Wrattler: Reproducible, live and polyglot notebooks

Tomas Petricek
The Alan Turing Institute
tomas@tomasp.net

Charles Sutton*
The University of Edinburgh
csutton@inf.ed.ac.uk

James Geddes
The Alan Turing Institute
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, most notebook systems use an architecture that makes reproducibility and versioning difficult and limits the interaction model.

In this paper, we present Wrattler, a new notebook system built around provenance that addresses the above issues. Wrattler separates state management from script evaluation and controls the evaluation using a dependency graph maintained in the web browser. This allows richer forms of interactivity, an efficient evaluation through caching, guarantees reproducibility and makes it possible to support versioning.

Keywords notebook, dependency graph, live coding, AI

1 Introduction

Notebooks [5, 14] are literate programming [6] develoment environments that allow data analyst to combine text, code and code outputs. Notebooks make it easy to run code and see results. To further aid reproducible, iterative and exploratory data science, notebook systems should also support:

Richer interaction model. Web browsers are increasingly powerful and allow moving parts of data exploration to the client-side. Simple code changes should be evaluated efficiently using a cache and notebooks should give live previews when writing code to perform simple data exploration.

Transparent state management. The state maintained by a notebook should be transparent and accessible to external tools. This would allow versioning of state and development of tools to explore and manipulate the notebook state.

Multiple languages and tools. A notebook should make it easy to combine multiple programming languages. A cell written in one language should be able to automatically access data frames defined in other languages.

Improved reproducibility. Changing code in a cell should invalidate results that depend on data frames defined in the cell. Reverting a change should immediately revert the result to the previous one and show it immediately using a cache.

*Also with The Alan Turing Institute and Google.

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

In this paper, we present Wrattler, a new notebook system that supports the above features. We follow a line of work combining provenance tracking with notebooks [7, 13], but rather than extending existing systems, we revisit two key aspects of the notebook architecture (Section 2).

First, Wrattler uses a *dependency graph* to track provenance between cells and even function calls inside individual cells (Section 3.2). When a cell is changed, relevant parts of a graph are invalidated. This guarantees reproducibility and enables more efficient re-computation. Second, Wrattler introduces a *data store* that separates state management from code execution (Section 3.3). The data store handles versioning and simplifies the support for polyglot programming.

Together, these two changes to the standard architecture of notebook systems make Wrattler notebooks (Section 4) polyglot, reproducible (with an easy and reliable state rollback) and live (with efficient re-computation on change that enables live preview during data exploration).

2 Wrattler architecture

Standard notebook architecture consists of a *notebook* and a *kernel*. The kernel runs on a server, evaluates code snippets and maintains state they use. Notebook runs in a browser and sends commands to the kernel in order to evaluate cells selected by the user. As illustrated in Figure 1, Wrattler splits the server functionality between two components:

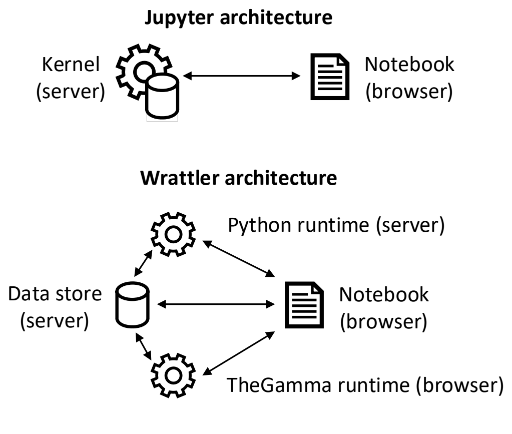


Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Language runtimes can run on the server-side (e.g. Python) or client-side (e.g. TheGamma).

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

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.

Notebook. The notebook is displayed in a web browser and orchestrates all other components. The browser builds a dependency graph between cells or individual calls. It invokes language runtimes to evaluate code that has changed, 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 the data store and multiple server-side language runtimes. We also provide an integration with a simple data exploration language TheGamma [12] that has a fully client-side language runtime.

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 parsed and evaluated in the browser. To illustrate this, we integrated Wrattler with TheGamma [12], a simple browser-based language for data exploration.

As an example, we use a data set with broadband speed published by the UK government [10]. The following script written using TheGamma loads the online CSV file, groups rows by whether they represent rural or urban area and then calculates average download speed for both area kinds:

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

Identifiers such as 'by Urban/rural' are generated by a *type provider* [17] 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. The parsing of code cells is done by language runtimes that parse and statically analyse code and constructs parts of the dependency graph.

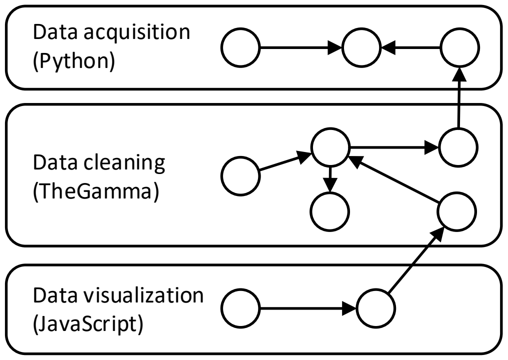


Figure 2. Dependency graph of a sample notebook with three cells. The cells are written in Python, Wrattler script and JavaScript. The browser builds a fine-grained dependency graph for Wrattler script that can be parsed and analysed in the browser.

Wrattler uses a *data frame* as a primary format for passing data between cells. Data frames represent tables with multiple (named) columns and (anonymous) rows. Data frames have a first-class support in most data science languages.

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 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.

The dependency graph is updated after each code change and so it needs to be constructed efficiently in the browser. Language runtime that run on the client-side create 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. This allows live update of results when code changes as we only need to re-evaluate small parts of code in a cell.

Dependency graph construction. The dependency graph is updated after every code change. Wrattler first parses each cell, producing a single syntactic element for Python and JavaScript cells (representing the entire source code) and a syntax tree for TheGamma (representing 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 using a process decribed in Figure 3. The *antecedents* of a node are the nodes that it depends on. This typically includes inputs for an operation or instance on which a member access is performed.

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 (Figure 3) 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 re-evaluates parts of the dependency graph on demand and 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:

1. For Python nodes, the language runtime 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, but it provides a quick feedback, because it does not need to evaluate the code.

3.3 Data store

The data store enables communication between individual Wrattler components and provides a way for persistently stroing input data. It stores raw input data, data frames and multiple versions of the dependency graph. Data frames stored in the data store are associated with the hash produced by the binding process outlined in Figure 3 and are immutable. When the notebook changes, new nodes with new hashes are created. New data frames are appended, rather than overwriting existing state.

External input data files imported into Wrattler notebooks (such as downloaded web pages) are stored as binary blobs. Data frames are currently stored in JSON format (as an array of records), but we intend to use a suitable database system in the future. Data frames are used for communication between multiple cells of a Wrattler notebook. To make a data frame available to subsequent cells, a cell can export and store a data frame into the data store (the key is the hash of the node representing the data frame).

The data store also supports a mechanism for annotating data frames with semantic information. Columns can be annotated with primitive data type such as date or floating-point number. Columns can 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 data source.

```
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.

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 pre-processing tasks.

As an example, datadiff [16] 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). The provenance information that is available thanks to this arhcitecture enable 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.

Refactoring. The dependency graph allows us to implement notebook refactoring. For example, it is possible to extract only code necessary to produce a given visualization. For code written in TheGamma, this extracts individual operations; for Python or JavaScript, we can currently extract code at cell-level granularity.

Live previews. The dependency graph makes it possible to give live previews during development. When code changes, only values for new nodes in the graph need to be calculated. The fine-grained structure of the dependency graph for TheGamma makes it possible to update 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.

4.2 Wrattler prototype

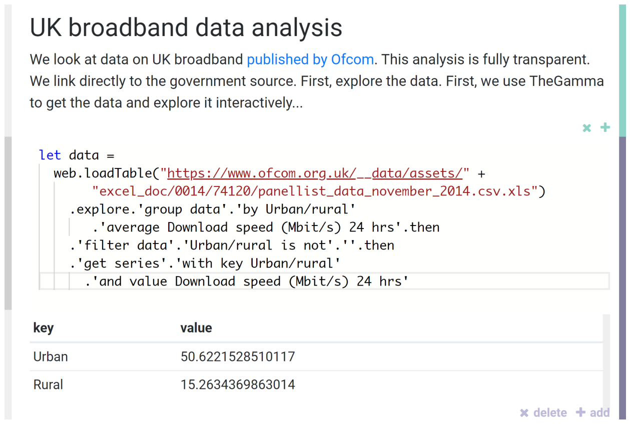
A prototype implementation of the Wrattler system is available on GitHub (http://github.com/wrattler). The prototype implements language runtimes for JavaScript, TheGamma script and R (Section 3.1). It builds a dependency graph (Section 3.2) and uses it to evaluate results of cells.

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 analysing the UK broadband data.

5 Related work

The work in this paper directly follows the work on IPython and Jupyter systems [5, 14]. 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 [7] are more sophisticated and attach unique hashes to cell evaluations. This allows the user to



UK broadband data analysis

We look at data on UK broadband published by Ofcom. This analysis is fully transparent. We link directly to the government source. First, explore the data. First, we use TheGamma to get the data and explore it interactively...

```
let data =
  web.loadTable("https://www.ofcom.org.uk/__data/assets/" +
      "excel_doc/0014/74120/panellist_data_november_2014.csv.xls")
    .explore.'group data'.'by Urban/rural'
      .'average Download speed (Mbit/s) 24 hrs'.then
    .'filter data'.'Urban/rural is not'.''.then
    .'get series'.'with key Urban/rural'
     .'and value Download speed (Mbit/s) 24 hrs'
```

key	value
Urban	50.6221528510117
Rural	15.2634369863014

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

refer to dependencies explicitly and, in effect, construct a dependecy graph similar to ours manually. Scientific workflow systems [2, 11] 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. The noWorkflow project [13] links the two approaches by instrumenting Jupyter kernel with a mechanism for capturing provenance based on light-weight code annotations.

Our method of dependency grpah construction is inspired by Roslyn [9] and extends our earlier work on TheGamma [12]. It is simiar to methods used in live programming languages [4, 8], as well as incremental compilation [15] 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 polyglot. The properties of Wrattler are enabled by provenance information that is maintained thanks to two changes to the standard architecture of notebook systems.

First, Wrattler separates the state management from code execution. This allows versioning, 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.

Acknowledgments

We thank to our colleagues from the AIDA (Artificial Intelligence for data analytics) project Chris Williams, Zoubin Ghahramani and Ian Horrocks and also numerous attendees of a recent AIDA workshop in The Alan Turing Institute. This work was in part supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1.

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] Donald Ervin Knuth. 1992. *Literate programming*. Center for the Study of Language and Information Stanford.

[7] 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.

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

[9] 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).

[10] Ofcom. 2018. Open data. Available online at https://www.ofcom.org.uk/research-and-data/data/opendata. (2018).

[11] 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.

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

[13] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. 2015. Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow. In *Workshop on the Theory and Practice of Provenance (TaPP), Edinburgh, Scotland*. 155–167.

[14] 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*.

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

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

[17] 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.