Splicing Data from a Multi-User Application

Mridu Nanda

May 13, 2020

Abstract

In this project we aim to provide systems level support for applications to efficiently implement "the right to erasure" as outlined in the General Data Protection Regulation (GDPR). As a starting point, we study a spreadsheet application which provides simple yet nontrivial semantics. We define deletion semantics for standard spreadsheet functions to ensure that functions evolve in a way that preserve the functionality of the application. We also support efficient deletion by introducing aggregation points, function decomposition and coalescing. We attempt to show that a spreadsheet application can be used to represent a database for a higher level application. To do so, we create spreadsheet functions to represent a list data structure and list operations. We then create a mapping between Trello, a workflow application, and our spreadsheet, and conclude that Trello is amenable to spreadsheet-like deletion semantics.

1 Introduction

In the age of big data, many applications have migrated users' private data from local machines to cloud storage. As a result, these applications are able to derive and disseminate information from stored server-side state, leaving users little control over how their private data is used. In response to the growing lack of user visibility into and control over data in the cloud, the EU passed the General Data Protection Regulation (GDPR). The GDPR outlines many high-level requirements for companies to follow, so that users can be re-equipped with control over their private data. For example, Article 17 of the GDPR mandates that users have the right to demand their personal data be deleted from an application. Unfortunately, however, the GDPR lacks the technical, systems level specifications to enforce such policies among a wide range of applications.

The high-level goal of this project is to implement Article 17, aka "the right to erasure", of the GDPR. While there exist some tools to create applications compliant with Article 17, these tools are neither very sophisticated, nor generalizable to arbitrary applications. For example, one can easily avoid detection from Office 365's tool, which uses regular-expression matching to find and remove sensitive data (e.g. SSNs). Therefore, we will attempt to provide systems-level support for splicing a user's data from a program. This problem is challenging because an arbitrary program written in an arbitrary language may include arbitrary data structures, each of which has application-specific semantics.

In this project, we study a subset of the automated user data deletion problem by focusing on a simpler, well-defined spreadsheet application. A spreadsheet application provides simple semantics because it only supports a limited number of data types and restricts the expressiveness of functions. A spreadsheet, additionally, provides nontrivial semantics to study because of its inclusion of formulas. For example, it is not immediately clear how a formula should evolve when its underlying data is deleted. It is also

unclear how the modification of one formula may affect the functionality of the rest of the spreadsheet. Therefore, the first goal of this project is to design semantics such that when values are deleted, formulas evolve in a way that (1) keep deletion efficient (2) preserve the functionality of the original spreadsheet.

A spreadsheet is also interesting to study because it can represent a database for a more complicated application. For example, we will show that Trello, a workflow application, can be modeled by our spreadsheet through converting it's high level objects into list data structures. As such, the second goal of this project is to determine which subset of applications in general are amenable to spreadsheet-like deletion semantics.

2 Background

Several projects have attempted to create GDPR compliant applications; however, these projects are unable to handle the data splicing problem for applications that contain multiple users. For example, Riverbed, a distributed web platform, provides a information flow control (IFC) mechanism to prevent sensitive data from flowing to disallowed sinks like storage devices. Applications running atop Riverbed are completely unaware that the IFC is occurring; therefore, Riverbed compatible with code that has not been explicitly annotated with traditional IFC labels. In order to avoid conflicts between different data flow policies, Riverbed spawns a lightweight copy of the service for each group of users who share the same policies. Each copy of the service is referred to as a universe. For example, if Alice and Bob's policies allow aggregation, but Charlie's policy does not, then Alice and Bob will be placed into one universe and Charlie will be placed into another. However, because Riverbed is application-agnostic, the platform has no way to automatically splice a user's data out of one universe and into another. Continuing from the above example, if Bob changes his policy, Riverbed must use application specific methods to extract Bob's data from Alice's. Riverbed, then, must also use application specific methods to delete Bob's data from the old universe, as well as re-inject Bob's data into the new universe. Therefore, Riverbed does not provide a clean solution for splicing user data out of universes.

Rust's ownership model provides an interesting insight on how to splice aggregated data. The ownership model specifies the following rules: 1) Each value in Rust has a variable that is called its owner and 2) There can only be one owner at a time (cite). As a consequence of these semantics, each program variable in Rust can only be referenced by a single incoming pointer. Therefore, the state of a Rust program can be represented by a set of trees, where each tree represents a collection of parent/child relationships between variables [1]. Lack of cycles in Rust's model would help restrict the possible cascading impacts of deletion. Unfortunately, however, Rust's ownership model is too restrictive to use for a spreadsheet application, since one cell may be referenced in many different formulas.

[1] Rust uses ownership semantics to make memory allocation and deallocation simpler: when an object's owner goes out of scope, the entire tree is deallocated.

3 Design

3.1 Definitions

A spreadsheet application consists of three abstractions: *values*, *cells*, and *functions*.

- A *value* can be an integer, float, or string. A value is owned by the user who inserted the value into the spreadsheet. Every value has a permanent owner for its lifetime inside the spreadsheet.
- A *cell* is a location in the spreadsheet, and is referred to by its grid-wise coordinates. For example, `A1`, and `B1` are both cells in a spreadsheet.
- A *function* is an expression composed of operators and operands. Operators include our standard notion of arithmetic, relational, logical and bitwise operators. Operands include values, references to cells, and functions. For example `SUM(A1, A2)` is a function that is composed of an arithmetic operator and two references to cells.

When a user inserts data into the spreadsheet, she is dynamically bounding her value to a cell. For example, if a user, Alice, inserts her SSN at cell `A1` inside the spreadsheet, the SSN is the value, Alice is the owner of that value, and the SSN lives at location `A1` inside the spreadsheet. Furthermore, a value may migrate to different cells over the evolution of the spreadsheet. For example, Alice's SSN may be moved to cell `A2` at some future point; however, we maintain the invariant that the SSN is still owned by Alice.

A function can be a standard spreadsheet function such as `SUM()`, or a user defined function constructed from several standard functions. In general, functions act as aggregation points for cells. For example, functions either aggregate values from the same owner, as done in function `SUM(C1, C2)` in Table 1, or aggregate values from multiple owners, as done in function `SUM(A1, A2, B1)` in Table 1.

We divide the overall spreadsheet application into two logical namespaces: the *cell namespace*, and the *function namespace*. The cell namespace will contain cells and values, and will be accessible to all users that must input data into the spreadsheet. The function namespace will contain functions. Every function that has a distinct entry in the function namespace is called a named function. For example, every user defined function is an named function. Functions 1, 2 and 3 in Table 1 are also examples of named functions. A function is considered to be an anonymous function if it serves as an operand to another function, but does not have a distinct entry in the function namespace. For example, the standard spreadsheet function `AVERAGE()` is implemented with anonymous `SUM()` and `DIV()` functions. The `DIV()` function nested inside `SUM(DIV(A1, A2), F2)` is another example of a anonymous function. There exists one function namespace per user who issues computations over the values inside the cell namespace. For example, if Alice chooses to sum over a range of cells, the function associated with this computation will be stored inside of Alice's function namespace. Then if Bob chooses to carry out a different

computation, the associated function will be stored inside of Bob's function namespace. While in general, we can have an arbitrary number of function namespaces, in Table 1, we have made the simplifying assumption that the "application owner" is only user who is computing over values; therefore, there is exactly one function namespace.

Cell Namesapce			
	A	B	C
1	666	7	-3
2	0.42	Hello World	77

Function Namespace	
f1	SUM(A1, A2, B1)
f2	SUM(C1, C2)
f3	SUM(DIV(A1, A2), *f2*)

Table 1: Alice enters values in column `A`, Bob enters values in column `B` and Charlie enters values in column `C`. The spreadsheet owner computes over the available values in the function namespace.

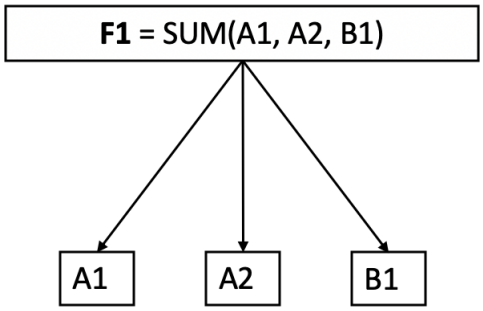


Figure 1: Object graph showing the relationship between cells and *f1* in Table 1

3.2 Threat Model

We assume that the owner of the spreadsheet is honest but curious. Specifically, our threat model excludes an actively malicious developer who would take steps to record user data. This includes a developer who might take snapshots of the spreadsheet state, to enable subsequent inspection of what user data looked like a prior time. This also includes a developer who would attempt to pass information via side channels by encoding meaning inside specific functions.

Similarly, we assume that the users of the spreadsheet are also honest but curious. Specifically, we exclude users who might try to take active steps to record other users' data. For example, we would exclude a user Bob who would actively view Alice's data, and re-enter that data into the spreadsheet as his own. User's are restricted to entering primitive values and functions into the spreadsheet; compound values, comments, and arbitrarily-encoded binary data are disallowed.

3.3 System Guarantees

Users can issue two types of deletion requests: (1) a value deletion request (2) a function deletion request. Value deletion requests can only be issued by the owner of the value and

function deletion request can only be issued by the user who defined the function. For example, in Table 1, Alice can issue a value deletion request for her values located in cell `A1` and `A2`. Similarly, the application owner can issue a function deletion request to delete `SUM(B1, C2)`.

To complete a function deletion request, we must delete the named function that was specified in the request. This includes deleting any anonymous functions that the named function might reference. However, we need not to delete any named functions that the original function might reference. For example, if the application owner requested to delete `SUM(DIV(A1, A2), f2)` in Table 1, we would also have to delete the anonymous function `DIV(A1, A2)`. However, we would not delete `SUM(B1, C2)`, which the original function contains a reference to via the cell name *f2*. Function deletion requests only modify the function namespace.

To complete a value deletion request, we first delete the the specified value from the cell namespace. Next, our runtime must adjust the formulas in the function namespace to reflect the deletions to the cell namespace. Therefore, value deletion requests modify both the cell and function namespaces. When modifying the function namespace, our runtime keeps two goals in mind: (1) keep deletion efficient, and (2) maintain the functionality of the original spreadsheet. In the following sections, we introduce aggregation points and function decomposition to implement efficient deletion, and deletion semantics to maintain the functionality of the spreadsheet.

3.4 Aggregation points

Aggregation points are useful for quickly locating a values in the cell namespace. In order to implement this optimization, our runtime will insert a aggregation point per user to track inputted values.

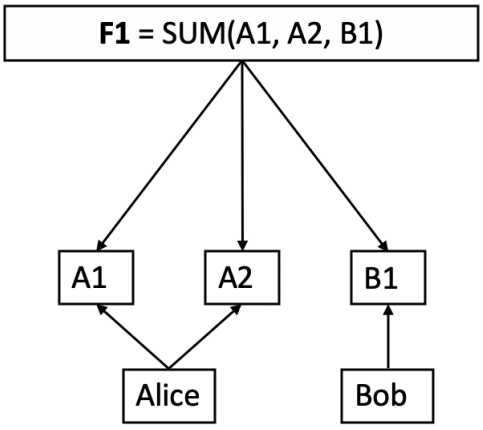


Figure 2: Object graph from Figure 1 updated with aggregation points

Then when the user issues a value deletion request, our runtime will look to the user's aggregation point and delete all requested values from the cell namespace. Therefore,

aggregation points support efficient deletion by eliminating traversals through the cell namespace.

3.5 Function Decomposition

When our spreadsheet receives a new function, it may internally decompose the function into a collection of sub-functions, each with one owner, that compute an equivalent result. Decomposition enables faster re-computation of a modified top-level function because the outputs of the constituent sub-functions can be cached; when a sub-function is deleted, the spreadsheet only needs to combine the cached outputs of the remaining sub-functions, instead of having to re-execute all of the remaining sub-functions. Function decomposition can only occur when (1) a function consumes two or more values (2) the input values belong to at least two different users, and (3) the sub-functions functions can be re-combined in a way that is equivalent to the original function.

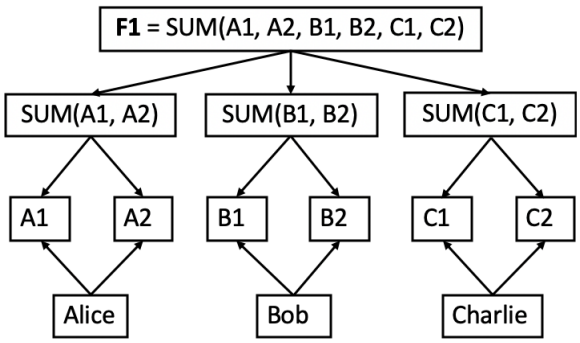


Figure 3: Object graph with aggregation points and function decomposition

For example, suppose we have a function SUM(A1, A2, B1, B2, C1, C2) where Ax belongs to Alice and Bx belongs to Bob and Cx belongs to Charlie. Then suppose that Bob requests deletion. If we cached intermediate computations SUM(A1, A2), SUM(B1, B2) and SUM(C1, C2), as shown in Figure 3, then upon Bob's deletion we would only need to add the pre-computed sums for Alice and Charlie to obtain our new sum. However, if we did not store these intermediate computations, then upon Bob's deletion, we would have to iterate through all of the remaining values to recompute SUM(A1, A2, C1, C2). The performance differences between the two schemes gets larger as the number of leaf nodes in the object graph increases. Therefore, storing intermediate computations from the same owner avoids unnecessary re-computation, and thus speeds up the deletion process.

3.6 Deletion Semantics

In this section we will specify deletion semantics for standard spreadsheet functions, in order to maintain the functionality of the original spreadsheet. The chart below shows standard spreadsheet functions grouped by the deletion semantic and type of operator. Note that these two categories are orthogonal; two operators with different types can have

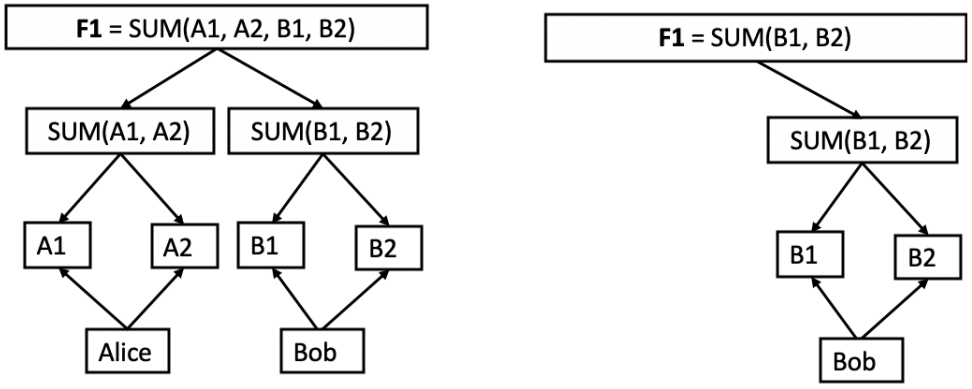
the same deletion semantics. Similarly, two operators with the same type may have different deletion semantics.

	Operator	Function	# of Operands	Deletion Condition
Semantic 1	Arithmetic	Sum Product Max Min Count	At least one	No operands left
	Logical	And Or		
Semantic 2	Arithmetic	Division Subtraction	Exactly two	Any operand deleted
	Relational	Equal Less Than Greater Than		
	Logical	Xor		
	Bitwise	And Or Xor		
Semantic 3	Arithmetic	Average	At least one	No operands left
Semantic 4	Other	If	Exactly three	Any operand deleted

Table 2: Standard spreadsheet functions grouped by operator and deletion semantics.

Semantic one includes commutative functions that take multiple arguments. All functions in this category make use of the function decomposition optimization discussed in the previous section. We note that this decomposition varies between functions in this category. For example, when `SUM` is decomposed, the resulting sub-functions and the parent aggregation function have the same type, that is `SUM`. However, when `COUNT` is decomposed the resulting sub-functions and the parent aggregation function have different types; the sub-functions are `COUNT`s while the parent aggregation function is a `SUM`. Regardless of possibly different decomposition characteristics, we maintain the invariant that functions in this category can be properly decomposed to avoid re-computation.

To demonstrate this semantic in action, consider the function `SUM(A1, A2, B1, B2)` where Alice owns cells Ax and Bob owns cells Bx. We start by decomposing this function into the intermediate computations `SUM(A1, A2)` and `SUM(B1, B2)`. If Alice requests her values to be deleted, we will delete `A1` and `A2` from the object graph. Next, we will delete `SUM(A1, A2)` since both of its operands have been deleted. This leaves us with the function `SUM(B1, B2)` pointing to Bob's intermediate computation `SUM(B1, B2)`.



(a) Object graph prior to Alice's deletion (b) Object graph after Alice's deletion

Figure 4: Object graphs demonstrating `SUM()` deletion semantics

Semantic two includes functions that are mathematically defined to take exactly two operands. We recall that a function can only be decomposed if it is possible to recombine the sub-functions in a way that is equivalent to the original computation. However, if we decompose a function in this category we would create sub-functions that have no mathematical meaning. As a result, it would be impossible to recombine the sub-functions in a way that is equivalent to the original computation. For example, if we tried to decompose `DIV(A1, B1)`, we would be left with some variation of functions that look like `DIV(A1)` and `DIV(B1)`. However, neither `DIV(A1)` nor `DIV(B1)` are mathematically defined, so there would be no way to re-combine them to obtain a value equivalent to the original `DIV(A1, B1)`. Therefore, we do not decompose functions in this category. Furthermore, because these functions are only mathematically defined to take two operands, we delete these functions if either operand is deleted.

Semantic three is only used by the `AVG()` function. This function is implemented with `SUM()`, `COUNT()` and `DIV()` functions. If any of the anonymous functions that implement `AVG()` get deleted, `AVG()` also gets deleted. For example, if all the values to be averaged get deleted, then according to semantic one both `SUM()` and `COUNT()` must be deleted, which would result in `DIV()` being deleted according to semantic two. Therefore, the enclosing `AVG()` function will also be deleted.

Semantic four is only used by the `IF()` function. The default version of this function takes in three operands and is deleted when any of the operands are deleted. For example, if we have the function `IF(A1 > B1, SUM(A1, A2) DIV(B1, B2))`, then the entire function will be deleted if any of `A1 > B1`, `SUM(A1, A2)` or `DIV(B1, B2)` are deleted. However, it is possible that the application owner might want to keep the `IF()` function even if the second or third operands are deleted. For example, the application owner might want to maintain the functionality of the previous `IF()` function even if `DIV(B1, B2)` was deleted. In this case, we allow the application owner to override the default deletion semantic so that a particular `IF()` function will not be deleted if either the second or third operands are deleted. Regardless, in both the default and customized deletion semantics,

we keep the invariant that if the first operand is deleted, then the `IF()` function must be deleted.

3.7 List Data Structure

So far our discussion of deletion semantics has only included functions available in a standard spreadsheet application. All of such functions are static, meaning that once created, excluding deletion, the number of operands inputted to these functions do not change. However, in order for a spreadsheet to represent the database of a higher level application, we must allow the number of operands to these functions to evolve over time. Therefore, in this section we will equip the spreadsheet application with a list data structure that can serve as an input to static functions.

We will define two functions to enable dynamic computations in our spreadsheet: `LIST(Head)` and `LIST(Head, Tail)`. The function `LIST(Head)` specifies all values in a column starting with `Head` and ending with the last non-empty cell in the column. Similarly, the function `LIST(Head, Tail)` specifies all values in a column starting at `Head` up to `Tail`. Both of these functions are typically inputs to a static spreadsheet function. For example, the function `SUM(LIST(A1))` is interpreted as "sum over all the values starting at cell `A1` ending with the last non-empty cell in column `A`." Just like static functions, the list functions live in the function namespace and can be invoked by any user.

When we delete a value that is referenced by a `LIST` function, we must update the `LIST` function to reflect the new range of populated cells. To do so we can either re-write the `LIST` function and keep the cell namespace the same or update the cell namespace and keep the `LIST` function the same. For example, if we have the function `LIST(A1)` and the value in cell `A1` is deleted, we could either re-write the function to `LIST(A2)` or move all the cells one position upwards to avoid modifying `LIST(A1)`. To support efficient deletion, we want to avoid function re-writing; therefore, upon deletion, we will update the cell namespace to minimize changes to the `LIST` functions.

In order to update the cell namespace, we will introduce a technique called coalescing. Coalescing is the act of re-arranging cells per column, such that the column contains a contiguous region of cells with values. Coalescing must preserve the ownership of values, and the spatial relationships between the cells that values are bound to.

	A	B	...
1	10		
2		22	
3	30		
4	40	44	
...	...		

(a) Cell namespace prior to coalescing

	A	B	...
1	10	22	
2	30	44	
3	40		
4			
...	...		

(b) Cell namespace after coalescing

Figure 5: Green cells are owned by Alice and blue cells are owned by Bob

Therefore, when a user issues a deletion request for a value referenced in a `LIST` function, coalescing happens upwards to the `Head` of the `LIST`. Coalescing also provides an advantage in improving spreadsheet readability.

Coalescing, however, does not prevent all `LIST` function re-writing. For example the function `LIST(Head, Tail)` will have to be re-written at every deletion in order to update the `Tail`. Furthermore, coalescing can cause a cascading number of function re-writes in the case of nested lists. For example, consider the following functions: `LIST(A1, A6)`,`LIST(A2, A5)` `LIST(A3, A4)`. In this case, if we were to delete `A1`, the functions would be updated to: `LIST(A1, A5)`,`LIST(A1, A4)`,`LIST(A2, A3)`. Notice, how for the latter two functions, both the `Head` and the `Tail` had to be re-written due to coalescing. In our model, we assume that nested lists will not be a common use case, and thus are not optimized for. However, we could imagine a scenario where each nested list was written in its own column to prevent extra `Head` re-writes.

Finally, in order to maintain spreadsheet functionality, we must specify the deletion semantics for both `LIST` functions. The function `LIST(Head)` is deleted when there are no longer any values in the column starting from `Head`. Similarly, the function `LIST(Head, Tail)` is deleted when there are no values contained between `Head` and `Tail`.

3.8 Assignment Operators

Finally, in order for a spreadsheet to model a higher level application we need to define a mechanism to populate values inside the cell namespace. To do so, we define the following assignment operators: `OVERWRITE(Cell, Value)`,`LIST INSERT(Cell, Value)`, `MOVE(Cell, Value)`,`LIST TRANSFER(Cell, Value)`.

The function `OVERWRITE(Cell, Value)` gives users the ability their input value at a specific location in the cell namespace. For example, `OVERWRITE(A1, 42)` allows a user to insert the value 42 at cell `A1`. If the cell is previously occupied, then the function will overwrite the old value, granted that the owner of the old value matches the user who invoked `OVERWRITE`. For example, if `A1` contained the value 666 owned by Alice, and Alice called `OVERWRITE(A1, 42)`, then cell `A1` would now contain the value 42.

However, if Bob called `OVERWRITE(A1, 42)` the function would return an error.

The function `LIST INSERT(Cell, Value)` allows users to append their `Value` to a location inside a preexisting list. We note that this function is orthogonal to the `OVERWRITE(Cell, Value)` function. For example, in order to overwrite a value in a list, we would have to use the `OVERWRITE` function. However, to add a new value to a list, we must use the `LIST INSERT` function. To implement `LIST INSERT`, we first move every value, starting from `Cell`, one position down. Then we insert `Value` at position `Cell`, and update any affected `LIST` functions. For example, if we called the function `LIST INSERT(A3, 42)`, we will move all cells starting from `A3` down one position, insert 42 in cell `A3`, and update the `Tail` for the appropriate `LIST` functions. This function will return an error if the `Cell` is not contained within a `LIST`.

To implement `MOVE(Cell, Value)`, we first find and delete `Value` from the cell namespace. Next, we simply call `OVERWRITE(Cell, Value)`. Similarly, to implement `LIST TRANSFER(Cell, Value)` we delete the `Value` from the cell namespace and call `LIST INSERT(Cell, Value)`.

Unlike all of the operators we have defined so far, we do not store these assignment operators in the function namespace. As a result, we need not define deletion semantics for these functions. However, we could imagine a scenario where we store functions in a separate log to show how the spreadsheet evolves over time. In this case, if a user requested to be deleted, we would also delete all of her logged `OVERWRITE`, `LIST INSERT`, `MOVE`, and `LIST TRANSFER` functions.

4 Trello

In this section we will show how the workflow application, Trello, can be mapped to a simple spreadsheet application. A Trello application consists of Lists, Cards, Checklists, Tasks, and Activity Log objects. The relationship between these objects is shown below.

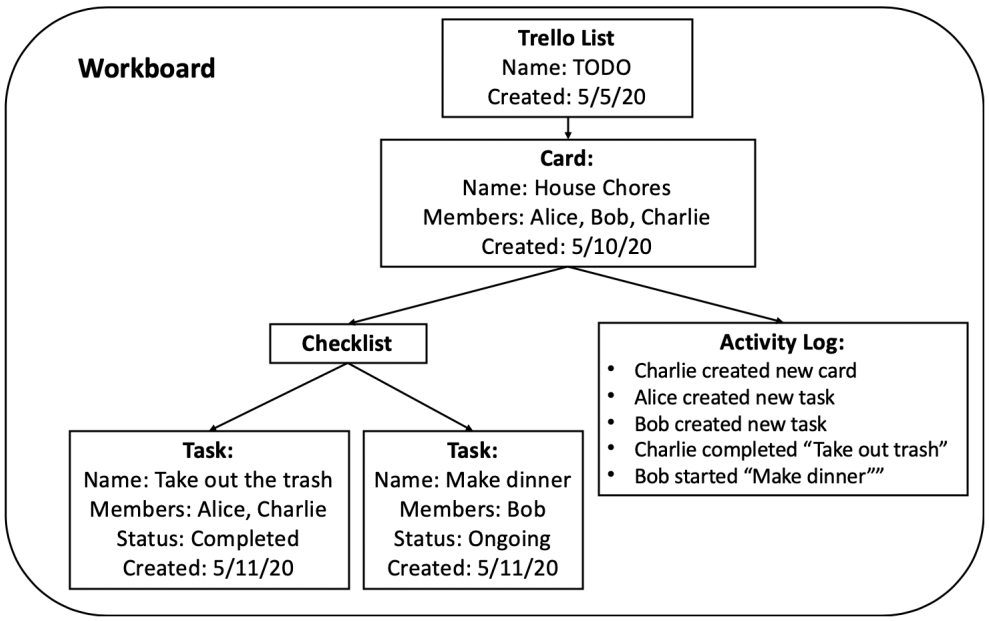


Figure 6: In this workboard, we have one list, that contains one card. This card has both a checklist and an activity log. The checklist contains two tasks.

To map Trello to our spreadsheet application, we first must decide ownership semantics per Trello object. We start by noting that users own their entries in the activity log. For example, in Figure 6, Alice owns ”Alice created a new task”. For the rest of the Trello objects, we give users ownership over their membership status, while giving the workboard owner ownership over the metadata associated with each object. For example, in Figure 6, Bob owns his membership status in the task and the workboard owner owns the task name, status, and created fields. These semantics prevent the scenario where deleting Bob's data results in deleting the entire task.

Next, we must convert each Trello object into our spreadsheet abstractions. The below figure shows the spreadsheet representation of Figure 6.

Cell Namespace

	A	B	C	D	E	F	G	H
1	Alice	Task name: Take out trash	Bob	Task name: Make dinner	Charlie created new card	Alice	Card Name: House Chores	List name: TODO
2	Charlie	Task status: Completed		Task status: Ongoing	Alice created new task	Charlie	Card created: 5/10/20	List created: 5/5/20
3		Task created: 5/11/20		Task created: 5/11/20	Bob created new task	Bob		
4					Charlie completed "Take out trash"			
5					Bob started "Make dinner"			

(a) Green cells owned by Alice, blue cells owned by Bob, yellow cells owned by Charlie and gray cells owned by Workboard owner

Function Namespace

f1	MEMBER(LIST(A1)
f2	METADATA(LIST(B1)
f3	MEMBER(LIST(C1))
f4	METADATA(LIST(D1))
f5	MEMBER(LIST(F1))
f6	METADATA(LIST(G1))
f7	METADATA(LIST(H1))

f8	TASK(*f1*, *f2*)
f9	TASK(*f3*, *f4*)
f10	CHECKLIST(*f8*, *f9*)
f11	ACTIVITY LOG (LIST(E1))
f12	CARD(*f5*, *f6*, *f10*, *f11*)
f13	TRELLO LIST(*f7*, *f12*)

Figure 7: Spreadsheet representation of the Trello application in Figure 6.

We start by creating a spreadsheet function per high level Trello object. Next, we must specify the operands to each function. The operand to the `ACTIVITY LOG` function will be a `LIST` that represents all log entries. For example, to convert the Figure 6's Trello activity log, we list all of the log entries in column E of the cell namespace and create function *f11*: `ACTIVITY LOG(LIST(E1))` in the function namespace of Figure 7. The operands to the `TASK` function will be a `METADATA` function and `MEMBER` function. The `METADATA` and `MEMBER` functions will each take in a `LIST` of values, where the values are owned by the semantics described previously. Then to convert Figure 6's Trello task "Take out the trash", we list the task's members in column `A` and task's metadata in column `B` in the cell namespace of Figure 7. We then create functions `MEMBER(LIST(A1)` and `METADATA(LIST(B1)` as operands to *f8*: `TASK(`*f1*`,` *f2*`)` in Figure 7. The operands to the `CHECKLIST` function will be `TASK` functions. For example, Figure 6's checklist is represented by *f10*: *f8*, *f9*) in Figure 7, where *f9* and *f10* are both `TASK` functions. The operands to a `CARD` function will include `METADATA` and `MEMBER` functions. A `CARD` function might also take a `CHECKLIST` and `ACTIVITY LOG` function, as operands as shown in

f12: TASK(*f5*, *f6*, *f10*, *f11*) of Figure 7. Finally, the operand to a TRELLO LIST function will be a METADATA list and as many CARD's associated with the Trello list. For example, Figure 6's Trello list is represented by *f13*: TASK(*f7*, *f12*) where *f7* is a METADATA list shown in column H of the cell namespace and *f12* refers to a CARD function.

The below table summarizes the deletion semantics for each of the Trello spreadsheet functions.

Function	**Possible Operands**	**Deletion Condition**
Trello List	Metadata, Card(s)	Metadata is deleted
Card	Metadata, Activity Log, Checklist, Member	Metadata is deleted
Checklist	Metadata, Task(s)	Metadata is deleted
Task	Metadata, Member	Metadata is deleted
Activity Log	List of values owned per user	Deletion condition for LIST
Member	List of values owned per user	Deletion condition for LIST
Metadata	List of values owned by workboard owner	Deletion condition for LIST

Table 3: Deletion semantics for Trello functions

From this table, we note that a TRELLO LIST, CARD, CHECKLIST, and TASK functions can only deleted if the associated METADATA function is deleted. As described above, the METADATA function has the typical LIST semantics, where every cell is owned by the Trello workboard owner. Therefore, consistent with the actual Trello application, the workboard owner controls the deletion of high-level Trello objects. Furthermore, users are able to control their membership for each object via the assignment operators described in the previous section.

Suppose Charlie, from Figures 6 and 7, requested all his data be deleted. According to our deletion semantics, our spreadsheet application would look like:

Cell Namespace

	A	B	C	D	E	F	G	H
1	Alice	Task name: Take out trash	Bob	Task name: Make dinner	Alice created new task	Alice	Card Name: House Chores	List name: TODO
2		Task status: Completed		Task status: Ongoing	Bob created new task	Bob	Card created: 5/10/20	List created: 5/5/20
3		Task created: 5/11/20		Task created: 5/11/20	Bob started "Make dinner"			
4								
5								

Figure 8: Cell namespace from Figure 7 after Charlie requests deletion. Note, this represents the cell namespace after coalescing. We do not show the function namespace of the spreadsheet, since it has not changed.

This would result in a Trello level application that looks like:

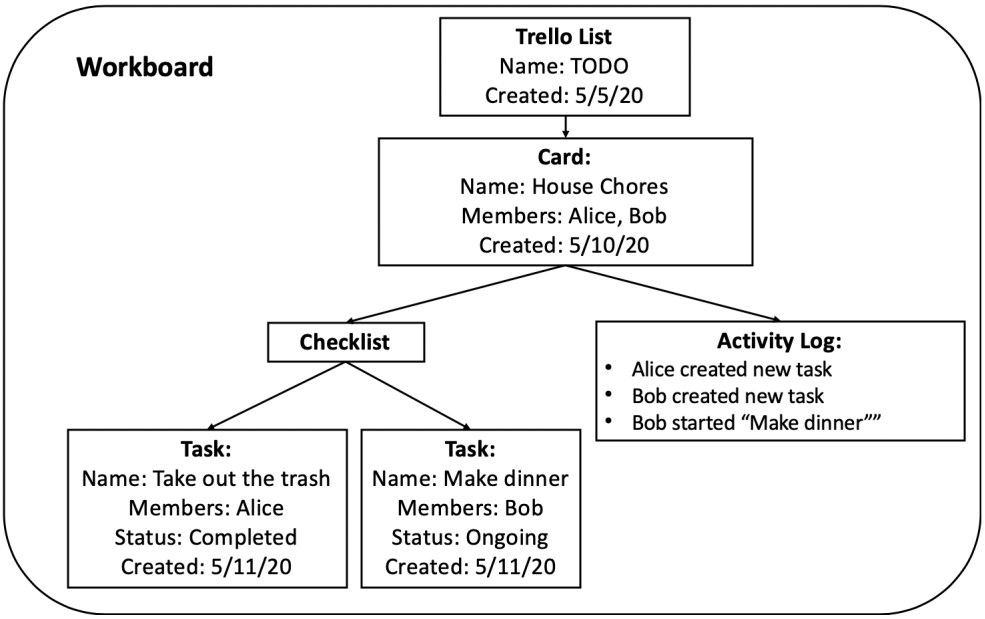


Figure 9: Representation of Trello application after Charlie requests deletion. Notice how the Members fields of the Card and Tasks have changed. Also notice Charlie's activity log entries are no longer present.

which looks exactly as expected. Therefore, we have successfully created a mapping from Trello to our spreadsheet application and we can conclude that Trello has spreadsheet-like deletion semantics.

5 Conclusion

In this project we studied a subset of the automated user data deletion problem by focusing on a spreadsheet application. We designed deletion semantics such that when values are deleted, formulas evolve in a way that preserve the functionality of the original spreadsheet. We supported efficient deletion by introducing aggregation points, function decomposition, and coalescing. In order for our spreadsheet to represent a database for a high level application, we created new functions to represent a list data structure and list operations. Through these tools, we created a mapping between Trello and our spreadsheet, and concluded that Trello is amenable to spreadsheet-like deletion semantics.

6 Future Work

The next step for this project will be to determine the general properties of a spreadsheet-amenable application. We hope to use this insight to find more applications that can be mapped to our spreadsheet.