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 an abundance of functions and operations to manipulate data, but it can be difficult to understand how programmers actually use these in practice. Furthermore, users can combine several functions into a complex formula, complicating matters for both researchers and practitioners who want to study formulae to improve spreadsheet practices. Therefore, we developed a tool that visualizes patterns of function combination in spreadsheets as an interactive tree of possibilities. Using spreadsheets from the public spreadsheet corpora, we then apply to the tool to real datasets and demonstrate its ability to capture the most common and most anomalous patterns of function combination and their contexts in actual workbooks.**

I. Introduction

Business and research alike a lot to spreadsheets, the tabular interface that offers users a structured way to store and manipulate potentially huge bodies of data. Their allure is in their versatility: while the novice can still work without a deep knowledge of programming, they can learn to expedite their work by building programs out of several available functions [**?**]. As such, it should not be surprising that when Scaffidi and colleagues estimated that over 50 million U.S. workers could be using them by 2012, with 25 million of them writing programs [1].

Considering this ubiquity, it's crucial to understand how people are actually using spreadsheets. After all, failures in their practice can be ruinous, as demonstrated in EuSpRIG's many horror stories of spreadsheets gone wrong. In 2013, for example, they detail the influence of an economics paper on debt and national growth which shaped many political initiates around it, only to discover that the spreadsheets used in the data analysis contained an error that reversed the findings. The other stories, likewise, do not augur well if shoddy spreadsheet practices persist [1].

Fortunately, many spreadsheet researchers have assembled and released a number of spreadsheet corpora to inform work of how people actually use these tools. Collections like EUSES [2], FUSE [3], and the Enron corpus [4] have already enabled fruitful work across the field. Hermans and colleagues, for example, used the EUSES corpus in 2011 to inform and evaluate their work on exploring code smells, or inelegant places in code that suggest the need for refactoring, between worksheets [5].

A tool, then, that empowers users of any intention 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 for a given dataset. Beyond this, though, it could also inform future work on recommending functions through an understanding of this patterns, or it could guide API improvement by suggesting functions to add or prune based how people actually use them. Furthermore, if the tool maintains the connection between its broad patterns and the actual instances of formulae in workbooks, it could serve as a boon to educators not only by showing which functions and combinations are most practical to learn but also by offering a bounty of instructive, and real, examples

This paper contributes a tool, informed by some of the aforementioned spreadsheet corpora, that seeks to accomplish these tasks by offering an interactive interface for exploring how people combine functions. It seeks to balance several needs, such as capturing these patterns in a broad enough overview while still be specific enough lead back to the precise places in spreadsheets where these patterns are realized. Along with the tool, then, is also a discussion of the various goals which the tool seeks to fulfill and the trade-offs that we negotiated at each point. All of this culminates in an evaluation of the tool through a study of possible cases to which the tool might apply.

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 [6], seek to visualize the hidden formulas within a spreadsheet by imposing the dataflow graphs over the cells. Likewise, Clermont [7] (and Hipfl [8], 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 [9] and Breviz [10] 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. Badame, for example, implements a number of Excel formula refactorings and uses

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

Euses in part to evaluate it [?]. Hermans, leveraging her work with the aforementioned dataflow diagrams, approached the same dataset to assess the prevalence of common inter-worksheet smells, such as feature envy and inappropriate intimacy [5]. Cunha and colleagues, likewise, constructed and refined a catalogue of smells and implemented the SmellSheet Detective tool to detect bad smells in chosen spreadsheets [?], which Abreu and colleagues later extended to detect faults. [?].

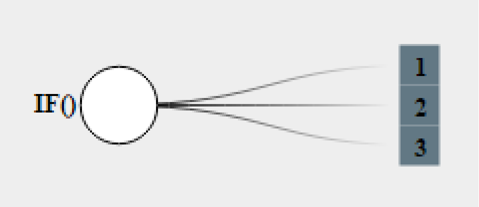
Implicit in this study is an exploration of how users interact with the Excel API. Previous research, such as the first study on the Enron spreadsheet corpus [4], on how users employ certain functions or commands. Researchers have applied such questions to other contexts as well, such as Murphy and colleague's study into how Java developers use Eclipse and what commands they use most often [?]. Others focus on the documentation of the API itself: Robillard and DeLine, for example conducted a series of surveys and interviews to diagnose common problems with learning APIs, and they found that code examples, one of the focuses of this tool, is one of the most important aids to have in learning a new system [?].

III. Approach/Methodology

A. Walkthrough of Tool

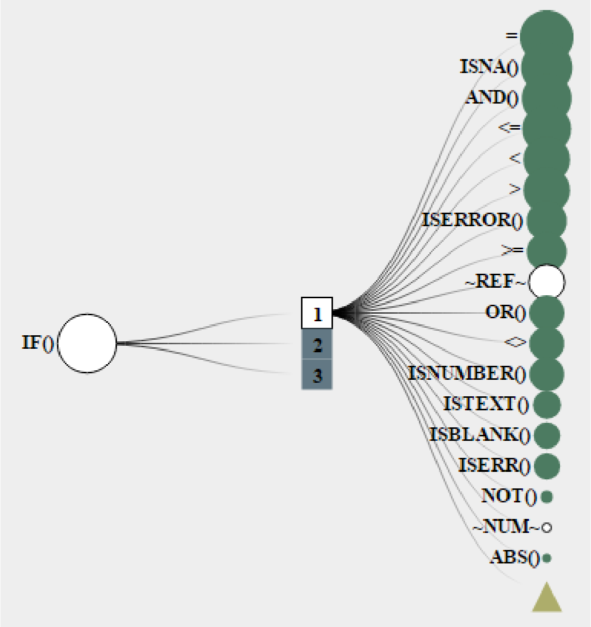
Before discussing the minutiae of how the tool was developed, it will help first to give a basic example of how the tool can be used and how to interpret its symbols. One such example might be to find the answer to this question: *What kinds of functions do people use to define the condition in an IF function?* The final visualization is available in figure 1; however, to understand the interaction, we will start rather from the beginning.

When the user first approaches the tree for a given top-level function – that is, a function nested within none other – only a few nodes are visible, as shown in figure **??**.



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.[2]

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:



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, which Figure 1 shows.

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. If this single instance is not enough, then the user also has the option to double-click the individual function node, which will open up a new tab with a table of many more examples from dataset.

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

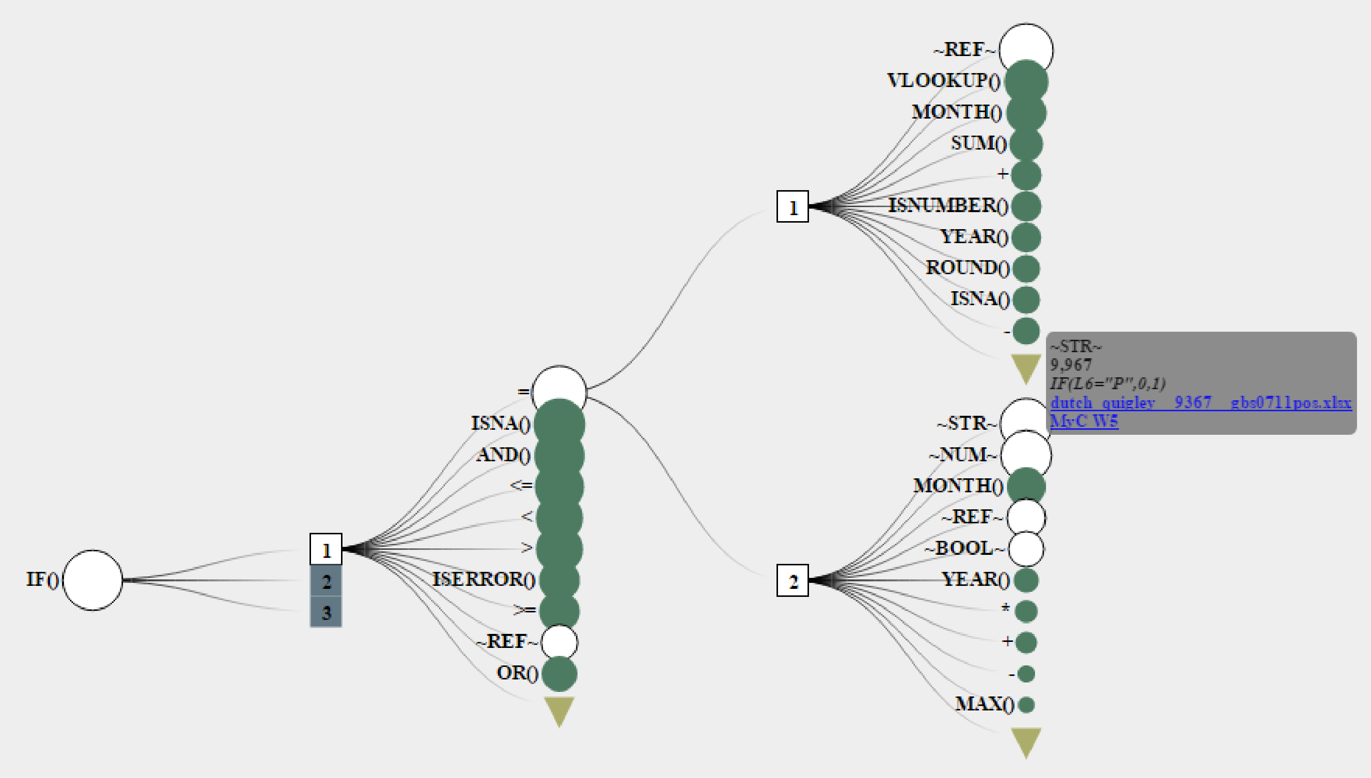


Fig. 1. A picture of the tool.

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 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.

C. Design Decisions

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, a sample of which we describe below, before we arrived at the design shown above.

- *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.
- *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?

  For example, in the pictured IF function, we found that people often use another IF statement as the second or third arguments of the top-level IF. If we only consider functions exactly where they're found embedded in the formula, then information about the same function – IF, in this case – will be scattered across different trees with no way to aggregate them. If we record every instance of a function by ignoring their context – that is, including an IF embedded within a SUM function in the same node as the top-level IF – then the tool will represent some functions in multiple nodes to capture every possible level of nesting. Both approaches have benefits and drawbacks, and so we included both.

- *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.
- *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 and represented as empty white nodes.
- *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. [3] Without prior evidence, it's possible that, when people use optional arguments, they use different kinds of types and functions for arguments versus the cases when they ignore optional 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.

D. Implementation Details

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[4] 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[5]. We chose D3 primarily for its accessibility: given the JSON, the visualization can be rather simply embedded into a webpage accessible through a browser.

A demonstration of the tool can be found at [LINK HERE].

IV. User Study

Midway through production, we conducted a brief, four-participant user study in order to gauge how well we were capturing the initial design goals. The study was, in general, a flexible and exploratory affair: rather than giving the users any specific tasks or pointed questions, we asked instead that they choose for themselves which trees to explore and report anything that they deemed interesting. Specifically, given trees rooted in the Enron dataset and at least 20 to thirty minutes, the participants situated themselves within the persona of a consultant asked to evaluate a company's spreadsheet practice, which could be arranged around questions like which functions on which the employees most depended, which sheets presented the most anomalous or dangerous designs, and so on. Interpretation of ”spreadsheet practice” can certainly vary from participant to participant, but since we wanted a range of views and backgrounds despite the persona, we readily accepted this interpretive wiggle room.

We clarified, furthermore, that these notes could be directed either at the quality of the data conveyed by the tool or at the tool's conveyance itself. Since the tool's philosophy was based in exploration, we decided that it would not have been appropriate to restrict what people were to look for, letting them find interesting stories of formula construction for themselves. Additionally, either kind of comment – on data or on tool – would direct us back to the central evaluation; a user's silence, or lack of any interesting findings, would tacitly indicate that perhaps the tool does not convey information well enough.

Still, several limitations cropped up during the study that complicate the evaluation. For one thing, though the source of the data was not explicitly stated at the outset of the session, several users studied the file names (unchanged as they were) to deduce its roots in finance, if not Enron itself. Because of these notorious connotations, then, some were inclined from this moment of discovery to orient their exploration around finance-related functions, potentially limiting their findings or perceived users for the tool – though some also said that, without this essential context, they could not have properly explored a dataset in the first place. Comments like these point out the trade-offs of unguided exploration, too: though it might not restrict them to a single purpose, it might, at the same time, not afford them any mindset at all with which to interpret the data. Another potential problem is that the data

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

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

[5] https://d3js.org/

which users explored was processed and created before we had fully completed the tool, meaning some of the counts and patterns in the trees were inaccurate. However, we decided that this was not a pressing concern, being that we were primarily with whether any interesting reports could be made at all, not with whether these reports were 100% accurate assessments of reality.

Nevertheless, these user studies were a formative moment in evaluating the tool's performance, generating a number of directions or corrections in the process.

A. Positive Responses

Gravitation to the grounded examples.
Indication of error.
Freedom of deciding errors.

B. Negative Responses

A lack of information.
A lack of comprehension.
A lack of ability.

V. 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.

Some examples are meaningful not for the space they take up but for how they clash with the expectations around a function. For example, Excel offers a number of functions that accept any number arguments given to them, up to a resource-defined limit. SUM and CONCATENATE are two such functions: adding more arguments simply adds or appends that element in the same way as every previous argument. Strange cases were found, then, in the cases where these functions were used with only one argument.

Hermans, in her paper on the nuances of VLOOKUP and its common misuses, explore the problem of the final, optional argument. A tool there helped, but how might a visualization such as this illuminated the same?

B. Education

Learning by example has long been a cornerstone of effective teacher techniques: by showing the student a concrete example of something, rather than dwelling in the realm of abstract description, comprehension increases [I'm sure there's a cite out there somewhere.] This is no different in software, where every explanation of a tricky programming concept is much untangled by the illustrative stub. Even APIs, researchers have found, by the inclusion of an in-practice example of the function described. Spreadsheet programming, then, should not be essentially different: examples of spreadsheet functions in use

C. 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.

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

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] 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.

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

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

[9] 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.

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