Wrattler: Making notebooks reproducible, live and smart

Tomas Petricek
The Alan Turing Institute
London, UK
tomas@tomasp.net

Charles Sutton
University of Edinburgh
Edinburgh, UK
csutton@inf.ed.ac.uk

James Geddes
The Alan Turing Institute
London, UK
jgeddes@turing.ac.uk

Abstract

Notebook systems such as Jupyter became a popular programming environment for data science, because they support interactive data exploration and provide a convenient way of interleaving code, comments and visualizations. However, notebooks also suffer from reproducibility issues and make versioning and provenance tracking difficult.

In this paper, we present Wrattler, a new notebook system with an architecture that addresses the above issues. Wrattler stores all state in a data store and separates state management from script evaluation. This makes it possible to support versioning, guarantee reproducibility, track data provenance, but also allow richer forms of interactivity and integrate AI tools that help automate routine data wrangling tasks.

Keywords notebook, dependency graph, live coding, AI

1 Introduction

Data science is an iterative, exploratory process that requires a collaboration between a computer system and a human. Notebook systems support this interaction model by making it easy to run snippets of code and see results without leaving the notebook. However, typical notebook systems, such as Jupyter suffer from a number of problems:

Limited reproducibility. Cells can be executed out-of-order or modified without re-evaluating the notebook. This means that running all cells from scratch may give different results than those originally seen in the notebook.

Opaque state management. All state is managed by the execution engine, or *kernel*. This makes the state invisible to tools such as version control systems and makes it hard to track provenance in a notebook.

Limited interaction. Notebooks execute code at the granularity of an entire cell. This means that even simple code change may trigger a long-running computation and, as a result, notebooks do not always provide rapid feedback.

Single language. When using a notebook, the state is managed by a single *kernel*. This makes it difficult to combine multiple programming languages in a single notebook, or build third-party tools that would inspect the state and provide data analysts with hints about data.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

In this paper, we present Wrattler, a new notebook system that addresses the above issues. This is made possible by two key aspects of the Wrattler architecture (Section 2).

First, Wrattler maintains a dependency graph between cells and often also function calls inside a single cell (Section 3.2). When a cell is changed, relevant parts of a graph are invalidated. This guarantees reproducibility and it enables more efficient re-computation, because values of unaffected nodes can be reused. Second, Wrattler separates state management from code execution (Section 3.3). The *data store* is responsible for managing state. It handles versioning, provenance tracking and enables multiple independent languages and tools to operate on the state of a notebook.

Together, these two changes to the standard architecture of notebook systems make Wrattler notebooks (Section 4) reproducible (with an easy and reliable state rollback), live (with efficient re-computation on change) polyglot and smart (allowing the integration of multiple execution engines and third-party AI tools).

2 Wrattler architecture

In standard notebook systems, such as Jupyter [5], the state and execution is handled by a *kernel*. The notebook running in a browser sends commands to the kernel to evaluate cells selected by the user. As illustrated in Figure 1, Wrattler splits the server functionality between the following components:

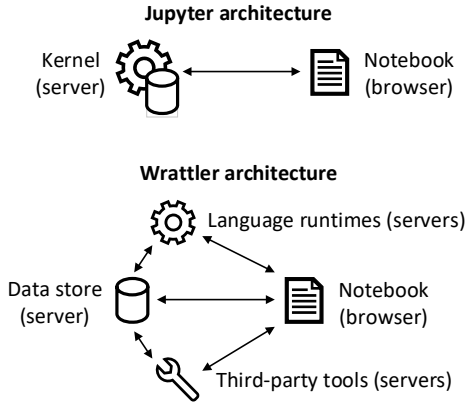

Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Data in the data store can be also accessed by other thrd-party tools.

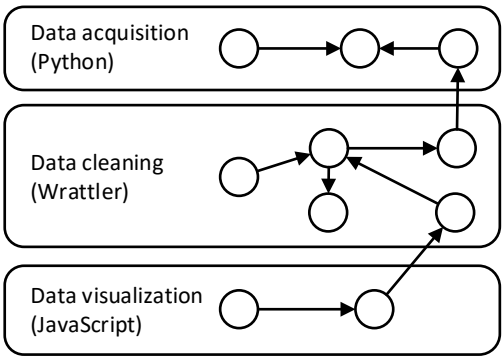

Figure 2. Dependency graph of a sample notebook with three cells. The cells are written in Python, Wrattler script and JavaScript. The browser builds a fine-grained dependency graph for Wrattler script that can be parsed and analysed in the browser.

Data store. Imported external data, results of running scripts and of running third-party tools are stored in the data store. The data store keeps version history and annotates data with metadata such as types, inferred semantics and provenance.

Language runtimes. Code in notebook cells is evaluated by language runtimes. The runtimes read input data from and write results back to the data store. Wrattler supports language runtimes that run code on the server (similar to Jupyter kernels), but also language runtimes that parse and evaluate code directly in the browser.

Third-party tools. In addition to language runtimes, other tools can contribute to data analysis by analysing data in the data store. In Wrattler, this includes AI assistants that help, for example, with data cleaning, type inference and data extraction from raw input data.

Notebook. The notebook is displayed in a web browser and orchestrates all other components. The browser builds a dependency graph between cells or individual expressions in the cells. It calls language runtimes to evaluate code that has changed, third-party tools to provide hints and reads data from the data store to display results.

3 Wrattler components

Wrattler consists of a notebook user interface running in the web browser, which communicates with a number of server-side components including language runtimes, the data store and third-party tools. In this section, we discuss the components in more detail.

3.1 TheGamma script

The Wrattler architecture supports languages that require running a separate language runtime as a server, but also languages that can be fully evaluated in the browser. To illustrate this feature, we integrated Wrattler with TheGamma [10], a simple browser-based language for data exploration.

As an example, we use a data set with broadband speed published by the UK government. The following TheGamma script loads the raw CSV file, groups rows by whether they represent rural or urban area and then calculates average download speed for both of the kinds of areas:

```
web.loadTable("https://.../bb2014.csv").explore.
  'group data'.'by Urban/rural'.
    'average Download speed (Mbit/s) 24 hrs'
```

Here, identifiers such as 'by Urban/rural' are generated by a *type provider* [14] and the user typically interacts by selecting one of generated members through an auto-complete (hence the long explicity names are acceptable). For the purpose of this paper, the most important aspect is that TheGamma scripts can be parsed, checked and evaluated in the browser.

3.2 Dependency graph

When opening a notebook, Wrattler parses the source of the notebook (consisting of text cells and code cells) and constructs a dependency graph, which serves as a runtime representation of the notebook.

As an example, consider a simpler notebook with three cells. The first cell, written in Python, downloads data and exports the result as a data frame; the second cell is written in TheGamma script and performs data cleaning and the third cell creates a visualization in JavaScript. The resulting dependency graph is shown in Figure 2.

Wrattler creates at least two nodes for each cell, representing the cell as a whole and its source code. In addition, it creates a node for each data frame exported by a cell (e.g. the rightmost node in the first cell). Code in subsequent cells can depend on data frames exported from earlier cell, which is captured as a dependency in the graph.

For code snippets that can be analysed in the browser (such as TheGamma cells), Wrattler creates a more fine-grained dependency graph with a node for each operation (such as member access or a function call). This is the case in the second cell in Figure 2 and it allows more efficient live update of results when code changes as we only need to re-evaluate parts of code in a cell. We briefly outline how the dependency graph is constructed and used.

Dependency graph construction. The dependency graph is reconstructed after every change in any of the cells. This is an efficient process, because the changes are typically small. Wrattler first parses each cell, producing a single syntactic element for Python, JavaScript and R cells (representing the entire source code) and a syntax tree for TheGamma (representing individual member accesses and other constructs). The syntax tree for the entire notebook is then a list of elements produced for each cell.

After parsing, Wrattler walks over the syntax tree recursively and binds a dependency graph node to each syntactic element in the tree. The *antecedents* of a node are the nodes

that it depends on. This typically includes inputs for an operation or instance on which an operation is invoked. The binding procedure is decribed in Figure 3.

Checking and evaluation. Nodes in the dependency graph can be annotated with information such as the evaluated value of the syntactic element that the node represents. An important property of the binding process is that, if there is no change in antecedents of a node, binding will return the same node as before. This means that if we evaluate a value for a given node in a dependency graph and attach it to a node, the value will be cached and reused.

Wrattler does not automatically re-evalaute the entire dependency graph, but allows the user to trigger re-evaluation. This provides a user experience familiar to other notebook systems. However, the displayed results and visualizations always reflect the current source code in the notebook. When the evaluation of a cell is requested, Wrattler recursively evaluates all the antecedents of the node and then evaluates the value of the node.

The evaluation is delegated to a language runtime associated with the language of the node and it can be done in several ways:

1. For Python or R nodes, the language runtime ensures that the values of the dependencies are stored in the data store and sends the source code, together with its dependencies, to a server that retrieves the dependencies and evaluates the code.
2. For TheGamma and JavaScript nodes, the language runtime collects values of the dependencies and runs the operation that the node represents in the browser.

Wrattler also uses the dependency graph for type checking. Just like evaluation, type checking is done by recursively walking over the dependency graph and annotating the nodes with their type. This step is only relevant for cells written in statically typed languages and provide quicker feedback, because it does not need to evaluate the cells.

3.3 Data store

The data store enables communication between individual Wrattler components and provides a way for persistently stroing data. The data stored in the data store is associated with the hash produced by the binding process outlined in Figure 3 and is immutable. When the notebook changes, new nodes with new hashes are created and so the new results are appended, rather than overwriting existing state.

The data store in Wrattler (currently) supports two data formats – external data files imported into Wrattler notebooks (such as downloaded web pages) and data frames. The latter are used for communication between multiple cells of a Wrattler notebook. A cell that exports a data frame so that it can be used by subsequent cells needs to store the data frame into a data store (the key is the hash of the node representing the data frame).

```
procedure bind(cache, syn) =
  let h = {hash(kind(syn))}
        ∪ {hash(c) | c ∈ antecedents(syn)}
  if not contains(cache, h) then
    let n = fresh node
    value(n) ← Unevaluated
    hash(n) ← h
    set(cache, h, n)
  lookup(cache, h)
```

Figure 3. When binding a graph node to a syntactic element, Wrattler first computes a set of hashes that uniquely represent the node. This includes hash of the kind of the node (e.g. the source code of a Python node or member name in TheGamma) and hashes of all antecedents. If a node with a given hash does not exist in cache, a new node is created. We set its hash, indicate that its value has not been evaluated and add it to the cache.

The data store additionally also supports a mechanism for annotating data frames with additional semantic information. Columns can be annotated with (and stored as) primitive data type such as date or floating-point number. Columns can also be annotated with semantic annotation that indicates the meaning of the column – for example, address or longitude and latitude. Finally, columns, rows and individual cells of the data frame can be annotated with other metadata such as their provenance.

In addition to storing the raw data, the data store also persistently stores the current and multiple past versions of the dependency graph constructed from the notebook (saved by an explicit checkpoint). This makes it possible to analyse the history of a notebook and track how data is transformed by the computation in a notebook.

3.4 Third-party tools

The Wrattler architecture enables integration of third-party tools that analyse the notebook or data it uses and provide hints to the data analyst. Most notably, such tools include AI assistants that help with tedious data cleaning and preprocessing tasks.

As an example, datadiff [13] is an AI assistant that reads two specified data frames from the data store and suggests a script that transforms the data in the second data frame to match the format of the first data frame. The AI assistant does not automatically transofrm the data, but instead, produces a new cell in TheGamma script that can be reviewed by the data analyst. For example:

```
datadiff(broadband2014, broadband2015)
  .drop_column("WT_national")
  .drop_column("WT_ISP")
  .recode_column("URBAN", [1, 2], ["Urban", "Rural"])
```

The script specifies that two columns should be dropped from the corrupted second data frame and one column needs to be recoded (turning 1 and 2 into strings Urban and Rural).

4 Properties of Wrattler

The Wrattler architecture outlined in Section 2 leads to a number of properties that are difficult to obtain with traditional notebook systems. In this section, we discuss the properties and briefly discuss our prototype implementation.

4.1 Reproducible, live and smart

The two most important aspects of the Wrattler architecture are that it separates the state from the language runtime (using a data store) and that it keeps a dependency graph based on the current notebook source code (on the client). This enables a number of properties.

Reproducibility. The evaluation outputs displayed in Wrattler notebook always reflect the current source code. When code changes, Wrattler updates the dependency graph and hides invalidated visualizations. Because the data store caches earlier results, it is always possible to go back without re-evaluating the whole notebook.

Provenance. The graph allows us to track data provenance. For code written in TheGamma script, we can track provenance at the level of individual variables and operations; for R, Python or JavaScript, we can currently only track provenance at cell level. This also enables implementing refactoring of notebooks, for example, to extract only code necessary to produce a given visualization.

Interactivity. Finally, the use of dependency graph makes it possible to update displayed results more efficiently, because only values for new nodes in the graph need to be calculated. The fine-grained structure of the dependency graph for TheGamma script makes it possible to often show live previews instantly.

Polyglot. Sharing the state via data store makes it possible to combine multiple langauge runtimes, as long as they support sharing data via data frames. In our prototype, this includes R, JavaScript and TheGamma script, but the extensibility model allows adding further languages.

Smart and explainable. The data store makes it possible to build third-party components, that analyze data in the notebook and provide automatic suggestion, such as how to merge entities across multiple data sets. Furthermore, the data store does not allow direct modification of the data, so such tools result in code that explains the operation and can be reviewed by the data analyst.

4.2 Wrattler prototype

A prototype implementation of Wrattler is available on GitHub (http://github.com/wrattler). The prototype implements language runtimes for R, JavaScript and TheGamma script (Section 3.1). It builds a dependency graph (Section 3.2) and uses it to evaluate results of cells. The data store (Section 3.3) stores data in Microsoft Azure in JSON format. Support for meta-data annotations and big data is not yet implemented. The protoype comes with one third-party tool – the datadiff AI assistant mentioned above (Section 3.3). Figure 4 shows the first cell of a sample Wrattler notebook.

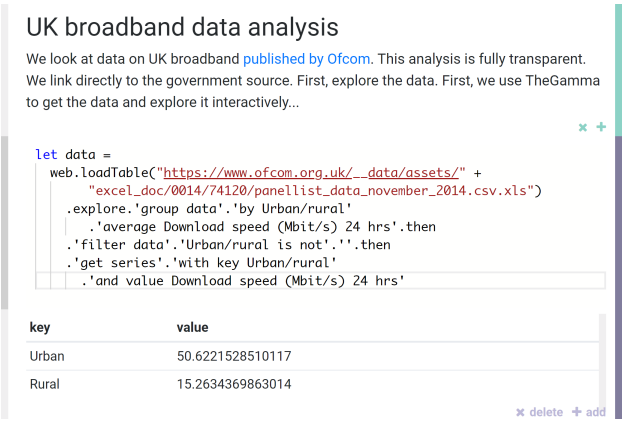

Figure 4. TheGamma script that downloads and aggregates UK government data, running in Wrattler notebook with a live preview.

5 Related work

The work in this paper directly follows the work on IPython and Jupyter notebook systems [11][5]. Wrattler shares many properties with those and aims to address some of their limitations. To address reproducibility, some Jupyter extensions and systems such as R markdown [1] lock cells after evaluation. Dataflow notebooks [6] are more sophisticated and attach unique hashes to cell evaluations. Scientific workflow systems [2, 9] manage evaluation of workflows over a dependency graph similarly to Wrattler, but often allow editing it directly via a GUI, rather than through code in a notebook.

Our method of dependency grpah construction is inspired by Roslyn [8] and extends our earlier work on TheGamma [10]. It is simiar to methods used in live programming languages [4, 7], as well as incremental compilation [12] and partial evaluation [3]. Wrattler adapts those methods to a notebook-like data science environment.

6 Summary

This paper presents early work on Wrattler – a new notebook system for data science that makes notebooks reproducible, live and smart. The properties of Wrattler follow from two changes to the standard architecture of notebook systems.

First, Wrattler separates the state management from code execution. This allows provenance tracking, polyglot notebooks and integration of third-party tools that can work directly with the data store. Second, Wrattler keeps a dependency graph on the client (web browser) and uses it to control evaluation. This guarantees reproducibility and allows faster feedback during development.

References

[1] J Allaire, Joe Cheng, Yihui Xie, Jonathan McPherson, Winston Chang, Jeff Allen, Hadley Wickham, Aron Atkins, Rob Hyndman, and R Arslan. 2016. rmarkdown: Dynamic Documents for R. *R package version* 1 (2016), 9010.

[2] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 423–424.

[3] Olivier Danvy. 1999. Type-directed partial evaluation. In *Partial Evaluation*. Springer, 367–411.

[4] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. *ACM SIGPLAN Notices* 40, 10 (2005), 505–518.

[5] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a format for reproducible computational workflows.. In *ELPUB*. 87–90.

[6] David Koop and Jay Patel. 2017. Dataflow Notebooks: Encoding and Tracking Dependencies of Cells. In *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*. USENIX Association.

[7] Sean McDirmid. 2007. Living it up with a live programming language. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 623–638.

[8] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. 2011. The Roslyn Project, Exposing the C# and VB compilerâĂŹs code analysis. *White paper, Microsoft* (2011).

[9] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. 2004. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20, 17 (2004), 3045–3054.

[10] Tomas Petricek. 2017. Data exploration through dot-driven development. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] M Ragan-Kelley, F Perez, B Granger, T Kluyver, P Ivanov, J Frederic, and M Bussonnier. 2014. The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication.. In *AGU Fall Meeting Abstracts*.

[12] Mayer D Schwartz, Norman M Delisle, and Vimal S Begwani. 1984. Incremental compilation in Magpie. *ACM SIGPlan Notices* 19, 6 (1984), 122–131.

[13] Charles Sutton, Tim Hobson, James Geddes, and Rich Caruana. 2018. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. (2018).

[14] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. 2013. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 workshop on Data driven functional programming*. ACM, 1–4.