Notes on Layout

Callum Mole

May 28, 2021

1 Introduction

In 'Grid types', James proposes an index type, *Grid*, for building array-based generalised spreadsheets. We want to construct rules that converts the generalised spreadsheet to a data structure that can be represented in a familiar two dimensional spreadsheet layout, plus formatting. We will call this data structure a 'Spreadsheet', denoted **S**. **S** should have within it all the layout and formatting information that the backends need to produce spreadsheets. The problem is a little circular: **S** should be able to extract the layout from the index types, but in order to type *Grid* it would be helpful to know how the layout we wish to aim for. These notes are partly to help with my understanding of the problem, but hopefully also to help feed back into developing *Grid*.

2 *Why* do we need types

Statically typed languages can see into the future. They know what computations makes sense before they are executed. We need similar functionality to create spreadsheets. When faced with any given expression we should know how to represent its underlying structure, not simply evaluate the result. A typed language means that we are able to recover the structure of expressions, and hopefully represent this structure in the layout and formatting of the outputted spreadsheet. Furthermore, we can also be assured that we will not encounter expressions that have a structure we do not know about, so that our layout rules, once finalised, will be applicable to any expression they will encounter.

3 Grid - recap

A *grid*, $\mathfrak{G}$, is a totally ordered, finite set. An *array* is a map $\mathfrak{G} \to P$, where P is a set of all *primitives*. We write $\varnothing$ for the empty grid (which is unique) and $\bullet$ for a distinguished grid with a single element.

A direct sum of grids, $\bigoplus_i \mathfrak{G}_i$, is a an array of the length of all $\mathfrak{G}$ combined but the output array knows how it got there. Conceptually, every grid is a direct sum of the unit grid $\bullet$. For example, an array of type $\mathbf{3} \to P$ is a direct sum of three $\bullet \to P$. Out of convenience we write an array of type $\bullet \to P$ as a 'bare value,' such as `42`.

The direct product of grids is the Cartesian product with 'dictionary order' on the elements. We write the direct product of two grids, $\mathfrak{G}$ and $\mathfrak{H}$, as $\mathfrak{G} \otimes \mathfrak{H}$. The array `[[10 20] [30 40] [50 60]]` is of type $\mathbf{3} \to (\mathbf{2} \to P)$, which is equivalent to $\mathbf{3} \bigotimes \mathbf{2}$.

4 Spreadsheet notation

We keep in mind the difference between dimensions that have order, such as spatial layout and colour, and dimensions that are categorical, such as textual emphasis. We also note that as much as possible our formatting conventions should be generalisable to any output. Therefore, some dimensions that are not generalised to any output, for example colour, should be optional enhancements rather than a core way to communicate information.

For conciseness we will use the following notation to refer to spreadsheet representation. The final spreadsheet that is laid out and formatting we call $\mathbf{S}$. A double arrow $\Rightarrow$ represents the process of layout and formatting that produces $\mathbf{S}$. Everything to the left of $\Rightarrow$ is in the Cell domain, everything to the right of $\Rightarrow$ is in the $\mathbf{S}$ domain. The amount of cells covered is represented by a subscript 2D vector, such that 3 rows and 2 columns would be represented $\mathbf{S}_{[3,2]}$. For convenience a single cell is $\mathbf{S}_\bullet$ Spatial Layout is denoted by arrows $(\downarrow\rightarrow)$. Colour is given by κ. If there is an order that is signified from less emphasis to more emphasis (e.g. light to dark) by $\overrightarrow{\kappa}$. Textual emphasis are indicated as follows: t, $\mathbf{t}$, $\underline{\mathrm{t}}$. Borders, from dashed to solid to emphasised solid (could be double lines or bold): $\mathbin{\vdots}, |, \|$.

5 Direct sums and spreadsheets

In $\mathbf{S}$ we should be able to distinguish between `[10 20 30]` and `{10 [20 30]}`. The first is $\mathbf{1} \bigoplus \mathbf{1} \bigoplus \mathbf{1}$ and the second is $\mathbf{1} \bigoplus \mathbf{2}$.

An application is a sum type. For example the expression `{+ 10 [20 30]}` has type $\mathbf{1} \bigoplus \mathbf{1} \bigoplus \mathbf{2} \to P$. Note that the expression `{[+ +] [10 10] [20 30]}` has a different type of $\mathbf{2} \bigoplus \mathbf{2} \bigoplus \mathbf{2} \to P$. We may wish to represent these differently.

Not all sum types are equal. The expression `{10 10 [20 30]}`, of type $\mathbf{1} \bigoplus \mathbf{1} \bigoplus \mathbf{2}$ may want to be treated differently than `{+ 10 [20 30]}`, which is also of type $\mathbf{1} \bigoplus \mathbf{1} \bigoplus \mathbf{2}$.

For now we do not know how we want to represent applications. We first tackle the question:

How do we represent the structure of direct sums of grids when the primitives are bare values?.

Rule 1 states that all unit grids corresponds to a single spreadsheet cell. This seems uncontroversial for situations where the primitives are not applications.

$$\bullet \to P \Rightarrow \mathbf{S}_{\bullet} \tag{1}$$

Next we need a rule to represent nested sums. Our array-based sum types can be represented as a hierachical tree structure (see 'Grid Types'), where each grid of type $\bullet \to P$ is a terminal node (or leaf). Note that at each level of the hierarchy there is also a *horizontal* order, given by the grid index , meaning that our expressions are a sort of ordered tree. Fig 1 gives examples of various sum types that need to be principally converted into $\mathbf{S}$.

5.1 A tree searching algorithm

Before describing the formatting choices we make explicit our the expression tree is traversed. First, we start at the root node. At any point if the current node is the root node then the algorithm terminates. If the root node has child nodes, we count the length of the longest branch of each child, then order the children by the length, with longest branch first. We move to the first child, then map down to the first leaf of the longest branch. In Fig 1a this would be `10`, in Fig 1e this would be `50`. At a leaf node (*terminal node*) there are are two potential maps, to the next sibling node ($\overrightarrow{s}$), or to the parent node ($\uparrow_p$). Let $\overrightarrow{s}$ be the default map, with $\uparrow_p$ being executed only if there are no more sibling (child) nodes.

If the current node is a parent node then it may have one or more sibling nodes, and we may have entered the set of sibling nodes at any position. Note that in our grid type system parent nodes are either direct sums or products (we will leave products for now), so in Fig 1a the parent node is `[10 20 30]`. If the parent node does not have siblings, it will be the root

node, so the search will terminate. If current node has siblings, we map to the first sibling, ($\overleftarrow{s_1}$), which can be self-referential. If the new node has already been encountered, we map to the next sibling. If this sibling has child nodes we move to the deepest and leftmost leaf, $\swarrow_l$, and resume the left-right traverse as before.

Pseudocode notation:

- root, parent, leaf, child and sibling nodes as r, p, l, c, s respectively
- current node is α
- previously encountered nodes are in the list **e**.
- The set of parent nodes is **p**.
- Position in the horizontal ordering is given by subscript i, e.g. s_i
- map direction is given by arrows.
- The shorthand $\swarrow_l$ means to recursively map to the child node with the longest branch until there are no more children, then map to the first sibling (i.e. the deepest, leftmost leaf).
- *val* refers to recording the node's value

Algorithm:

1. Start at r.
2. If $r \in \mathbf{p}$, then $\mathbf{c} = [c_i, c_{i+1} \dots c_n]$, where $depth(c_i) > depth(c_{i+1})$
3. for c in **c** :
   (a) if $c \in \mathbf{p}$:
      i. do until $\alpha = c$:
         A. if $\alpha \in \mathbf{p}$: $\swarrow_l$
         B. *val*
         C. do $\overrightarrow{s}$, *val* until !$\exists$ s_{i+1}
         D. $\uparrow_p$.
         E. $\overleftarrow{s_1}$, do $\overrightarrow{s}$ until $\alpha \notin \mathbf{e}$
         F. repeat for s_is
      ii. continue
   (b) else, *val*, continue

5.2 Formatting the search

It should be clear from the formatting how the tree has been traversed. In the following description the algorithm steps are translated to formatting steps. Algorithm steps that do not need to be represented in the formatted are coloured grey. Formatting steps are coloured green for ease of reading.

1. Start at $r \Rightarrow$ new column (enter heading...)
2. If $r \in \mathbf{p}$, then $\mathbf{c} = [c_i, c_{i+1} \dots c_n]$, where $depth(c_i) > depth(c_{i+1})$
3. for c in $\mathbf{c}$:
   (a) if $c \in \mathbf{p}$: $\Rightarrow$ if it isn't the first child add an empty row
      i. do until $\alpha = c$:
         A. if $\alpha \in \mathbf{p} : \swarrow_l \Rightarrow$ add a dashed border
         B. $val \Rightarrow$ enter value in current cell
         C. do $\overrightarrow{s}, val$ until $!\exists\ s_{i+1} \Rightarrow$ for each sibling, move down a cell and enter value
         D. $\uparrow_p . \Rightarrow$ add a solid border
         E. $\overleftarrow{s_1}$, do $\overrightarrow{s}$ until $\alpha \notin \mathbf{e}$
         F. repeat for s_i
      ii. continue
   (b) else, val, continue $\Rightarrow$ enter value into current cell, move down a cell

In other words, the formatting rules, where x is a node and only leaf nodes return $value(x)$, are:

$$\bigoplus_i \mathfrak{G}_i \Rightarrow \text{a column} \tag{2}$$

$$value(x) \Rightarrow \text{cell value} \tag{3}$$

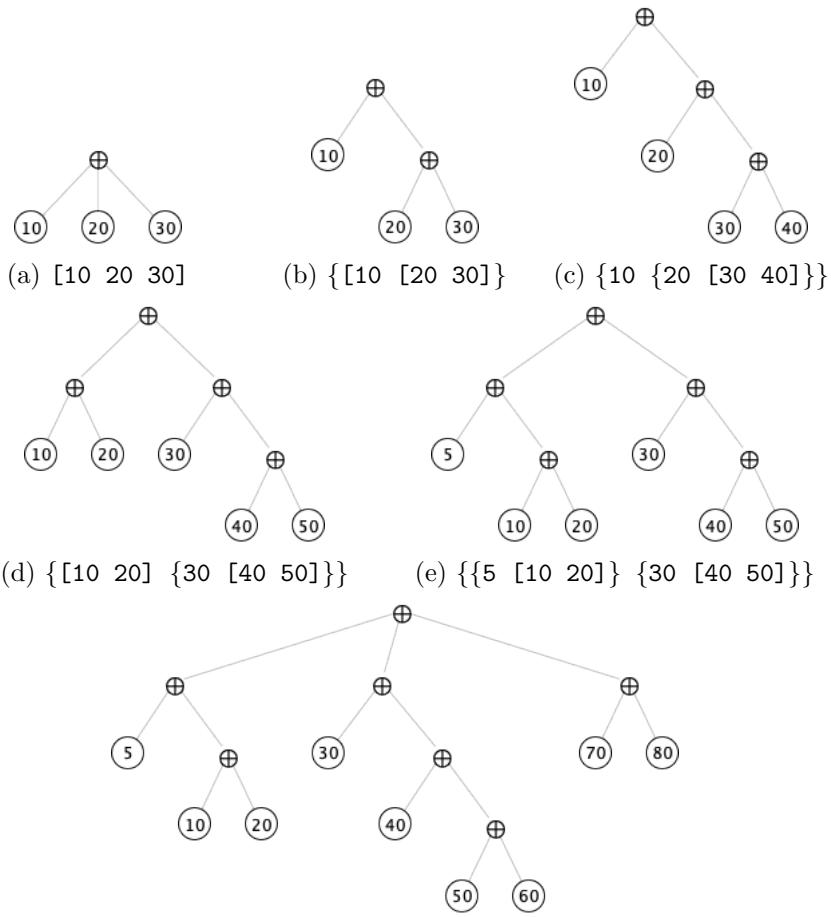
$$deepestleaf(x) \Rightarrow \text{dashed border} \tag{4}$$

$$sibling(x) \Rightarrow \text{no border} \tag{5}$$

$$parent(x) \Rightarrow \text{solid border} \tag{6}$$

$$branch?(x) \Rightarrow \text{empty row if not the first and has more than 1 level} \tag{7}$$

These rules would give the following formatting for the examples in Fig 1.



(a) `[10 20 30]` (b) `{[10 [20 30]}` (c) `{10 {20 [30 40]}}`

(d) `{[10 20] {30 [40 50]}}` (e) `{{5 [10 20]} {30 [40 50]}}`

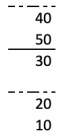
(f) `{{5 [10 20]} {30 {40 [50 60]}} [70 80]}`

Figure 1: Examples of different Grid sum type structures

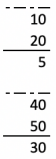
(a) `[10 20 30]`

(b) `{[10 [20 30]}`

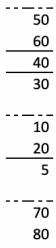
(c) `{10 {20 [30 40]}}`



(d) `{[10 20] {30 [40 50]}}`



(e) `{{5 [10 20]} {30 [40 50]}}`



(f) `{{5 [10 20]} {30 {40 [50 60]}} [70 80]}`

Figure 2: Examples of formatting for the Grid sum type structures shown in Fig 1