FOP Design: Layout Managers

Break Possibility Proposal

$Revision: 426576 $

by Karen Lease

Table of contents

1. Introduction

As explained in Layout, the hierarchy of Layout Managers is responsible for building and placing areas. Each Layout Manager is responsible for creating and filling areas of a particular type, either inline or block. This document explains one potential algorithm for this process. It is based on the the generation of *break possibilities* (BP for short). The Layout Managers (LM for short), will generate one or more BP and choose the best one. The BP is then used to generate the corresponding areas.

2. Anatomy of a Break Possibility

A break possibility is represented by the BreakPoss class. A BreakPoss contains size information in the stacking direction and in the non-stacking direction (at least for inline areas, it must have both). Flags indicating various conditions (ISFIRST, ISLAST, CAN_BREAK_AFTER, FORCE_BREAK_AFTER, ANCHORS etc). A BreakPoss contains a reference to the top-level LayoutManager which generated it.

A BreakPoss contains an object implementing the BreakPoss.Position interface. This object is specific to the layout manager which created the BreakPoss. It should indicate where the break occurs and allow the LM to create an area corresponding to the BP. A higher level LM Position must somehow reference or wrap the Position returned by its child LM in its BreakPoss object. The layout manager modifies the flags and dimension information in the BP to reflect its own requirements. For example an inline FO layout manager might add space-start, space-end, border and padding values to the stacking or non-stacking dimensions. It might also modify the flags based its on keep properties.

3. Turning Break Possibilities into Areas

Once break possibilities have been generated, the galley-level layout manager selects the best one and passes it back to the LayoutManager which generated it to create the area. A LayoutManager is responsible for storing enough information in its Position objects to be able to create the corresponding areas.

4. A walk-through

Layout Managers are created from the top down. First the page sequence creates a PageLM and a FlowLM. The PageLM will manage finding the right page model (with help from the

PageSequenceMaster) and managing the balancing act between before-floats, footnotes and the normal text flow. The FlowLM will manage the normal content in the main flow. We can think of it as a *galley* manager.

In general, each LM asks its child LMs to return sucessive break possibilities. It passes some information to the child in a flags object and it gets back a break possibility which contains the size in the stacking direction as well as information about such things as anchors, break conditions and span conditions which can change the reference area environment. This process continues down to the lowest level of the layout manager hierarchy which corresponds to atomic inline-level FOs such as characters or graphics.

Each layout manager will repeatedly call getNextBreakPoss on its current child LM until the child returns a BP with the ISLAST flag set. Then the layout manager moves on to its next child LM (ie, it asks the next child FO to generate a layout manager.) Galley level layout managers which are Line and Flow will return to their parent layout managers either when they have finished their content or when they encounter a a BP which will fill one of their areas.

The break possibilities are generated from the bottom up. All inline content must first be broken into lines which are then stacked into block areas. This is done by the LineLayoutManager, which creates line areas. The LineLM asks its child LM to generate a break possibility, which represents a place where the line can end. This initially means each potential line-end (primarily spaces or forced linefeeds and a few other potential line-end characters such as hard hyphens.) The text LM returns an object which stores the size in the stacking direction as a MinOptMax triplet and a *cost*, which is based on how well this break would satisfy the constraints. The Text LM keeps track of its position in the text content and returns the total size of the text area it would create if it were to break at a given point. The returned BP object also contains information about whether the break is forced (linefeed) or whether this is the last area which can be generated by the LM (ISLAST flag). If a textFO ends on a non-break character, the ISLAST flag is set, but the CAN_BREAK_AFTER flag isn't, since we don't know if there is any following text in another inline object for example.

Variable size content is taken into account from the bottom up. Each LM returns a range of sizes in the stacking direction, based on property values. For text, this comes from variable word-space values or letter-space values. For other inline objects, it may include variable space-start and space-end values (after calculation of the entire sequence of space specifiers at a particular break possibility.)

The main constraint for laying out lines is the available inline-progression-dimension (IPD) for the line area to be created. This depends on the IPD of the reference area ancestor, on the indents of the containing fo:block, and on any side-floats which may be intruding on this line.

Note:

See below Getting the Reference IPD for discussion of how the reference area IPD is transmitted to the Line LM.

For now, let's assume that only the LineLM knows about the IPD available to it. Therefore only it can make a decision about which BP is the best one; the lower level inline layout managers can only return potential break points.

Note:
There are certainly optimizations to this model which can be examined later.

So the Line LM will ask its child LM(s) for break possibilities until it gets back a BP whose stacking dimension *could* fill the line. This means that the BP.stackdim.max >= LineIPD.min. It can look for further BP, perhaps one whose stackdim.opt is closer to the LineIPD.opt. If it isn't happy with the choice of break possibilities, it can go past the end of the line to the next one, and then try to find a hyphenation point between the last one which fits and the first one which doesn't. If no possibility is found whose min/max values enclose the available IPD, some constraint will be violated (and reported in the log.) The actual strategy is up to the Line LM and should be able to be easily replaced without changing the architecture (Strategy pattern).

The definition of a good break possibility depends on the properties at the block and inline level which govern things such as wrapping behavior and justification mode. For example, if lines are not to be wrapped, only an explicit linefeed can serve as a BP. If lines are wrapped but not justified then there is no requirement to completely fill the IPD on each line, but a sophisticated layout manager will try to achieve "aesthetic rag".

Note that no areas have actually been created yet. Once the LineLM has found a potential break point for the inline content, it can calculate the total size of the line area which would be created. The size in the IPD is determined by the Line LM based on the chosen BP. The size of the line area in the the block-progression-dimension depends on the size of the text (or other inline content). These values are set by the inline-level LM in their returned BP (in terms of ascender and descender heights with respect to the baseline). The LineLM adds spacing implied by the current line-stacking strategy and line-height property values. It stores a reference to the chosen inline BP and "wraps" that in its own Position object which it stores in the BP it returns to its parent LM (the block layout manager).

The block LM now has a potential break position after its first line. It assigns that possibility a cost, based on widow, orphan and keep properties. It can also calculate the total size of the block area it would create, were it to end the area after this line. It does this by adding any padding and border (taking into account conditionality). It also calculates space-before and space-after values, or contributes to building up a sequence of such values. With this information, the block LM creates a new BP (or updates the existing one). It stores a Position object in this BP which wraps the returned BP from its child Line LM. It returns the new BP to its parent and so on, back up to

the FlowLM.

Obviously there is more complicated logic involved when dealing with lists and tables. These cases need to be walked through in detail.

The FlowLM sees if the returned stacking dimension will still fit in its available block-progression-dimension (BPD). It repeatedly calls getNextBreakPoss on its child LMs until it reaches the maximum BPD for the flow reference area or until there is no more content to lay out. If one child LM is finished, it moves on to the next until the last child LM has returned a BP with the ISLAST flag set. If any child LM returns a BP with a FORCE_BREAK_BEFORE or SPAN flag set, the FlowLM will force layout of any pending break possibilities and return to its parent (the PageLM) in order to handle the break or span condition.

If the returned BP has any new before-float or footnote anchors in it (ANCHOR flag in the BP), the FlowLM will also return to the PageLM. The PageLM must then try to find space to place the floats, possibly asking the FlowLM for help if the body contains multiple columns.

5. Some issues

Following are a few remarks on specific issues.

5.1. Where Line Layout Managers are created

If the first child FO in a block FO is an inline-level FO such as text, the block LM creates an intermediate level LineLM to layout the sequence of inline content into Lines. Note that the whole sequence of inline FOs is managed by a single instance of LineLM. The LineLM becomes the parent to the various inline-level LM created by each individual inline FO. Since an fo:block can have both block and inline content, its LM may create a sequence of intermixed BlockLM and LineLM.

5.2. Getting the reference IPD

When the layout process starts, with the FlowLM asking its first child LM for a break possibility, the IPD isn't known, since we don't know whether the first FO might be spanning, or on which page it might start. (Of course, if all page masters in the sequence have the same region-body IPD and all have only a single column, the IPD will never change and could already be calculated before starting layout.) The FlowLM gets its first child LM and calls its getNextBreakPoss method. That is a child LM for some block-level FO. For now, suppose it's an fo:block. The BlockLM will create its first child LM, which may be another block-level LM in the case of nested blocks or a LineLM as explained above. (Question: do we need a START flag

for layout status?)

We keep calling getNextBreakPoss on lower level layout managers until we get down to the inline level or to a level which cannot have break-before properties, such as a list-item-label. At that point, we assume we are going to have to layout some actual content. But we can't do that yet since we don't know the inline-progression-dimension. So we return a BP object which has 0 size in the stacking dimension, but which has flags set to signal to higher-level layout managers what needs to be done. If it has a break-before property or a span property, it stores these in the BP. If no reference IPD is yet defined, it sets a flag to get that. It then returns to its parent. The parent LM will inspect the BP object returned. In general, it "wraps" it with information about its own needs. If the returned BP is not actually returning any potential areas, the LM can still add information about its own break or span requirements. This return path continues back up to the PageLM. It will then check break and span requirements and create a new page if necessary using the appropriate page-master. At that point, the reference IPD for the main flow is known and is set in the flags object used for the next getNextBreakPoss call to the lower level LM.

Using this information, the BlockLM parent can now calculate the available IPD for its LineLM child, based on its indents. (If there are any side-floats information about the intrusion must be passed down by the FlowLM to lower level managers.) The LineLM can now generate a series of BreakPoss objects, which it passes back to its parent LM.

5.3. Hyphenation

The LineLM is responsible for initiating hyphenation if it is allowed by the properties and if no satisfactory BP can be found without hyphenating. The hyphenation manager is passed two break possibilities, one whose IPD is less than the desired line area IPD and one whose IPD is greater. These break possibilities might have been generated by different inline-level layout managers (text + a wrapper with a color change for example), though frequently they represent two positions in a single text run. If hyphenation is successful, a new BP is returned. The LineLM may look for several intermediate BP based on the "cost" of the returned possibilities. If no intermediate BP is found, the line will be "short", the white-space stretch will be exceeded, or perhaps the content will be overflowed or clipped, depending on various property settings.

5.4. Optimizing

It obviously seems inefficient to go down to the lowest level LM and back up to the FlowLM for every possible line-break decision. It seems like it would be possible to optimize by letting the lower level layout managers run until they had exceeded the current limit in the stacking direction. They would then return control to the "galley" level (LineLM or FlowLM) which would fine-tune the break decision by asking the lower level LM to find a previous BP which

would fit. At the inline level, this means hyphenation as described above.

Another interesting question is at what point pending break possibilities can be turned into areas.The idea is to wait until we are sure we won't have to redo the breaking. This depends on the sophistication of the layout strategy. For example, if a linebreak can be considered final if the line is full and there are no anchors on the line, we could create the LineArea at that point. But if we are willing to change a previous line-end decision to get a better overall composition of a whole group of lines (to prevent multiple hyphens for example), we might wait until the LineLM had finished laying out all its material and then make all the Lines at once.