A Tool for Visualizing Patterns of Spreadsheet Function Combinations

Justin A. Middleton
North Carolina State University
Raleigh, North Carolina

Abstract—**Spreadsheet environments often come equipped with a plethora of functions to manipulate and calculate data, but it can be difficult to understand how end-users employ these functions in practice. Without this knowledge, both researchers and practitioners lack information about how end users construct sophisticated programs from these basic elements. We developed a tool that visualizes patterns of how functions are combined into formulae within Excel spreadsheets. Using the Enron spreadsheet dataset as an example, this paper shows how the tool can display both common and anomalous formulas and their respective contexts in an actual workbook.**

I. Introduction

Business and research alike owe their debts to spreadsheets, the table-based interface which empowers users to organize and manipulate huge bodies of data [citation for definition]. Their allure is in their versatility: while the novice end user can work without a deep knowledge of programming, the expert can expedite their work with a variety of built-in operations, or functions.[1] As such, it should be of little surprise when Scaffidi and colleagues estimated that by 2012, over 50 million U.S. workers could be using them, including the 25 million who would be writing programs out of the functions included [1].

Considering this ubiquity, it's crucial to get spreadsheets right. Our failures in their use can be ruinous, as in 2012 when the influential findings on economic growth were reversed by a selection error, among other horror stories.[2]

Fortunately, the vanguards of spreadsheet research have assembled, organized, and released a number of spreadsheet corpora to inform work on how people actually use these tools in different contexts. Collections like EUSES [2], FUSE [3], and the Enron corpus [4] have already enabled fruitful work across the field, such as detecting code smells in spreadsheets [5] [6].

A tool, then, which empowers users of many intentions to explore these datasets would be rife with potential. On a simple level, the tool could provide concrete statistics on which functions are used in practice most (or least) often, and what other functions that work in conjunction with them. These numbers, then, could inform future work done on intelligently recommending functions to users or generating new formulas from these patterns of function use. Spreadsheet APIs, likewise, could be augmented or pruned through the discovery of frequent combinations or anomalous use of functions. Furthermore, if the tool maintains the connection between patterns and the actual, it could serve as a boon to educators as well, guiding lesson plans to the most commonly employed functions and offering a bounty of instructive (and real) examples.

This paper presents such a tool, informed by such varied sources, that visualizes not how people employ individual Excel functions but how they combine to make more sophisticated spreadsheet formulas.

II. Related Work

This tool comes from a line of spreadsheet visualization tools before it, each with a different focus. Some tools, such as Igarashi and colleague's fluid visualizations [7], seek to visualize the hidden formulas within a spreadsheet by imposing the dataflow graphs over the cells. Likewise, Clermont [8] (and Hipfl [9], who extended his work) explored various ways of visualizing groups of related cells through similar functions, neighbors, or references. Others focus on creating visualizations outside of the sheets: Hermans and colleagues address a spreadsheet programmer's information needs through their work on the tools GyroSAT [10] and Breviz [11] to make dataflow diagrams of individual spreadsheets. These, however, tend to focus on individual spreadsheets and not bodies of them.

Nevertheless, other studies and tools focus more on evaluating the content of the cells therein. (Hermans, Smellsheet)

Other studies focus on API and built-in function use outside the domain of spreadsheets. [RESEARCH ON API STUDIES]

III. Approach

A. Goals

In visualizing the spreadsheet data, I outlined a few core goals for what the tool should accomplish:

1 **Draw an interactive interface to explore observed function combinations.** The combination space for all Excel functions is massive, let alone the space for observed formulas. Working from data with actual referents in practice, the tool must aid the user in navigating this space.

2 **Emphasize the quantitative patterns in formula construction.** The tool, accommodating datasets of a few spreadsheets to millions, should address the questions of how often the end users employed a certain function and

[1]https://support.office.com/en-us/article/Excel-functions-alphabetical-b3944572-255d-4efb-bb96-c6d90033e188

[2]http://www.eusprig.org/horror-stories.htm

where they used it. In this way, it must show precise metrics, such as frequency of use and depth of function nesting, of the dataset it conveys.

3 **Promote a qualitative understanding of the patterns.** Lest these patterns remain abstract, the tool should supplement its observations with concrete instances of relevant formulas from the corpus. Ideally, it should even direct users to the exact cell in the spreadsheets where the formula was used, contextualizing the functions.

Likewise, to bound our scope, we outlined a few goals to specifically avoid accomplishing with this:

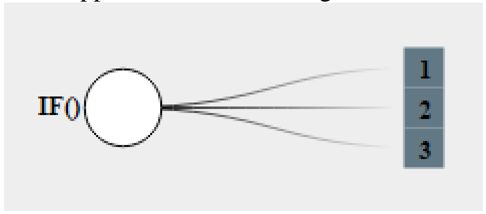
!1 **Do not try to directly explain what a function does.** Though the tool tries to foster tries understanding by linking pattern to example, it won't provide a precise description of what a function accomplishes. The tool's user must infer this.

!2 **Do not create new formulas.** This is essentially an exploratory tool, not a generative one. It should not produce any information other than new views of the original data.

!3 **Do not visualize individual formulas.** Though there is room to explore the place of a single function in the group, the tool must design the core visualization around the entire body of data, not the other way around.

B. Description

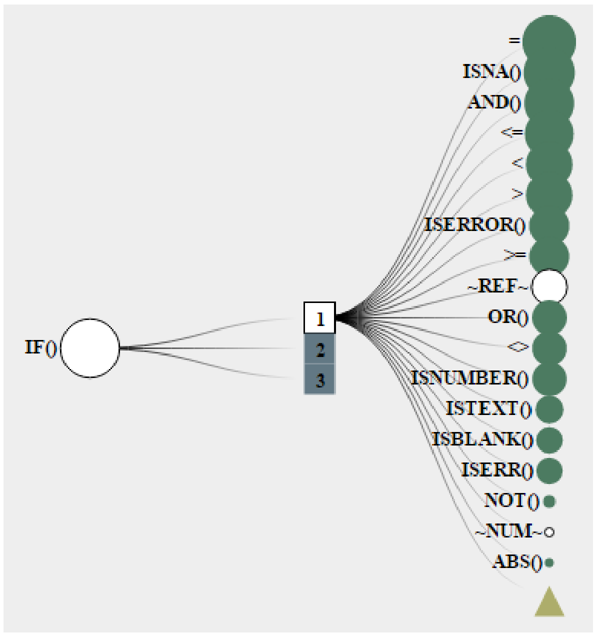
To show how this all came together and demonstrate the interface, we will use an example of a quick task: discovering what kind of conditions people use in IF statements. Considering only formulas in which the IF function itself is the top-level function (that is, not nested within another function), we will approach a tree that begins like this:



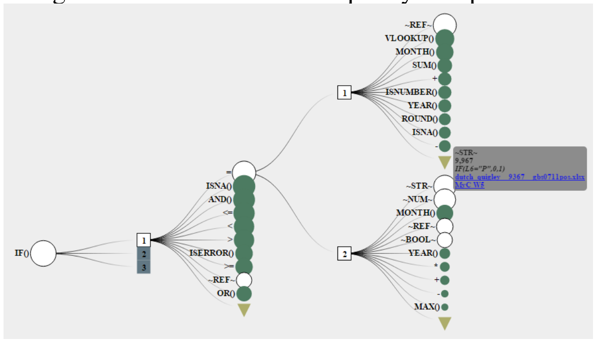
The visualization so far comprises two types of node: the circle, which represents a discrete function in the formula and is size according to its relative frequency (the top-level node, by definition, will be the largest); and the numbered squares, which represent the positions of arguments within its parent function. From this, we can infer that, of the times it was observed, IF can have at most three arguments passed into it, which corresponds with its specifications in the API.[3]

Knowing that it is the first argument which contains the conditionals, we click on the square labeled "1" to explore. To save space, when more than 10 unique arguments have been observed in any position, the tool displays only the first ten, with an option to display the rest. The results are shown below:

[3]https://support.office.com/en-us/article/IF-function-69aed7c9-4e8a-4755-a9bc-aa8bbff73be2



As expected, we see that IF contains as its first argument a number of comparison operators, such as = and ¡=, and boolean-returning functions, like ISNA and AND, with simple equality being the most common and some use as ABS being the least seen of everything actually used. From here, we can further explore the common options among these functions. Clicking on the "=" node will yield two arguments, it being a binary operator, and expanding each of them will peer into the range of common values of equality comparison:



For both sides of the equal sign, the operator has certain types of arguments which predominate over the others: on the left side is most often a reference to another cell; on the right, a string or number literal, which makes sense for the case of confirming a value in another cell before assigning this one. Furthermore, if the concept is difficult to imagine is practice, a tooltip accompanies each function node in the tree, providing a concrete example of a function that uses this structure and where it can be found.

C. Design

Early in the process, we decided that the tree form would be a suitable fit to represent the parent-child or caller-callee relationships inherent in the data, given the composition of formulas as functions and their arguments (which could be yet more functions). By adapting this structure to accommodate a broad range of possibilities for nested functions, the branching factor depends on both the number of arguments in a function and the number of possible functions observed as an argument in a function.

Guided by the goals in section A, we faced a number of decision points, which we describe below, before we arrived at the design shown above.

1a *Copied formulas*: Excel allows users to spread a formula over an area, repeating the same task in each cell with minor adjustments. Without checking for this, the analysis may not reveal the functions most commonly used together but rather the formulas most often applied to large areas. To combat this, we converted formulas from their native A1 format to the relative R1C1, in which copied formulas should be identical, and reduced the records to unique R1C1 formulas per sheet.

1b *Optional arguments*: How should the tool handle functions which accept a variable number of arguments? SUM, for example, can have anywhere from 1 to 255, and IF can accept either 2 or 3. [4] Without prior evidence, it's possible that spreadsheet programmers use different techniques for different numbers arguments. To account for this, we separate and analyze the different quantities of arguments observed for each function; the tool, however, uses as default all of the options collapsed into a single representation.

2a *Importance of depth*: When a function appears within another, should it be analyzed only as a nested function or would it also be valid to analyze the nested function on its own? If the former, then information about the same function will be scattered across different trees with no way to aggregate them. If the latter, then the tool will analyze some functions multiple times to capture every possible level of nesting. Both approaches have benefits and drawbacks, and so we included both.

2b *Pattern density*: How should the tool quantitatively order its elements: by a function's raw frequency or by the unity of patterns it leads to? For example, if SUM has for its first argument two possibilities, one which itself contains 1000 unique argument possibilities with 1 occurrence each (high frequency, low pattern density) and another seen with 2 argument possibilities of 100 occurrences each (low frequency, high pattern density), which would be more interesting to emphasize? The answer depends on the nuances of the questions, but for simplicity, I've shown the former.

2c *Non-functions*: How should the tool represent everything in the formula that isn't a function: numbers, string literals, errors, references, etc? Since these don't accept arguments, they will adorn the tree as leaves, and their precise content won't affect the functions around them as long as their types are known. As such, in the visualization, all of these nodes are replaced and aggregated under their types.

3a *Suitable examples*: Goal 3 supports the inclusion of examples in the visualization as a way of tying pattern to example, but how should examples be chosen? For this, the simpler is the better, and we made the broad working assumption that shorter (by character length) functions are simpler, and thus chose the shortest available.

3b *Spreadsheet connections*: Is it possible to contextualize these examples even better? Yes, by leading the user directly back to the originating spreadsheet. The tool hyperlinks each example, then, back to the file that contains it, providing sheet, row, and column numbers if it doesn't open directly there.

D. Implementation

We can view the final visualization as the product of two discrete processes:

- *Collection*: Given a set of Excel sheets, the tool, written mostly in Java, uses Apache POI[5] to identify and iterate over every cell containing a valid formula. Afterward, it calls POI's formula parser to break the formula text into an ordered set of individual tokens, which the tool then parses into the tree-like form by which it is recorded. When all formulas have been analyzed like this, it produces JSON files for each top-level function in the set.
- *Presentation*: The JSON files, meanwhile, feed into the presentation code, implemented in Javascript with much help from the visualization library D3[6].

When the presentation code displays the tree, it shows two types of nodes with different meanings: the circles represent a function that has been observed at that structural place, and hovering over these yields a tooltip with supplementary information and examples; and squares, which represent the argument positions for the function which is their parent. For example, if the IF function has been observed with three arguments, 3 squares will extend from it as children. Clicking on the square labeled '1' will yield more circles, all of which are functions or values that the tool observed as the first argument in an IF function; clicking on '2' will yield those possibilities in the second position; and so on.

E. Limitations

Because the data collection depends on POI's formula parser, it affords no leeway or partial information from a formula; it either processes it perfectly or throws it out. As such, the visualization inhibits insight into anything with syntactical errors or third-party functions that can't be evaluated without additional tools. Note, however, that this does not include standard Excel errors like #REF! and #DIV/0, which the parser handles well.

Many functions have an expected order of arguments, but some, like SUM and MATCH, can be reordered in various ways and maintain their value. However, the representation that this tool uses reinforces for all functions that the order is important, even in these exceptions.

[4]https://support.office.com/en-us/article/Excel-functions-by-category-5f91f4e9-7b42-46d2-9bd1-63f26a86c0eb

[5]https://poi.apache.org/

[6]https://d3js.org/

A larger and related problem is the loss of argument combination. In the example in section B, we saw how the tools conveys which arguments are used most often on each side of an equals sign, but it does not encode how these whether references were actually compared to strings and numbers the most or whether it corresponded more to all the arguments beneath those two

IV. Case Study

The obvious question to pose to any visualization is this: Why does it matter that we see this way? In other words, it is not enough to prove that something can be visualized; instead we must also demonstrate what we can be gained from any particular visualization. As such, we raised a few tasks in the introduction in which we thought the tool could help: identifying places for API improvement, detecting bad smells or common programmer misunderstanding, and guiding spreadsheet education. Thus, to demonstrate the tool's applicability, we sought out places in the tree which best exemplify these respective concerns.

A. Bad Smells

Fowler's (?) seminal description of code smells underscored an important point of code quality: between perfect code and bug-crippled spaghetti, there is a spectrum of code designs which, by themselves, are not faulty but nevertheless may point to problems and vulnerabilities in how code is written. Code smells, as we call them, soon commanded attention and earned wide-ranging applicability, eventually making it to the art of spreadsheet design. Since then, Hermans and others catalog the odors of spreadsheets, often built off Fowler's (?) original specifications to articulate the subtle problems in spreadsheet design.

Collected, some of the intra-spreadsheet smells fall well within the purview of this visualization tool. Some such smells can be found in a figure [FIGURE].

The smells described above, then, entail a corresponding structure visible in the visualization: the deeper a branch runs, or the taller the range of arguments, the more it emits the stench of suboptimal design.

To be sure, these rank occurrences must be leveraged with their prevalence across the dataset. One person's anomaly, though perhaps still interesting, will not be as impactful as a smell of thousands.

B. Education

V. Limitations

VI. Conclusion

[to be filled in]

-extensible to other spreadsheets?

Acknowledgment

This material is based upon work supported in whole or in part with funding from the Laboratory for Analytic Sciences (LAS). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the LAS and/or any agency or entity of the United States Government.

References

[1] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the numbers of end users and end user programmers," in *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*. IEEE, 2005, pp. 207–214.

[2] M. Fisher and G. Rothermel, "The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[3] T. Barik, K. Lubick, J. Smith, J. Slankas, and E. Murphy-Hill, "Fuse: a reproducible, extendable, internet-scale corpus of spreadsheets," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 486–489.

[4] F. Hermans and E. Murphy-Hill, "Enron's spreadsheets and related emails: A dataset and analysis," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 7–16.

[5] F. Hermans, M. Pinzger, and A. v. Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 441–451.

[6] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 2012, pp. 243–244.

[7] T. Igarashi, J. D. Mackinlay, B.-W. Chang, and P. T. Zellweger, "Fluid visualization of spreadsheet structures," in *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE, 1998, pp. 118–125.

[8] M. Clermont, *A scalable approach to spreadsheet visualization*, 2003.

[9] S. Hipfl, "Using layout information for spreadsheet visualization," *arXiv preprint arXiv:0802.3939*, 2008.

[10] F. Hermans, M. Pinzger, and A. Van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 451–460.

[11] F. Hermans, M. Pinzger, and A. van Deursen, "Breviz: Visualizing spreadsheets using dataflow diagrams," *arXiv preprint arXiv:1111.6895*, 2011.