Wrattler: Reproducible, live and polyglot notebooks

Tomas Petricek
University of Kent
The Alan Turing Institute
tomas@tomasp.net

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

Charles Sutton
The University of Edinburgh
The Alan Turing Institute and Google
csutton@inf.ed.ac.uk

Abstract

Notebooks such as Jupyter became a popular environment for data science, because they support interactive data exploration and provide a convenient way of interleaving code, comments and visualizations. Alas, most notebook systems use an architecture that leads to a limited model of interaction and makes reproducibility and versioning difficult.

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.

1 Introduction

Notebooks [5, 15] are literate programming [6] systems that allow interleaving text, code and outputs. To aid reproducible, exploratory data science, notebook systems should provide:

Richer interaction model. Web browsers are increasingly powerful and allow moving parts of data exploration to the client-side. Notebooks should leverage this and 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 that provide hints based on 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.

Supporting these is a challenge that combines several research areas. We need programming language techniques to efficiently update live previews during editing, provenance methods to track data dependencies and data representation that can be shared across langauges.

Wrattler is a new notebook system that supports the above features. We follow a line of work combining provenance tracking with notebooks [7, 14], but rather than extending existing systems, we revisit two core 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 3.1). 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 3.2). 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:

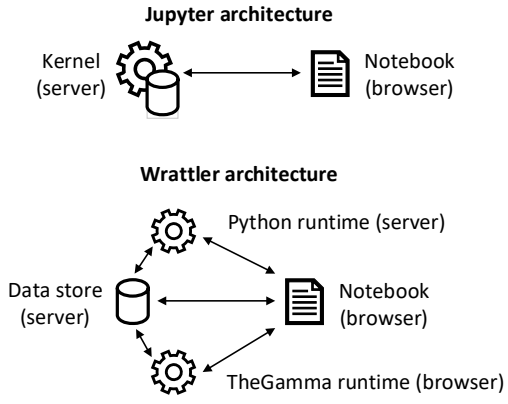

Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Language runtimes can run on the server-side (e.g. Python) or client-side (e.g. TheGamma).

TaPP 2018, July 11–12, 2018, London, UK

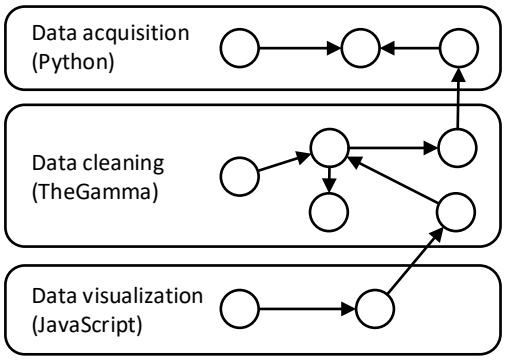

Figure 2. Dependency graph of a sample notebook: The first (Python) cell downloads data and exports the result as a data frame; the second (TheGamma) cell performs data cleaning and the third (JavaScript) cell creates a visualization. Language runtime for TheGamma runs in the browser and creates a fine-grained graph (which allows an efficient live previews), while Python and JavaScript runtimes create just one node for the whole source code.

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), but also browser-based langauge runtimes.

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 runs in the web browser and communicates with the data store and language runtimes that may run on the server or in the browser. An example of browser-based language runtime is TheGamma [12], discussed in Section 3.3.

3.1 Dependency graph

At runtime, Wrattler maintains a dependency graph that is composed from sub-graphs created by individual language runtimes. The graph is acyclic and a node can only depend on earlier nodes. Each node has a value which may be:

- A language-specific value that Wrattler does not understand. This is kept in the browser and re-computed when the notebook is re-opened.
- A data frame. Data frame is stored in the data store and browser keeps a reference (URL) of the frame. This is understood by all language runtimes and provides a way of exchanging data between multiple languages.

Figure 2 shows a sample dependency graph. Wrattler creates two nodes for each cell (representing the cell and its source code) and a node for each data frame exported by a cell (e.g. the rightmost node in the first cell). Nodes in subsequent cells may depend on data frames exported by earlier cells.

Wrattler treats data frames in a special way. They are stored in data store and each langauge runtime is responsible for loading them into a native language representation (e.g. pandas in Python and array of records in JavaScript).

Dependency graph construction. The dependency graph is updated after every code change. Wrattler invokes individual langauge runtimes to parse each cell. Language runtimes that run in the browser (e.g. TheGamma) produce a fine-grained syntax tree. The result of parsing the whole notebook is then a list of elements obtained for each cell.

Wrattler then walks over the syntax tree and binds a dependency graph node to each syntactic element 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. As a result, previously evaluated values attached to nodes in the graph are 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.

3.2 Data store

The data store enables communication between individual Wrattler components and provides a way for persistently storing input data. 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 and appended to the data store. This means that language runtimes can cache them and avoid fetching data from data store each time they need to evaluate a code snippet.

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

External inputs imported into Wrattler notebooks (such as downloaded web pages) are stored as binary blobs. Data frames are stored in JSON format (as an array of records), but we intend to use a suitable database in the future. During the binding process (Section 3.1), a langauge runtime identifies imported and exported data frames for each cell (e.g. by static analysis of the code). Those are then represented as hashes (keys) referring to a location in the data store.

The data store also supports a mechanism for annotating data frames with semantic information. Columns can be annotated with primitive data types (date, floating-point number) and semantic annotation indicating their meaning (address or longitude and latitude). Columns, rows and individual cells of the data frame can also be annotated with custom metadata such as their data source or accuracy.

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.

3.3 TheGamma script

The Wrattler architecture supports 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.

The Figure 4 shows TheGamma cell in Wrattler during editing. The example uses broadband speed data published by the UK government [10] and calculates average download speed in urban and rural areas, respectively. TheGamma supports a rich interactive model in two ways:

- The script is evaluated on-the-fly during editing and a live preview is shown (below the code editor).
- All code can be written using autocomplete that offers available members (representing aggregation operations). Rather than writing code, user repeatedly selects one of the offered members (which are provided by a type provider [18] running in the browser).

For the purpose of this paper, the most important aspect of TheGamma is that scripts can be parsed and evaluated in the browser. This allows more interactive style of data exploration without round-trips to re-evaluate modified code.

4 Properties of Wrattler

The Wrattler architecture outlined in Section 2 allowed us to develop a prototype system with a number of properties that are difficult to obtain with traditional notebooks.

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 language 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 TheGamma script (Section 3.3), R and JavaScript. It builds a dependency graph (Section 3.1) and uses it to evaluate results of cells.

The data store (Section 3.2) stores data in Microsoft Azure in JSON format. Support for meta-data annotations and big data is not yet implemented. Storing notebook state in the data store also allowed us to develop an integration with Data diff [17], which provides data cleaning recommendations.

```
web.loadTable("https://ofcom.org.uk/.../bb2014.csv")
  .explore.'group data'.'by Urban/rural'
    .'average Download speed (Mbit/s) 24 hrs'
    .av
      average Download speed (Mbit/s) 8-10pm weekday
      average Download speed (Mbit/s) Max
      average Headline speed
      average Id
      average Market
      average Upload speed (Mbit/s)24-hour
      average Upload speed (Mbit/s)Max
```

Urban/rural	Download speed (Mbit/s) 24 hrs
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.

5 Related work

The work in this paper directly follows the work on IPython and Jupyter systems [5, 15]. 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 lock cells after evaluation.

Dataflow notebooks [7] attach unique hashes to cell evaluations. This allows the user to refer to dependencies explicitly and, in effect, construct a dependecy graph manually. Scientific workflow systems [1, 11] manage evaluation over a dependency graph similarly to Wrattler, but allow editing it directly via a GUI, rather than through code in a notebook.

The noWorkflow project [14] links the two approaches by instrumenting Jupyter kernel with a mechanism for capturing provenance based on light-weight annotations. Vizer [4] focuses on integrating notebooks with spreadsheet-like interface. It internally uses a data store component similar to ours, but does not keep dependency graph on the client.

Our binding process is inspired by Roslyn [9] and extends an earlier work on TheGamma [13]. It is simiar to methods used in live programming languages [3, 8], incremental compilation [16] and partial evaluation [2]. Wrattler adapts those methods to a notebook 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

The authors would like to acknowledge the funding provided by the UK Government's Defence & Security Programme in support of the Alan Turing Institute and the EPSRC grant EP/N510129/1. We thank to our colleagues Chris Williams, Zoubin Ghahramani and Ian Horrocks and attendees of a recent AIDA project workshop.

References

[1] 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*. IEEE, 423–424.

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

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

[4] Juliana Freire, Boris Glavic, Oliver Kennedy, and Heiko Mueller. 2016. The Exception That Improves the Rule. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. ACM, 7:1–7:6.

[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 compiler's 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 *Proceedings of ECOOP*, Vol. 74. Schloss Dagstuhl.

[13] Tomas Petricek. 2018. Design and implementation of a live coding environment for data science. Unpublished draft. Available online at http://tomasp.net/academic/drafts/live. (2018).

[14] 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)*. 155–167.

[15] 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*.

[16] Mayer D Schwartz, Norman M Delisle, and V S Begwani. 1984. Incremental compilation in Magpie. *SIGPLAN Not.* 19, 6 (1984), 122–131.

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

[18] 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 DDFP*. ACM, 1–4.