

“Fine-Grained Lineage for Safer Notebook Interactions”

M. Chalke, F. Escudero, N. Lyubenko

mchalke@hawk.iit.edu

fdelrealescudero@hawk.iit.edu

nlyubenko@hawk.iit.edu

Illinois Institute of Technology

Introduction

Programming notebook environments are dynamic and interactive platforms for code to execute in a non terminal fashion in order to facilitate the ad hoc exploratory nature of data science. Computational notebooks use cells as the atomic execution unit and allow users to modify and re-execute previous cells with tighter feedback. However these notebooks have a few drawbacks such as out-of-order cell execution, cell deletion, cell editing and re-execution which make it difficult to maintain notebook state. This can be problematic when cells are manipulated non topologically. These interactions can cause desynchronization of the hidden intermediate state, making execution behavior difficult to trace thus producing bugs and irreproducible results. The authors present a notebook library for Jupyter that manages the topological lineage of cells and automatically synchronizes the hidden intermediate state upon changes to dependent cells. We will describe the innovations of the NBSAFETY library and how it solves the problem of desynchronization in notebook environments.

Research Challenges

The goal of this paper is to develop techniques to identify and prevent unsafe cell executions, while maintaining the notebook semantics. Some challenges that the authors encountered are:

1. Automatically detecting unsafe interactions

They determined that static analysis, although necessary, alone is insufficient as all branches of the control flow must be considered. The idea is to find the right balance.

2. Automatically resolving unsafe behavior with suggested fixes

Besides detecting potentially unsafe interactions it would be helpful to identify which cells to run in order to resolve staleness issues. Automatically rerunning cells when a staleness issue is detected is an option, however, there could potentially be many cells which need re-execution to resolve a single staleness issue.

3. Maintaining interactive levels of performance

Detecting and resolving unsafe interactions may introduce unacceptable latencies or memory usage. The key is to control lineage metadata size and efficiently identify cells that resolve staleness issues. Comparing NBSAFETY to other notebooks, Dataflow notebooks require users to annotate cells with their dependencies, and force the re-execution of cells with changed dependencies. Nodebook and the Datalore kernel attempt to enforce a temporal ordering of variable definitions in the order that cells appear compromising on flexibility. Dataflow notebooks observe reactive execution order, while Nodebook and Datalore's kernel observe forced in-order execution.

NBSAFETY, in its attempt to address these challenges, requires a single installation command and can be used by users of JupyterLab and traditional Jupyter notebooks as replacement for the Python 3 kernel. It combines runtime tracing with static analysis in order to detect and prevent notebook interactions that are unsafe due to staleness issues. It also efficiently detects cells whose re-execution would resolve staleness issues using a technique called initialized variable analysis. This brings staleness resolution complexity down from time quadratic in the number of cells in the notebook to linear, crucial for large notebooks.

Architecture

The tracer runs through program statements to annotate program variable definitions with parent dependencies and cell execution timestamps. This metadata is then used by a static checker that combines metadata with static program analysis techniques to determine whether any staleness issues are present prior to the start of cell execution. This allows NBSAFETY to highlight the necessary cells and warn about cells that are unsafe to execute due to staleness issues preserving atomicity of cell executions.

When the user submits a request to run a cell, the tracer instruments the executed cell and updates the lineage metadata associated with each variable as each line executes. In its attempt to be non-intrusive when maintaining the lineage of notebook objects, it creates shadow references to each symbol. After each statement has finished executing, the tracer performs a lineage update. The checker then performs liveness analysis and initialized variable analysis for every cell in the notebook. By combining the results of these analyses with the lineage metadata computed by the tracer, the frontend highlights cells that are unsafe due to staleness issues. The extra visual cues help the user make a more informed decision about which cell to next execute.

Lineage Tracking

NBSAFETY traces cell execution in order to maintain symbol lineage metadata, and this metadata aids in the detection and resolution of staleness issues. To begin, the authors define the following:

Symbol: A symbol is any piece of data in notebook scope that can be referenced by a name. [1] They can be thought of as variables allowing us to treat nameable objects in a unified manner. Each symbol contains its own lineage metadata in the form of timestamps and dependencies.

Timestamp: A symbol's timestamp is the execution counter of the cell that most recently modified that symbol.[1] Similarly, a cell's timestamp is the execution counter corresponding to the most recent time that cell was executed.

Stale symbols: A symbol s is called stale if s has a parent that is either itself stale or more up-to-date than s . [1]

NBSAFETY is able to track lineage at a finer granularity than simply top-level symbols. When a symbol is updated, a depth first search starting from each child is performed in order to propagate the staleness to all descendants.

It is difficult to maintain acceptable performance in such cases. When the tracer traps due to a function call, if the called function is externally defined, tracing is disabled until control returns to the notebook.

Liveness and Initialized Variable Analysis

The checker performs liveness analysis and initialized variable analysis. These techniques help to identify cells that are unsafe to execute due to stale references, as well as the cells that help resolve staleness issues.

Liveness Analysis[1] is a program analysis technique for determining whether the value of a variable at some point is used later in the program. It is used to determine whether a cell has references to stale symbols.

Initialized variable analysis[1] can be used to efficiently determine which cells to resolve staleness.

Cell Oriented Analysis

The authors again state a few definitions required for the evaluation -

Live symbols: Given a cell c and some symbol s , we say that s is live in c if there exists some execution path in c in which the value of s at the start of c 's execution is used later in c .

Dead symbols: Given a cell c and some symbol s , we say that s is dead in c if, by the time control reaches the end of c , every possible path of execution in c overwrites s in a manner independent of the current value of s . Denoting such symbols as $DEAD(c)$,

Stale cells: A cell c is called stale if there exists some $s \in LIVE(c)$ such that s is stale.

Fresh symbols: Given a cell c and some symbol s , we say that s is fresh w.r.t. c if (i) s is not stale, and (ii) $ts(s) > ts(c)$

Fresh cells: A cell c is called fresh if it (i) it is not stale, and (ii) it contains a live reference to one or more fresh symbols; that is, $\exists s \in LIVE(c)$ such that s is fresh with respect to c .

Refresher cells: A non-stale cell c_r is called refresher if there exists some other stale cell c_s such that $STALE(c_s) - STALE(c_r \oplus c_s) = \emptyset$ where $c_r \oplus c_s$ denotes the concatenation. That is, the result of merging c_r and c_s together has fewer live stale symbol references than does c_s alone.

Initialized Variable Analysis can be thought of as the inverse of liveness analysis. The authors call liveness analysis a backwards-may technique that computes symbols whose values may be used in a cell. Whereas, initialized analysis is a forwards-must technique that computes symbols that will be definitely assigned by the time control reaches the end of a cell.

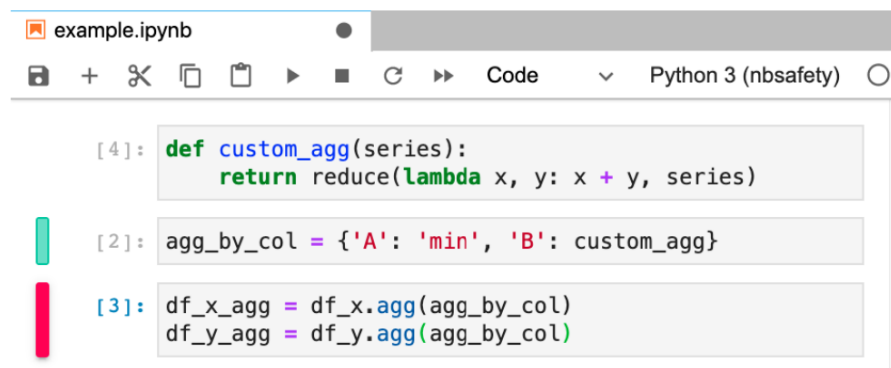
Cell Highlights

As described in the paper, highlighted cells are a subset of the entire cell set that have a relation among themselves in a particular point in time.

This subset is conformed by the set of stale cells, the set of fresh cells and the set of refresher cells. Also there are more subsets that come from these subsets, like the new fresh cells and the new refresher cells.

All these subsets will later be highlighted respectively in the interface depending on their content. This highlight will appear always at the left side of the cell. All the stale cells will show a staleness warning, and the refresher and fresh cells will show a cleanup suggestion warning.

The colors of the warnings will also prompt the user to re-execute the refresher cells in order to recover the expected state of the notebook and to avoid the cells that are stale.



```
[4]: def custom_agg(series):  
      return reduce(lambda x, y: x + y, series)  
  
[2]: agg_by_col = {'A': 'min', 'B': custom_agg}  
  
[3]: df_x_agg = df_x.agg(agg_by_col)  
      df_y_agg = df_y.agg(agg_by_col)
```

The computation of the highlighted sets is trivial because we just need to run the liveness checker to get the metadata for each symbol and determine its staleness or freshness.

However, computing refresher cells is a little more complicated, and so we need to find a way to do it efficiently. Checking the brute force algorithm, we would require $O(n^2)$ liveness analysis.

Thanks to theorem 1, described in the paper, we can compute the refresher cells by using cells that are considered dead. With that we just need to perform $O(n)$ liveness analysis and $O(n)$ initialized variable analysis.

Predictive Power Results

Given these definitions,

\mathcal{H}_s , the set of stale cells in a notebook;

\mathcal{H}_f , the set of fresh cells in a notebook; and

\mathcal{H}_r , the set of refresher cells in a notebook.

$\Delta\mathcal{H}_f^{(t)} = \mathcal{H}_f^{(t)} - \mathcal{H}_f^{(t-1)}$ (new fresh cells); and

$\Delta\mathcal{H}_r^{(t)} = \mathcal{H}_r^{(t)} - \mathcal{H}_r^{(t-1)}$ (new refresher cells)

\mathcal{H}_n , or the *next cell highlight*, which contains only the $k+1$ cell (when applicable) if cell k was the previous cell executed; and

\mathcal{H}_{rnd} , or the *random cell highlight*, which simply picks a random cell from the list of existing cells.

NBSAFETY discovered that 117 sessions out of the 666 encountered staleness issues at some point. Positive highlights like \mathcal{H}_f and \mathcal{H}_r correlated strongly with user choices.

The average $P(\mathcal{H}_*)$ (predictive power of highlighted cells) is summarized as follows:

Quantity	\mathcal{H}_n	\mathcal{H}_{rnd}	\mathcal{H}_s	\mathcal{H}_f	\mathcal{H}_r	$\Delta\mathcal{H}_f$	$\Delta\mathcal{H}_r$
$\text{AVG}(P(\mathcal{H}_*))$	2.64	1.02	0.30	2.83	3.90	9.17	6.20
$\text{AVG}(\mathcal{H}_*)$	1.00	1.00	2.71	3.73	2.31	1.73	1.81

$\Delta\mathcal{H}_f$ had the highest predictive power, while \mathcal{H}_s had the lowest predictive power suggesting that users do, in fact, avoid stale cells. Users were over $3\times$ less likely to re-execute stale cells than they are to re-execute randomly selected highlights of the same size as \mathcal{H}_s — strongly supporting the hypothesis that users tend to avoid stale cells.

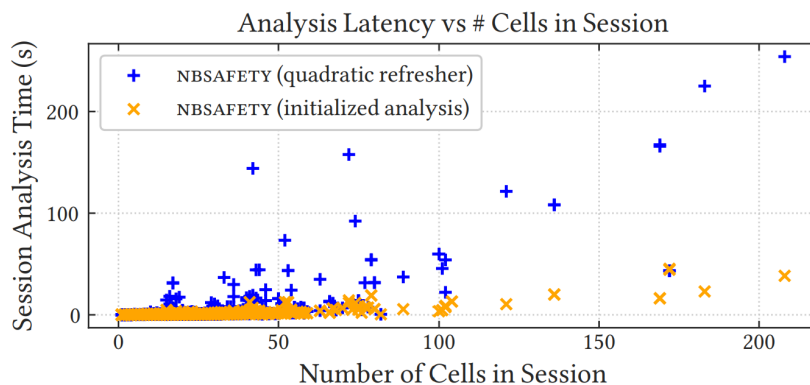
Whereas all other highlights satisfied $P(\mathcal{H}_*) > 1$ on average, and $P(\Delta\mathcal{H}_f)$ larger than the others, suggesting that users are more than $9\times$ more likely to select newly fresh cells to re-execute.

Furthermore, none of the values was larger than 4 on average, suggesting that cues are useful, not overwhelming.

Of the 666 sessions replayed, 1 or more safety issues were detected (due to the user executing a stale cell) in 117. In sessions with safety errors, users were more likely to select the next cell (H_n), and less likely to select fresh or refresher cells.

Benchmark Summary

NBSAFETY introduces additional overhead which is within the same order-of-magnitude as vanilla Jupyter, taking less than $2\times$ longer to replay all 666 sessions, with typical slowdowns less than $1.5\times$. Without initialized analysis for refresher computation, the total reply time increased to more than $3\times$ the time taken by the vanilla Jupyter kernel. The benefit of using initialized analysis for efficient computation of refresher cells, was measured by the latency of just nbsafety’s analysis component, and for each session, they plotted the time versus the total number of cells created in the session.



Unacceptable per-cell latencies were observed for notebooks with more than 50 cells. The linear variant that leverages initialized analysis, however, scales nicely for large notebooks in our execution logs.

Related Work

Evaluating NBSAFETY they found that it does correlate with other notebook systems, NBSAFETY being the first system that preserves existing any-order execution semantics. Having surveyed

Conclusion

The authors of the paper described NBSAFETY – an integrated tracer, checker and frontend for existing Jupyter workflows. This innovative tool allows for efficient detection and correction of unsafe notebook interactions while minimizing the library specific knowledge users are required to have for successful use. NBSAFETY was shown to scale gracefully to any-sized notebooks and successfully reduced error-proneness which was demonstrated on a corpus of real users.

The authors of the paper plan to work to include interactions other than staleness in future iterations of the project.

References

- [1] [Macke, S., Gong, H., Lee, D. J. L., Head, A., Xin, D., & Parameswaran, A. (2021). Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment*, 14(6), 1093–1101. <https://doi.org/10.14778/3447689.3447712>

Related Work:

<https://par.nsf.gov/servlets/purl/10129817>

<https://dl.acm.org/doi/pdf/10.1145/3290605.3300500>

<https://multithreaded.stitchfix.com/blog/2017/07/26/notebook/>

https://austinenley.com/pubs/Chattopadhyay2020CHI_NotebookPainpoints.pdf