{acc} \uplus x.s_o;\ c_{acc} \hookleftarrow x.c_o\}))^*)$$

$$finishline$$

$$\{n.uh = bh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ c_{acc}[1] + c_{acc}[2]);\ n.s_o =\ s_{acc} \uplus \Downarrow (c_{acc})\})$$

$$(n = INLINE_{inline}[\,])\ \{\overrightarrow{n.ch} = \overrightarrow{n.parent.ch};\ s_{acc} \hookleftarrow (\emptyset,\ 0);\ c_{acc} \hookleftarrow n.c_i)\} \rightarrow$$

$$^{\triangle}((((\{f.s_i = s_{acc} \uplus n.s_i;\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{x.s_i = s_{acc} \uplus n.s_i;\ x.c_i = c_{acc}\}\ x = (INLINE_{inline}\ |\ INLINE_{block})\ \{s_{acc} \hookleftarrow s_{acc} \uplus x.s_o;\ c_{acc} \hookleftarrow x.c_o\}))^*)$$

$$\{n.s_o = s_{acc};\ n.c_o = c_{acc}\}$$

$$(n = INLINE_{block}[h = \widehat{i}])\ \{\overrightarrow{n.ch} = \overrightarrow{ch_{b,ib}}(\overrightarrow{n.parent.ch}, \widehat{i});\ s_{acc} = (\emptyset,\ 0);\ c_{acc} \hookleftarrow (0,\ 0,\ 0,\ \widehat{n.uw},\ [\,],\ [\,])\} :$$

$$(\ |\ \rightarrow\ ^{\triangle}((((\{f.s_i = s_{acc} \uplus n.s_i;\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{b.c_i = c_{acc};\ b.s_i = \searrow (s_{acc} \uplus n.s_i,\ c_{acc})\}\ b = BLOCK$$

$$\{s_{acc} \hookleftarrow\ s_{acc} \uplus \nwarrow (b.s_o,\ c_{acc});\ c_{acc} \hookleftarrow \delta_b(b.h,\ c_{acc})\}))^*)$$

$$\{n.uh = rh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ c_{acc}[1],\ s_{acc})\}$$

$$|\ \rightarrow\ ^{\triangle}((((\{f.s_i = s_{acc} \uplus n.s_i;\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{x.s_i = s_{acc} \uplus n.s_i;\ x.c_i = c_{acc}\}\ x = (INLINE_{inline}\ |\ INLINE_{block})$$

$$\{s_{acc} \hookleftarrow s_{acc} \uplus x.s_o;\ c_{acc} \hookleftarrow x.c_o\}))^*)$$

$$finishline$$

$$\{n.uh = rh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ c_{acc}[1] + c_{acc}[2],\ s_{acc})\}$$

$$|\ \{n.uh = LH\}\ \mathrm{C}^+$$

$$\{(n.c_o,\ n.s_o,\ (xys_{ib},\ xs_{ib}),\ (xys_f,\ xs_f)) = \delta_{ib}(n.c_i,\ n.uw,\ n.uh,\ s_{acc},\ n.s_i,\ n);$$

$$\text{for } i\ \ldots\ |xs_f| : xs_f[i].x = xys_f[i][0];\ xs_f[i].y = xys_f[i][1];$$

$$\text{for } i\ \ldots\ |xs_{ib}| : xs_{ib}[i].x = xys_{ib}[i][0];\ xs_{ib}[i].y = xys_{ib}[i][1]\ \}$$

$$(n = FLOAT_{left}[h = \widehat{i}])\ \{\overrightarrow{n.ch} = \overrightarrow{ch_{b,ib}}(\overrightarrow{n.parent.ch}, \widehat{i});\ s_{acc} = (\emptyset,\ 0);\ c_{acc} \hookleftarrow (0,\ 0,\ 0,\ \widehat{n.uw},\ [\,],\ [\,])\} :$$

$$(\ |\ \rightarrow\ ^{\triangle}((((\{f.s_i = s_{acc};\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{b.c_i = c_{acc};\ b.s_i = \searrow (s_{acc},\ c_{acc})\}\ b = BLOCK$$

$$\{s_{acc} \hookleftarrow\ s_{acc} \uplus \nwarrow (b.s_o,\ c_{acc});\ c_{acc} \hookleftarrow \delta_b(b.h,\ c_{acc})\}))^*)$$

$$\{n.uh = rh(\overrightarrow{n.parent.ch},\ \overrightarrow{n.ch},\ c_{acc}[1],\ s_{acc})\}$$

$$|\ \rightarrow\ ^{\triangle}((((\{f.s_i = s_{acc};\ f.c_i = c_{acc}\}\ f = FLOAT_{left}\ \{s_{acc} \hookleftarrow s_{acc} \uplus f.s_o;\ c_{acc} \hookleftarrow f.c_o\})$$

$$|\ (\{x.s_i = s_{acc};\ x.c_i = c_{acc}\}\ x = (INLINE_{inline}\ |\ INLINE_{block})$$

$$\{s_{acc} \hookleftarrow s_{acc} \uplus x.s_o;\ c_{acc} \hookleftarrow x.c_o\}))^*)$$

$$finishline$$

$$\{(f.x,\ f.y) = \delta_f(n.c_i,\ (n.s_i[0] \cup n.parent.s_i,\ n.s_i[1]),\ \widehat{f.uw},\ f.uh);$$

$$n.s_o \hookleftarrow ([n.x,\ n.x + \widehat{n.uw}] \times [n.y,\ n.y + n.uh],\ n.y)\}$$

Figure 13: Full height and relative position calculations: structure.

$$
\begin{aligned}
s_1 \uplus s_2 &= (s_1[0] \cup s_2[0],\ \max(s_1[1],\ s_2[1])) \\
\searrow (s,\ c_{acc}) &= (s[0] - (0, c_{acc}[1]),\ s[1] - c_{acc}[1]) \\
\nwarrow (b.s_o,\ c_{acc}) &= (b.s_o[0] + (0,\ c_{acc}[1]),\ b.s_o[1] + c_{acc}[1]) \\
starts(cw,\ my,\ s,\ w,\ h) &= (x,\ y) \text{ where } \operatorname{argmin}_x\ \operatorname{argmin}_y([x,\ \infty] \times [y,\ y+h] \cap s_i[0] = \emptyset) \\
&\quad \wedge\ (y \geq my) \wedge (x \neq 0 \rightarrow x + w < cw) \\
\downarrow (c,\ s) &= \text{let } (x_s,\ y_s) = starts(c[3],\ c[1],\ s,\ max(1,\ (\operatorname{argmax}_{x \in c[4]} x.uh).uh),\ c[2]) \text{ in} \\
&\quad fold(\lambda(n,\ (x,\ s_{acc})).\ (x + n.uw,\ s_{acc} \uplus ([x.x,\ x.x + x.uw] \times [y_s,\ y_s + x.uh])), \\
&\qquad (x_s,\ (\emptyset,\ 0)),\ c[4])[1] \\
\Downarrow (c,\ s) &= \text{let } (x_s,\ y_s) = starts(c, s) \text{ in} \\
&\quad fold(\lambda(x,\ (y_{acc},\ s_{acc})). \\
&\qquad \text{let } (x,\ y) = starts(c[3],\ y_{acc},\ s_{acc},\ x.uw,\ x.uh) \text{ in} \\
&\qquad\quad (y,\ s_{acc} \uplus [x,\ x + x.uw] \times [y,\ y + x.uh]), \\
&\qquad (y_s + (\operatorname{argmax}_{x \in c[4]} x.h).h,\ \downarrow (c,\ s)),\ c[5])[1] \\
\delta_b(b.h\ , c_{acc}) &= (0,\ c_{acc}[1] + b.h,\ \bot,\ c_{acc}[3],\ [\,],\ [\,])
\end{aligned}
$$

$$
\begin{aligned}
&\overrightarrow{ch_{b,ib}}(\cdot,\ \cdot) : \\
&\quad |\ (\overrightarrow{ch},\ \widehat{\text{auto}}) = \text{auto} \\
&\quad |\ (\overrightarrow{\text{auto}},\ \widehat{p\,\%}) = \text{auto} \\
&\quad |\ (\overrightarrow{n},\ \widehat{p\,\%}) = n * p \text{ px} \\
&\quad |\ (\overrightarrow{ch},\ \widehat{p \text{ px}}) = p \text{ px}
\end{aligned}
$$

$$
\begin{aligned}
\delta_f((x,\ y,\ \_,\ w,\ \_,\ fs),\ (P_f,\ \_),\ w_f,\ h_f) &= (x_c,\ y_c) \text{ where} \\
&\operatorname{argmin}_{y_c} \operatorname{argmin}_{x_c}\ (x_c + w_f \leq w)\ \wedge\ ((y = y_c) \rightarrow (x \leq x_c))\ \wedge\ (y_c \geq y) \\
&\qquad \wedge\ (\neg\exists (x_f,\ y_f) \in P_f.\ (x_f,\ y_f) \in ([x_c,\ x_c + w_f] \times [y_c,\ y_c + h_f])) \\
&\qquad \wedge\ (\neg\exists (x_f,\ y_f) \in P_f.\ (x_f > x_c)\ \wedge\ (y_f \geq y_c))
\end{aligned}
$$

$$
\begin{aligned}
&bh(\cdot,\ \cdot,\ \cdot) : \\
&\quad |\ (h_{parent},\ \overrightarrow{n\,\%},\ \_) =\ h_{parent} * n \\
&\quad |\ (\_,\ \overrightarrow{n \text{ px}},\ \_) =\ n \\
&\quad |\ (\_,\ \overrightarrow{\text{auto}},\ h_{children}) =\ h_{children} \\
&rh(\cdot,\ \cdot,\ \cdot,\ \cdot) : \\
&\quad |\ (h_{parent},\ \overrightarrow{n\,\%},\ \_,\ \_) =\ h_{parent} * n \\
&\quad |\ (\_,\ \overrightarrow{n \text{ px}},\ \_,\ \_) =\ n \\
&\quad |\ (\_,\ \overrightarrow{\text{auto}},\ h_{children},\ s) =\ \max(h_{children}, \max_{(\_,\ y) \in s[0]}(y))
\end{aligned}
$$

$$
\begin{aligned}
position_{ib}(c = (x_c,\ y_c,\ \_,\ w_c,\ \_,\ \_),\ w_n,\ h_n,\ s_i) &= (x_n,\ y_n) \text{ where} \\
&\operatorname{argmin}_{y_n} \operatorname{argmin}_{x_n}\ (w_n \leq w_c \rightarrow x_n + w_n \leq w_c) \\
&\qquad \wedge\ ((y_c = y_n) \rightarrow (x_n \geq x_c)) \\
&\qquad \wedge\ (y_n \geq y_c) \\
&\qquad \wedge\ (w_n > w_c \rightarrow x_n = 0) \\
&\qquad \wedge\ ((s_i \uplus \downarrow (c,\ s_i))[0]\ \cap\ ([x_n,\ x_n + w_n] \times [y_n,\ y_n + h_n]) = \emptyset) \\
&\qquad \wedge\ (y_n \neq y_c \rightarrow\ (\Downarrow (c,\ s_i))[0]\ \cap\ ([x_n,\ x_n + w_n] \times [y_n,\ y_n + h_n]) = \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
\delta_{ib}(c_i,\ w_n,\ h_n,\ s_i,\ n) &\text{ where } (x_n,\ y_n) = \text{position}_{ib}(c_i,\ w_n,\ h_n,\ s_i) : \\
&|\ ((x_c,\ y_c,\ my_c,\ w_c,\ line,\ fs),\ w_n,\ h_n,\ s_i) \text{ where } (y_c = y_n) \\
&\quad =\ ((x_n,\ y_c,\ \max(my_c,\ h_n),\ w_c,\ line :: n,\ fs),\ (\emptyset,\ 0),\ [\,],\ [\,]) \\
&|\ ((x_c,\ y_c,\ my_c,\ w_c,\ line,\ fs),\ w_n,\ h_n,\ s_i) \text{ where } (y_c \neq y_n) \\
&\quad =\ ((x_n,\ y_n,\ h_n,\ w_n,\ [\,],\ [\,]),\ ([x_n,\ x_n + w_n] \times [y_n,\ y_n + h_n]) \uplus \Downarrow (c_i, s_i),\ reflow,\ fs)
\end{aligned}
$$

Figure 14: Full height and relative position calculations: helpers