Typst Improved Tables - Planning

PgBiel <https://github.com/PgBiel>

2023-12-10 (Last update: 2023-12-26)

Contents

1 Introduction and initial details

This document is meant to coordinate efforts towards improved Typst tables. Right now, Typst's `table` elements aren't very flexible - you can't customize individual lines, merge cells, have repeated headers and/or footers, and so on. That's where the author's Tablex [1] came by, providing a pure Typst alternative to Typst's default tables, bringing many of those very important features - including advanced per-cell customization, customization of each line, merging cells vertically and horizontally, and so on. The intention of the efforts described in this document is to provide these features in the default Typst tables as well, through Rust code instead of Typst code.

1.1. Important definitions

- A **track** is a column or a row in the table.

2 Ideas

Below, I will list all ideas that have been brought up so far, in a pretty unordered and informal manner. Then, I will collect them in the "Requirements" section once we are sure on what's going to be done. There, the ideas will be more organized, labelled and prioritized.

2.1. Per-cell customization

1. **OK:** Use a `table.cell` element with customizable settings for each cell. It could look something like

```typst
#table(
  fill: green,
  table.cell(fill: red)[Hello world! This is red],
  [This is green!]
)
```

2. **OK:** It should have similar properties to tablex's implementation. In particular, regarding the most basic properties, `fill`, `align` and `inset` are desirable to override the default table setting. Additionally, a cell would have a `body` or `content` field to access its inner content.

> **Note 2.1.1**: Theoretically, it'd be possible to change cells' `x`, `y`, `colspan`, `rowspan` fields through `set` rules. That's not necessarily a problem for tables themselves, as the style chain at the grid generation stage of the table code would likely already have the final values of those fields, considering `set` rules and such. However, using such rules should be avoided, as they can produce unexpected consequences.

3. **OK (Proposal 5):** It should be possible to customize cells in bulk, akin to tablex's `map-cells`.
   - In particular, while `fill: (x, y) => pick-color(x, y)` and `align: (x, y) => pick-align(x, y)` work, they can't depend on other properties the cell has; in par-

ticular, the cell's body (e.g. you might want to set the fill for all cells with the text "3" to green).

- **Proposal 1:** We could change their signatures to `fill: cell => value` **(breaking change)**. The idea is that a `table.cell` would also have `x, y` fields to let you "locate" it, and a `body` or `content` field.
- **Proposal 2:** We could use show rules. However, they are currently limited due to problems in the styling system:

```typst
#show table.cell: it => {
  // This doesn't work
  set table.cell(fill: red)
  it
  // This doesn't work either (recursion, extra fields etc.)
  // Requires heaps of workarounds
  table.cell(..it.fields(), fill: red)
  // This also doesn't work
  it.fill = red
  // What do we do (in a sane way)?
}
```

  - **Investigation needed:** Additionally, how would tablex detect changes in those fields and apply them? Can an `impl Layout for TableElem` interact properly with the style-chain post-show rules?
    - **Temporary conclusion:** This is probably one of the main problems behind `show x: set x()`, and an alternative proposal to `set` cell properties would likely be ideal.
  - Still, this would probably be the most "Typst"-y way to customize cells, and **would be viable after a style system rework.**
- **Proposal 3:** Bring tablex's `map-cells: cell => cell` option to tables.
  - Would be the simplest option and the easiest to implement.
  - However, the interface **wouldn't be very consistent** with the rest of Typst.
- **Proposal 4 (Pg):** Have `table.cell` properties accept both simple values (e.g. `fill: red`, `align: left` etc.) *and* functions of the form `data => value` (e.g. `fill: (x, y, body) => (red, yellow, green, blue).at(x + y)` or `align: data => if data.body == [1] { left } else { right }`). This would allow us to circumvent the show rule problem by using set rules instead: `set table.cell(fill: data => ...)`. Show rules would thus be restricted to modifying the body of each cell.
  - **Which data to provide to the properties' functions?** While we could, in theory, provide a dictionary with all of the cell's fields, *would that be desirable?*
  - **Proposal 4A:** Provide only `(x, y, body)` to each function (perhaps in a dictionary for more flexibility).
  - **Proposal 4B:** Provide **all "static" cell properties** (`x`, `y`, `colspan`, `rowspan`, `body` and any others which wouldn't accept a function) in a dictionary to each function.
  - **Proposal 4C:** Provide **all cell properties** (copied to a dictionary before the function is called, which then is provided to it) to each property.
    - This would raise questions regarding which property's function would be called first, etc. So you can't reliably have align depend on fill and fill depend on align. But such a case could perhaps just be considered pathological.

        - This would be the most flexible option, but 4B is also more balanced in the sense that there is no margin for such “race conditions”.
    - **Proposal 5 (Laurenz + Pg):** Allowing cell properties to depend on cell body inspection is probably not a good idea (we shouldn’t encourage content inspection). So, for now, perhaps per-cell properties should just be simple (`fill: red`, `align: left` etc.), and anything depending on content should be done with show rules.
    - In addition, we could allow cells to have `fill: auto` (or `fill: data => auto`) and similar to just use the global setting.

4. **To be discussed:** It should be possible to place cells arbitrarily in the table by setting their `x, y` positions manually.

Note 2.1.2: Throughout the document, we will assume that cell coordinates are **zero-indexed**, that is, the cell with `x: 0, y: 3` is in the **first** column (left-to-right) and **fourth** row (top-to-bottom). Note also that **ordinal numbers** (first, second, …) **are one-indexed** to follow what the English language mandates, but sometimes we will mix both coordinates and ordinal numbers, so be aware of that (the ordinal numbers are always the coordinates plus one).

- For instance, `table.cell(x: 5, y: 2)` would place it at the 6th column, 3rd row.
    - Cells which would normally be automatically placed at such a position would just skip it.
- If we follow tablex’s system for this, we would have those properties default to `auto`, meaning they would be, by default, automatically positioned based on where they were specified in the table’s parameters.
- However, **what if one of the coordinates is omitted** (e.g. `table.cell(x: 5)` or `table.cell(y: 3)`)? The other one would likely be `auto` as well, but then how would we calculate it?
    - **Proposal 1 (Laurenz[1]):** We could use the first available position in the selected track. For instance, if we specify `table.cell(y: 3)`, and the 4th row only has a single cell to the left, then we would pick `x: 1` as that’s the first free position in the row. Similarly, for `table.cell(x: 3)`, we would pick the first available position, from bottom to top, in the fourth column.
        - Likely more expected in general.
    - **Proposal 2 (Tablex’s approach):** The missing coordinate should be determined without change from the `table.cell(x: auto, y: auto)` case. That is, if the cell `table.cell(y: 3)` is specified right after `table.cell(x: 1, y: 1)`, then the former cell’s missing `x` coordinate will be calculated to be `x: 2` (the previous plus one). With a missing `y` coordinate, the cell would stay in the same row as the previous cell unless the row is entirely filled, in which case the cell would go to the next row.
        - It works, but might be a bit surprising. Would be nice to have more opinions here.

5. **OK (Proposal 1):** Which `table.cell` properties should be applied *before* its show rule, and which should be applied *after*?
    - That is, **what is truly overridable** through show rules? Only the cell’s body? Its alignment and inset? All of it?
    - **Proposal 1:** Alignment and inset are applied *before* show rules (thus **overridable**), but fill is applied *after* show rules (**not overridable**, other than by covering it).

    - This is consistent with how tables are currently implemented (alignment and inset are applied to cells' bodies before they are laid out, while fill is applied after all cells were already laid out).
    - This allows completely overriding inset, which may be an advantage in terms of flexibility as it allows a package such as 3rd-party `diagbox` [2] to create custom cells which completely occupy a cell's available space, without regard for inset (one of the main annoyances when using said diagbox package, which requires workarounds).
    - However, this places a burden on the show rule user, which must write `#align(cell.align, block(inset: cell.inset, cell.body))` at the end of every show rule to ensure the original properties will be kept.
    - **Investigation needed:** A possible compromise might be to fall back to the table's global settings for `align` and `inset` when the cell's settings are ignored in a show rule, but that seems difficult to implement.
  - **Proposal 2 (Tablex, sort of):** Allow alignment, inset and fill to be overridden on show rules.
    - Tablex's `map-cells` follows this idea, as just returning content will discard all of those properties, while you can modify one or another property in the incoming `cellx` object and just return that modified `cellx` object to keep only the properties you didn't modify. That is, you can change any aspect of the cell in that manner.
    - This is the option which provides the **maximum flexibility**, and, in theory, would **have some consistency with other Typst "sub-elements"**, such as **`outline.entry`** (which can be completely overridden) - however, `outline.entry` cannot have its indentation overridden, for instance, **which could be an argument against Proposals 1 and 2**.
    - Additionally, the burden on the show rule user is even higher now, as not only will you have to keep alignment and inset through above, but also `fill` with `block(fill: cell.fill, ...)`.
  - **Proposal 3:** All properties applied *after* show rules.
    - **Minimum flexibility.** The user cannot modify alignment, inset or fill through show rules.
    - However, it's **more practical** when the user just wants to override the cell's body based on e.g. its coordinates, which would be available along with other fields.

6. **To be discussed:** There should be a built-in method to create a `diagbox`-style cell, with a divider across its diagonal separating two bits of text / content.
  - While this can be done with a third-party package such as diagbox [2], it requires manual adjustment to ensure the diagbox fits the entire cell, as tables have an `inset` parameter (which defaults to `5pt` on all sides, and can even change for each different side), and knowing and specifying the table inset is required to have the diagbox expand enough to "bypass" inset.
    - This can only be done transparently when the table itself renders the diagbox, as it then provides the correct `inset` parameter.
    - The solution is to have a built-in diagbox element or similar.
  - **Proposal 1:** Have a separate diagbox element, e.g. `table.diagbox(flipped: true)[a][b]`.
    - This could translate to internal cell properties, or even just be wrapped as the body of a `table.cell`.
  - **Proposal 2:** Use properties on `table.cell`, e.g. `table.cell(diagbox: (left: [a], right: [b], flipped: true))`.

7. **To be discussed:** Cell customization could be made powerful enough to allow for spreadsheet-like powers, where one cell's content, fill and other properties depend on other cells' properties (especially content). For instance, the rightmost cells of each row could be the sum of the numbers displayed in the other cells in the row (you'd, in that case, parse its content with `int(cell.body.text)` or similar).
   - Tablex partially allows for this capability, in a restricted manner, using `map-cols` and `map-rows`: you can modify rows and columns of the table grid in bulk, thus allowing one cell's content and properties to depend on properties from other cells in the same row and in the same column as itself. The first example in tablex's README showcases this, where a cell's text is even colored differently depending on its own body (which is a number calculated as the sum of other cells in the row).
   - While a thin wrapper over table (which modifies `..args` before they are passed to `table()`) can simulate this behavior, it is worth noting that a wrapper over table doesn't have access to the table's final grid, as cells can be specified in any order (when we consider that we will allow specifying arbitrary coordinates for cells). The table wrapper would have to position cells in the grid by itself, so it would end up not being very "thin", hence why native support could be helpful here.
   - **Proposal 1 (Tablex's approach):** Add a way to modify rows and columns in bulk, like tablex's `map-cols` and `map-rows`.
     - This isn't powerful enough to allow a cell depend on *any other cell*, but is good enough for a good amount of usecases and less complex to implement.
     - One would need to decide the order in which those modifications are applied to avoid ambiguities (columns or rows first? Show rule last?).
   - **Proposal 2:** Allow a variant of `map-cells` which provides the entire grid for each cell which needs it to determine its content and properties.
     - This would be extremely powerful, but we would have to consider whether the potential extra complexity - both in table parameters and in the internal code - is worth it.
     - **Investigation needed:** Perhaps the grid passed should only be a copy of the initial grid in order to avoid unpredictable interdependence of cells. This could also make the operation faster by potentially allowing us to rely on COW mechanics (if the cell doesn't modify the grid at all, then it isn't copied, which can be expensive if it is too large).
   - **Proposal 3:** Move such behavior to a separate element, e.g. `spreadsheet`, with otherwise similar API to `table`.
     - Could be overkill.

2.2. Merging cells

1. **OK:** It should be possible to merge cells horizontally (colspan) and vertically (rowspan).
2. **OK:** This should be a property of `table.cell` elements, should we create them.
3. **OK:** This should likely be implemented as in tablex: when merging cells, one cell is kept as its "parent" in the grid, and other cells participating in the merge are kept as "child" cells which only point back to the parent. Thus, when analyzing which positions are available

[1]See discussion on Discord: https://discord.com/channels/1054443721975922748/1117839829191901286/1118120012968894475

in the grid (in order to position a cell with automatic positioning), you'd just skip child cells as well as normal cells.

4. **To be discussed:** Should we provide tablex-like helper functions, such as `table.colspan(2)[body]` (equivalent to `table.cell(colspan: 2)[body]`) and `table.rowspan(3)[body]` (equivalent to `table.cell(rowspan: 2)[body]`)? Would we also accept `table.colspan(2, table.rowspan(3)[body])`?
   - **Proposal 1:** Those helper functions could be added and function similar to `text`: they would just apply `set table.cell(colspan: ...)` and `set table.cell(rowspan: ...)` rules. That would also allow nesting them. Their usage is optional.
   - **Proposal 2:** The helper functions wouldn't be added, and you'd always construct `table.cell(colspan: ..., rowspan: ...)[Hi]`.

5. **To be discussed:** When colspans span auto columns and rowspans span auto rows, how should those columns and rows expand?
   - **Proposal 1 (Tablex):** Always expand the auto column / row with the largest index (i.e. the rightmost auto column and the bottommost auto row), as needed.
   - **Proposal 2:** Perhaps consider expanding auto columns / rows evenly.

6. **To be discussed:** When a merged cell spans both one or more `auto` tracks and one or more `1fr` (/fractional in general) tracks, what should happen to the `auto` tracks?
   - Currently, both in tablex and (seemingly) native tables, `auto` track sizes are calculated *before* fractional track sizes. This is necessary because fractional track sizes can only occupy the available space remaining in the page width / height after other non-fractional columns are laid out.
   - However, with merged cells, we'd have cells depending on both fractional and non-fractional tracks. On tablex, this leads to a few visual bugs [3], [4]. In particular, for all intents and purposes, a fractional track is considered to have size 0 before its effective length is calculated, so auto tracks expand to fit the entire merged cell when, in reality, the fractional track could have expanded enough to fit the merged cell.
   - So the question is, how to best sidestep this problem?
     - **Proposal 1:** If a merged cell spans all fractional tracks in the table, it should **not** cause `auto` tracks to expand. This is enough to tackle [3].
       - This is based on the idea that, by spanning all fractional columns, a colspan cell will have **access to all available space** within the page/container width. So, even if its `auto` columns expanded, they would never expand, together, more than that fractional column, because **`auto` columns are resized to fit in the available page width** after being calculated.
       - However, it might be best to **restrict this behavior to colspans** (and not rowspans), at least in row-major tables (the default), as **`auto` rows are not resized to fit the page height** - either they are fully sent to the next page (tablex), or they cross both the current page and the next (native tables), while **fractional rows' sizes only consider the current page.**
     - **Proposal 2:** Some form of multi-pass algorithm could be employed: calculate `auto` columns, calculate fractional tracks, and check mathematically if fractional tracks would have been enough to fit merged cells; if so, repeat the process (perhaps at most once).

        - **Investigation needed:** This could use some more formal definition of algorithm and/or the calculations required.
    - **Proposal 3 (Tabularray [5, p. 27]):** Allow choosing whether or not a particular merged cell (or all such cells) can cause an `auto` column to expand (in our case, the last `auto` column), or if column sizes should ignore that (or all) merged cell (and thus only be calculated using non-merged cells / other merged cells).
        - This would then allow the table user to choose what looks best for their needs, and thus sidestep this problem in a way that potentially pleases more users.

7. **To be discussed:** Which algorithm should be used to **break down table lines** such that they **do not cross merged cells**?
    - **Proposal 1 (Tablex 0.0.2 [6]):** Break down lines such that each occupies the border of a cell.
        - This is likely the most sane approach, but the problem is that this forces lines to occupy the *entire* border of each cell, leading to issues such as [7].
        - Doing this in a greedy way like tablex also leads to issues such as [8], where one line may be broken down into two consecutive lines for no reason, leading to dashed patterns and gradients in lines' strokes being "restarted" in the middle of the line (which is actually two visually joined lines) for no reason, making the patterns look "broken" / "uneven".
            - A way around this is to set the `phase` for each dashed stroke and `relative: "parent"` for each gradient / pattern stroke.
            - In parallel, however, some other less greedy algorithm could be used - either via backtracking or something else - which **merges adjacent lines**.
    - **Proposal 2:** In order to solve line merging, perhaps instead of analyzing every cell and seeing which lines could be their borders, we could do the opposite: take each line and keep advancing it until either a merged cell is reached (in which case a new line is formed, and it continues after that merged cell if possible) or a gutter is hit (and the line is configured to not cross gutters - not currently possible in native tables, but possible in tablex).
        - This would have some problems. How would we know when we hit a merged cell? Perhaps there could be a cell from 50 rows above us with a rowspan of 51 which could conflict with a horizontal line somewhere. We'd, thus, need to check every single cell before the current position every time, and there'd likely be more cells than lines, thus possibly making this algorithm less efficient.
        - One way to speed up this algorithm considerably could be to keep a separate list of cells with colspan ≠ 1 or rowspan ≠ 1. There would, thus, be no lookup at all when there are no merged cells, and no lines would be broken down.

2.3. Line customization

1. **OK:** It should be possible to control the appearance of every single line in the table. Among the fundamental customization properties, you have the `stroke` of the line.
    - This should allow removing all vertical lines, for instance, or all horizontal lines easily.

2. **Under discussion:** How to specify these lines?
    - **Proposal 1**: Perhaps use some sort of `lines: (list, of, lines, ...)` or `lines: (x, y) => line-properties` property on the table, similar to `fill` and `align`.

- **Proposal 2 (Tablex's approach):** Rely fully on special `table.hline` and `table.vline` elements which are placed alongside cells in the table. This would allow for some degree of automatic positioning (e.g. place an `hline` under a row by just creating it right next to the cells of the desired row).
- **Proposal 3 (Laurenz):** Allow `stroke` to take a function with cells as input where you'd specify the stroke for each border of the cell.
- **Proposal 4:** Perhaps allow multiple of the specification methods above simultaneously.

3. **To be discussed:** Should we use special `table.hline` and `table.vline` elements for table lines? Should we just use built-in `line` elements instead somehow?
   - Noting that reusing the `line` element could perhaps restrict the available customization options, or not integrate too well with the table's coordinate system, which might make such an option less viable.
4. **Under discussion:** When a cell spans a pagebreak, which lines should appear right before the pagebreak, and which lines should appear right after?
   - Currently, all lines are identical, so this doesn't matter.
   - **Proposal 1 (Pg):** Lines right before the pagebreak (in the current page) should be copies of the lines which come right under the cell. Lines right after the pagebreak (in the new page) should be copies of the lines which come right above the cell.
5. **To be discussed:** Should it be possible to specify not only vertical and horizontal, but also diagonal lines?
   - At the moment, this sounds very far-fetched, especially since handling diagonal lines going through pagebreaks would be a bit hairy, and require some math.
   - The idea can be kept here for the future though. For the moment, we should focus on horizontal and vertical lines.
6. **To be discussed:** How to ensure line customization interacts properly with merged cells?

2.4. Grid and table unification

1. **OK:** `grid` and `table` should be much closer to each other in terms of available settings. Maybe even have the same API!
   - **Initial idea:** You'd have, for instance, a `grid.cell` element. However, **that'd be different from** `table.cell`. Show rules applying to one shouldn't apply to the other.
   - **Initial idea:** Similarly to tablex, the main difference between the two - other than the semantical difference - would be that a `grid` has `stroke: none` (or, rather, no lines at all) by default, while `table` has all lines (horizontal and vertical) by default.
   - **Investigation needed:** We will probably need to have some sort of "Cell-like" trait so that both a `GridCellElem` and a `TableCellElem` can be specified for the `GridLayouter`.
     - **Conclusion:** We will use a `Cell` trait, with `Layout` as a supertrait.
   - **Investigation needed:** How would this affect other elements which depend on `GridLayouter`, such as `list`, `enum` and the like?
     - **Conclusion:** They will use `Content` as the cell type (GridLayouter is generic over `T: Cell`).
2. **OK:** `grid` and `table` should have the same fields available for customization. The main difference would be that `grid` would default to having no lines or inset at all (`stroke: none` and `inset: 0pt`), while table would keep its `stroke` and `inset: 5pt` defaults.

3. **OK:** `grid.cell` should be made available with the same properties and behavior as `table.cell`, but shouldn't be affected by `table.cell` show rules and be distinct.

2.5. Repeatable headers

1. **OK:** It should be possible to specify a set of rows as the **header rows** of a table. These header rows **would be repeated across pages**, that is, every time the table is broken across pages, the header's cells would be repeated right at the top.
2. **To be discussed:** What's the best way to specify the header rows of a table?
   - **Proposal 1 (Tablex approach):** Specify `header-rows: n`, and the first `n` rows will be considered the header rows of the table. Thus, the cells in the header are integrated with the rest (their coordinates indicate they belong to the first `n` rows), but their laid out contents are repeated on each pagebreak.
   - **Proposal 2:** Specify `header: (list, of, cells, ...)` separately, as an option to the table. This would make it less clear whether the cells would be integrated with the table and have proper coordinates.
     - Without proper coordinates, we would have to rethink how `table.cell.x` and `table.cell.y` work.
     - **Proposal 2A:** `table.cell.x = table.cell.y = none` for cells in the header.
       - Would likely make it weird for show rules and whatnot.
     - **Proposal 2B:** The coordinates of cells in the header would be relative to the top left cell in the header (the "relative" `(0, 0)` cell), but would be detached to the coordinates in the table; that is, there would be two cells in the table with coordinates `(0, 0)` (and possibly others with the same coordinates).
       - Could be pretty awkward to work with in general.
     - **Proposal 2C:** Automatically integrate header cells with the rest of the table, such that the top left cell of the header is also the top left cell of the whole table, and thus coordinates are automatically adjusted.
       - While this seems more sensible, this would potentially interfere with arbitrary cell positioning.
   - **Proposal 3:** A mix of proposals 1 and 2 (perhaps tending more towards 2C), we could specify a `table.header` element above cells (but among them), making it clearer that it could affect their coordinates.
     - Maybe a bit overkill?

2.6. Repeatable footers and other kinds of repetition

1. **To be discussed:** There could be a feature similar to repeatable headers, but repeating footer rows across pages.
   - The same concerns regarding coordinates from headers would remain: use the last n rows as the coordinates? Have independent coordinates from the rest of the table (top left cell of the footer is `(0, 0)`)?
   - Should the API be identical to repeatable header's? Have `footer-rows: n`? `footer: (some, cells, ...)`? `table.footer(...)`?
2. **To be discussed:** There could be a way to repeat **table captions** (e.g. Table 1 appears on the first page, Continuation of Table 1 appears on the second, ...)

- This, however, would likely be **better suited for figure**, as it's the figure element which provides the table's caption. In general, figures could be adapted to repeat captions across pages for anything they main contain, not only tables.
- This could, therefore, end up being out of scope for work on tables, although such work on figures would be incentivized and/or welcome.

2.7. Table direction and orientation

1. **Under discussion:** There could be a way to change the orientation of a table, such that it **expands to the right** instead of to the bottom. This means that **the rows will be fixed,** while **the amount of columns can expand** indefinitely. Basically, switches the roles of columns and rows.
   - For instance, you could specify `rows: (auto, 2pt)` to have two rows with those sizes, and `columns: (1em, auto)` to have one `1em` column and unlimited `auto` columns afterwards.
   - The cells are specified in **column-major order** - that is, the first cell specified to `table` will be in the first row of the first column; the second cell, in the second row of the first column; and so on, until the first column is fully filled.
     - Currently, all tables are in **row-major order** - cells fill rows instead of columns.
   - **Proposal 1:** Use a `transpose: true` argument to enable this behavior. Not only would this have cells end up being specified in column-major order (as this would swap rows and columns), but this would also swap (due to semantics of matrix transpose) the `rows` and `columns` parameters, such that the sizes of rows would correspond to the sizes of columns and vice-versa.
     - Can be confusing.
   - **Proposal 2:** Use a `major: "row" / "column"` argument (or `column-major: true` or similar) to specify if the table is row-major (default) or column-major. Enabling column-major order **does not swap the `rows` and `columns` parameters** (due to different semantics from the previous proposal), but it does affect the order in which cells are specified (increasing the `y` coordinate).
2. **To be discussed:** If the table is changed to be column-major (through either of the proposals above), then when the table grows to the right and reaches the page width, **the table wraps** - not necessarily into a new page, but into a new “subtable” below it with the same rows, but with the post-wrap columns.

2.8. Misc

1. Perhaps it should be possible to specify any kind of input as table cells. E.g., it could be possible to write

```typst
#table([a], 2, 3, -1.5, table.cell(...), $ 5 + x $)
```

   - Currently, the above errors as `int` and `float` aren't `content`. Ideally, we'd just convert them to content automatically with `repr` or similar.
   - **Investigation needed:** should we implement the above by ourselves by taking arbitrary `Value`s and displaying them if they're `Content`, using `Repr` otherwise? Or should we postpone this to the Type Rework, with which Content will be reworked (and thus, in theory, all types could become “showable”)?

2. It could be interesting to be able to specify a fixed number of rows (by specifying only the `rows` parameter, but not `columns`) and have the amount of columns auto-adjust. **You'd still specify cells in the natural ("row-major") order.**
   - Consider the example below:

     ```typst
     #table(
       rows: 3,
       [a], [b], [c]
       [c], [d], [e]
       [e], [f]
     )
     ```

     - With that setup, there would be exactly three rows, and the amount of columns would be calculated with `ceil(num cells / rows)`; in this case, there are 8 cells, so $\lceil 8/3 \rceil$ would result in 3 columns.
     - Otherwise, the table would **behave exactly as if given `columns: 3`**.
     - This is particularly simple to implement.

3 Requirements

Formalized and consolidated ideas. **This section is WIP**.

The requirement labels have some prefixes. "F" indicates a functional requirement (related to adding functionality), while "NF" is non-functional (specifying some desired characteristic, or something more general). This allows us to refer to those requirements with precision in discussions. Additionally, **PR:** indicates the pull request which implements the associated requirement.

3.1. Per-cell customization

[FPC1] We should create `grid.cell` and `table.cell` elements, which will contain settings customizing the cell's appearance and other properties. The former only works for `grid` cells, while the latter, only for `table` cells. **PR:** [9]

[FPC2] It should be possible to fully customize the appearance of table cells through show rules. **PR:** [9]

[FPC3] It should be possible to use `table.cell` in place of content in a `table()` call (and similarly for `grid`) to customize the look and properties of a particular cell. For instance, one could do `table([a], table.cell(fill: red)[b])` to override the fill for that particular cell. **PR:** [9]

[FPC4] `table.cell` should have the following **initial** properties (which change with the following proposals): `fill` (any color-like type), `align` (any `alignment`), `inset` (anything similar to table's `inset`). **PR:** [9]

[FPC5] When applying show rules on `grid.cell` and `table.cell`, their final properties must already be known. That is, `#show grid.cell: it => (it.fill, it.align, it.inset)` will contain the cell's final properties, regardless of whether they were customized or not. **PR:** [9]

3.2. Merging cells

[FMC1] It should be possible to **merge cells horizontally**, through a mechanism called `colspan`.

[FMC2] It should be possible to **merge cells vertically**, through a mechanism called `rowspan`.

3.3. Grid and table unification

[FGTU1] Grid should have all properties of table available. **PR:** [10]

[FGTU2] Grid should default to having no lines or inset (i.e. `stroke: none` and `inset: 0pt`), while table keeps its default `stroke` (all lines shown) and `inset` (`5pt`). **PR:** [10]

[FGTU3] Regarding the `table.cell` proposal, there would be a separate `grid.cell` element, with the same properties as `table.cell` but not affected by its show/set rules. **PR:** [9]

4 Increments / Waves

The idea here is to assign features to separate "waves" of table features.

4.1. MVP

- **ETA:** 3-5 weeks
- **Features:**
  - Grid and table cell unification
  - `table.cell`
  - Merging cells

4.2. First increment

- **ETA:** TBD
- **Features:**
  - Line customization
  - Repeatable headers
  - Repeatable footers?

4.3. Second increment

- **ETA:** TBD
- TODO

Bibliography

[1] PgBiel, “Tablex”. [Online]. Available: https://github.com/PgBiel/typst-tablex

[2] PgBiel, “typst-diagbox”. [Online]. Available: https://github.com/PgBiel/typst-diagbox

[3] PgBiel, “Tablex Issue #56: Colspan over 1fr prompting auto columns to needlessly expand”. [Online]. Available: https://github.com/PgBiel/typst-tablex/issues/56

[4] @manveru, “Tablex Issue #78: Column sizes with mixed fractional and auto sizes don't fit the page”. [Online]. Available: https://github.com/PgBiel/typst-tablex/issues/78

[5] Jianrui Lyu, “tabularray - Documentation”. [Online]. Available: https://mirrors.ctan.org/macros/latex/contrib/tabularray/tabularray.pdf

[6] PgBiel, “Tablex Commit 875ebaf: switch to drawing lines per whole cells”. [Online]. Available: https://github.com/PgBiel/typst-tablex/commit/875ebaf220a62d1cebe8db4d97636a69f8f53108

[7] PgBiel, “Tablex Issue #27: H/vlines ending in the middle of col/rowspan not drawn”. [Online]. Available: https://github.com/PgBiel/typst-tablex/issues/27

[8] @Enivex, “Tablex Issue #81: dashed and dotted hlines have uneven spacing”. [Online]. Available: https://github.com/PgBiel/typst-tablex/issues/81

[9] PgBiel, “Typst PR #3037: Initial table per-cell customization [More Flexible Tables Pt. 2a]”. [Online]. Available: https://github.com/typst/typst/pull/3037

[10] PgBiel, “Typst PR #3009: Grid and Table API Unification [More Flexible Tables Pt.1]”. [Online]. Available: https://github.com/typst/typst/pull/3009

[11] Jianrui Lyu, “tabularray (LaTeX)”. [Online]. Available: https://ctan.org/pkg/tabularray