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, 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:

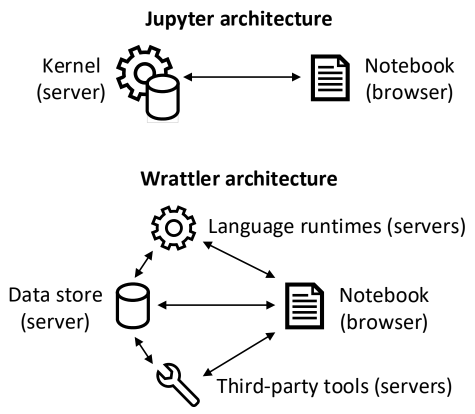

Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Data in the data store can be also accessed by other thrd-party tools.

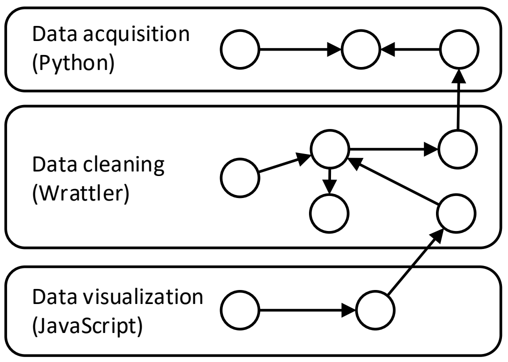

Figure 2. Dependency graph of a sample notebook with three cells. The cells are written in Python, Wrattler script and JavaScript. The browser builds a fine-grained dependency graph for Wrattler script that can be parsed and analysed in the browser.

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 [X], a simple browser-based language for data exploration.

As an example, we use a data set with broadband speed published by the UK government [X]. 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* [X] 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($c$) | $c \in$ 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* 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 Results

Properties. The dependency graph enables several features that are difficult for most notebook systems:

- When code changes, Wrattler only needs to recompute a small part of the graph. This makes it possible to provide live previews, especially for simple data analytical DSLs.
- Refactoring can extract parts of the notebook that are needed to compute a specified output.
- Refactoring can translate nodes of an analytical DSLs (generated by an AI assistant) into R or Python.
- The graph can be used for other analyses and refactorings, such as provenance tracking or code cleanup.

Finally, the script is represented in a fine-grained way (second cell in Figure 2). Once it is interactively constructed, it can be translated to multiple supported languages such as R, Python or JavaScript.

5 Related work

dataflow notebooks

Data science is an iterative, exploratory process that requires a collaboration between a computer system and a human. A computer can provide advice based on statistical analysis of the data and discover hidden structures or corner cases, but only a human can decide what those mean. A data science environment of the future thus needs to support an interaction model that keeps the human in the loop, allows an efficient human-computer interaction and can be extended with new AI asistants that provide advice about data. In this report, we present Wrattler, a notebook system that is:

Interactive. Wrattler enables an efficient interaction by bringing computation closer to the human. Notebooks run in the browser, cache partial results of computations and provide previews of script results on-the-fly during development.

Reproducible. Wrattler separates the task of running scripts from the task of managing state. A data store tracks the provenance and semantics of data, supports versioning and keeps the history, making the data analyses fully reproducible.

Polyglot. Multiple languages can be used in a single notebook and share data via the data store. Analysts can use R and Python, but also interactive languages for data exploration that run in the browser and provide live previews.

Smart. Wrattler serves as a platform for AI assistants that use machine learning to provide suggestions about data. Such AI assistants connect to the data store to infer types and meaning of data, provide help with data cleaning and joining, but also help data exploration by finding typical and atypical data points and automatically visualizing data.

6 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 AI assistants. In this section, we discuss the components in more detail, starting with the dependency graph that is maintained on the client-side, by the web browser.

6.1 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 structure is ilustrated in Figure 2. Top-level nodes (squares) represent individual notebook cells. Code (circles) is represented either as a single node (R, Python) or as a sub-graph with node for each operation (DSLs understood by Wrattler). Any node of a cell can depend on a data frame (hexagons) exported by an earlier cell. When code in a cell changes, Wrattler updates the dependency graph, keeping existing nodes for entities that were unaffected by the change.

The dependency graph enables several features that are difficult for most notebook systems:

- When code changes, Wrattler only needs to recompute small part of the graph. This makes it possible to provide live previews, especially for simple data analytical DSLs.
- Refactoring can extract parts of the notebook that are needed to compute a specified output.
- Refactoring can translate nodes of an analytical DSLs (generated by an AI assistant) into R or Python.
- The graph can be used for other analyses and refactorings, such as provenance tracking or code cleanup.

6.2 AI assistants