Tow Center for Digital Journalism
A Tow/Knight Report

MUCK: A BUILD TOOL FOR DATA JOURNALISTS

GEORGE KING

Columbia Journalism School

Funded by the Tow Foundation
and the John S. and James L. Knight Foundation

Acknowledgments

I would like to thank the Tow Center for sponsoring my research, as well as Mark Hansen and Claire Wardle, whose support made this work possible. Additionally, I am grateful to Gabe Stein for his efforts during the early phases of the project and his encouragement throughout.

February 2018

Contents

Executive Summary

Veracity and reproducibility are vital qualities for any data journalism project. As computational investigations become more complex and time consuming, the effort required to maintain correctness of code and conclusions increases dramatically. This report presents Muck, a new tool for organizing and reliably reproducing data computations. Muck is a command line program that plays the role of the build system in traditional software development, except that instead of being used to compile code into executable applications, it runs data processing scripts to produce output documents (e.g., data visualizations or tables of statistical results). In essence, it automates the task of executing a series of computational steps to produce an updated product. The system supports a variety of languages, formats, and tools, and draws upon well-established Unix software conventions.

A great deal of data journalism work can be characterized as a process of deriving data from original sources. Muck models such work as a graph of computational steps and uses this model to update results efficiently whenever the inputs or code change. This algorithmic approach relieves programmers from having to constantly worry about the dependency relationships between various parts of a project. At the same time, Muck encourages programmers to organize their code into modular scripts, which can make the code more readable for a collaborating group. The system relies on a naming convention to connect scripts to their outputs, and automatically infers the dependency graph from these implied relationships. Thus, unlike more traditional build systems, Muck requires no configuration files, which makes altering the structure of a project less onerous.

Muck's development was motivated by conversations with working data journalists and students. This report describes the rationale for building a new tool, its compelling features, and preliminary experience testing it with several demonstration projects. Muck has proven successful for a variety of use cases, but work remains to be done on documentation, compatibility, and testing. The long-term goal of the project is to provide a simple, language-agnostic tool that allows journalists to better develop and maintain ambitious data projects.

Key findings

- Building completely reproducible data journalism projects from scratch is difficult, primarily because of messy input data. Practitioners often have to take many steps to clean data, and may use a variety of computational tools in a single project.
- Muck effectively solves this problem by formalizing the step-by-step approach in a way that aids, rather than hinders, the programmer. The system is language-agnostic and encourages the programmer to break down their complex data processing problems into well-named constituent parts, which can then be solved piece by piece. The resulting code structure clearly describes the relationships between parts, which improves the clarity of the solution.
- Not all data cleaning tasks can be clearly expressed as code. In particular, there is a class of "one-off" corrections like missing commas and correcting obvious outliers for which programmatic solutions appear very cryptic upon review. Muck provides a mechanism for capturing and reproducing manual edits to data files as "patches," which is crucial for cases where automated approaches are ineffective. Patching techniques are particularly compelling for journalism because so many original data sources are very messy. Furthermore, the approach may offer a means by which non-programmers can efficiently contribute to a substantial data cleaning job, while maintaining the automated reproducibility of the overall project.
- A new system for professionals needs to be compatible with the wide range of existing tools that practitioners know and trust. In practice, no program can be all things to all people. Unlike data tools that present non-programmers with graphical interfaces, Muck is a command line tool developed specifically for programmers who are already familiar with one or more languages, as well as Unix operating system fundamentals. This makes Muck unsuitable for complete beginners, but also means that it is much more easily integrated into professional environments.

Introduction

Journalism increasingly relies on programming tools for the acquisition, analysis, and presentation of data. Most of this technology originates from other industries, but perhaps the specialization of data journalism as a field warrants specialized software. What kinds of new tools could help data journalists write their code more effectively?

Journalists work on tight deadlines, where veracity is of the utmost importance. They must repeatedly verify their conclusions from messy data as they work toward publication. Editors often do not possess the technical expertise to understand internal complexities, but nevertheless need to review the work in progress. Requirements and input data can change substantially during the course of a project. From the software engineer's perspective, these challenges are familiar but exaggerated by journalism's short timeframes, the relative lack of technical expertise across the newsroom, and the wide variety of subject matter.

The precise nature of programming makes it very error-prone. Because mistakes abound, the ability to reproduce a computation swiftly and accurately is crucial. One way the software industry mitigates this type of challenge is to use a "build system": a program that automatically rebuilds the complete product from its various sources whenever changes are made, so that the latest version is always available. Build systems benefit the development process by:

- ensuring that all participants can reproduce the product from original sources reliably and quickly
- automating potentially tedious build steps
- allowing programmers to take a more modular, disciplined approach to the work, where they might otherwise be tempted to take shortcuts for the sake of expediency

Muck is a build tool that helps data journalists make their data projects reproducible. Using it can improve the correctness and clarity of program logic, as well as the overall speed of project development. Data cleaning, manipulation, and formatting tasks are treated as computational steps, each of which produces one or more files as output. Muck's main role is to run these steps in the correct order, on demand. While Muck is not a tool for data cleaning or visualization specifically, it can be used to facilitate

such jobs. This kind of workflow only makes sense for projects that can be broken down into multiple steps, but the approach is nevertheless useful for small jobs.

A typical data journalism project consists of inputs (e.g., datasets and collections of files, either stored locally or downloaded from the network); outputs (e.g., graphics, html, or text files); and transformations from inputs to outputs (e.g., programs, scripts, and shell commands), possibly involving intermediate outputs. Unlike a web application that is constantly serving user requests, a data project tends to have relatively fixed inputs and outputs. Unlike traditional build systems, Muck is designed specifically to accommodate the structure of data-oriented projects.

Projects built with Muck require no special configuration files; the relationships between files are inferred automatically from file names and the references within files. To build a specified product Muck works backwards, finding the necessary dependencies and running each script individually. This approach facilitates a tight, iterative process that avoids recomputing steps for which the inputs have not changed, while guaranteeing that steps whose inputs have changed will be recomputed. Muck also reduces programming errors by encouraging the programmer to break code down into more modular scripts. In essence, the tool is a solution for developing and consistently reproducing or "compiling" documents. With a single command, anyone possessing a project's code can reconstruct the final products, whether they are single files or whole collections of web pages containing embedded charts, individual graphics, and calculated statistics.

This report describes the motivations for building Muck, the basic features of the tool, and preliminary experiences building demonstration projects. Muck is currently useful for general-purpose data work, so long as datasets and intermediate results fit on the local disk. However, there is still much work to be done, particularly on documentation and improving ease of use. The core program is quite small and is designed to remain simple as it grows to support various programming languages and data formats. The system should also be useful for a wide variety of work beyond data journalism, including static website generation, technical documentation and blogging, data science, computational humanities projects, and

more—anywhere that the inputs and outputs can be framed as discrete sets of items.

Background and Motivation

Muck was born from a research question posed by Mark Hansen, a professor at the Columbia University Graduate School of Journalism: What would a language for data journalism look like? To explore this question, we hosted a conversation with working data journalists in the fall of 2015.[1] The discussion revealed several priorities for us to focus on:

- storydriven process
- strong support for iterative (and often messy) data transformation
- reproducibility

Additionally, there was strong demand for a system that would be comprehensible not just by professional programmers but by less technical journalists as well.

Input from Practitioners

Our guests agreed that their best data stories started with a question, and often from reporters who don't work on the data team. Several prominent examples[2] [3] encouraged us to focus on helping journalists construct data analyses to answer their questions, rather than on tools for open-ended data exploration. For these kinds of investigative stories, we have heard repeatedly that data cleaning and manipulation are usually the limiting factor.[4] [5]

Further complicating this process, data journalists often use different tools to process and analyze data than they do to produce the published story or interactive application. This makes the analysis less accessible to nontechnical collaborators, because there are multiple systems involved, each requiring expert knowledge. Complexity can also make it harder for the data journalist to stay focused on the larger story while analyzing data. One participant advocated for using JavaScript throughout the entire analysis and publication process, but in general data journalists use a variety of languages and tools; Python, R, SQL, JavaScript, Shell, and Excel are all in wide use.

This technology landscape is hardly unique to data journalism. Each of the languages just mentioned has been inherited from the much larger field of data science, and of those, Python, SQL, JavaScript, and Bash are

considered general-purpose languages. As statistical analysis is becoming more and more integrated into all kinds of industries, a preference for more general programming languages is emerging.[i] [6]

The Benefits of Well-Structured Code

Several participants in our discussions noted that programming in a deadline-driven environment can force data journalists into "writing spaghetti code until it works." But skipping the refactoring efforts necessary to untangle poorly structured code often leads to practical problems later on when checking and verifying a story. Participants mentioned strategies for verifying their results: formal sanity checks on their projects to look for red flags (but rarely comprehensive code reviews due to time constraints); journals documenting how teams arrived at a given result; and reproducibility exercises from journals, with result "checkpoints" to help with the verification steps.

From this discussion, two major shortcomings to these approaches emerged. The first is that editors outside of the data team rarely check conclusions, because the rest of the newsroom usually lacks the required analytical knowledge or the ability to read code.[ii] The disconnect between data journalists and traditional journalists makes verification expensive and time-consuming. For large, high-impact stories, WNYC performs full code reviews, and ProPublica brings in outside experts to validate conclusions. In every example raised, data journalists faced pressure to complete projects on a deadline and their credibility (as well as that of their publications) rested on the accuracy of the results.

The second shortcoming of existing methods is that unless the record of modifications to the data is perfect, auditing work from end to end is impossible. Small manual fixes to bad data are almost always necessary; such

i. The most immediate difference in discussions of technology in data journalism compared to those in the data science industry is that the hype around "big data" systems is mercifully absent.

ii. While we cannot expect the average editor to become code-literate overnight, good project structure does make a dramatic difference for novices. The key calculations that deserve the most review are often quite simply stated in the code—it is finding them amid all the supporting code that is frustrating.

transformations often take place in spreadsheets or interactive programming sessions and are not recorded anywhere in the versioned code. Several participants expressed concerns with tracking and managing data. These problems are compounded by the need for collaboration among teammates whose technical abilities vary.

Content management systems for sharing and versioning documents have existed for decades, but as Sarah Cohen of *The New York Times* put it, "No matter what we do, at the end of a project the data is always in thirty versions of an Excel spreadsheet that got emailed back and forth, and the copy desk has to sort it all out . . . It's what people know." Cohen's observation speaks to a common thread running through all of our discussion topics: the deadline-driven nature of the newsroom makes it a challenging environment in which to introduce new technology. Participants cautioned against assuming that clever software would easily solve such multifaceted problems, or underestimating the challenge of learning a new system when team members have such diverse skill sets.

Pedagogy

Hansen also posed a second, more concrete question focused on education: What would be the equivalent of "Processing" (a programming environment for visual and multimedia programming) for data journalism? From the project overview:

> Initially created to serve as a software sketchbook and to teach programming fundamentals within a visual context, Processing has also evolved into a development tool for professionals.[7]

Currently, interactive notebook environments like Jupyter[8] play this role (without the novel language) in data science education. These are browser-based systems that let the user write or edit small blocks of code and then run them immediately, with the result of each block conveniently rendered in an interleaved style. This workflow has an immediate appeal when exploring data, but from the perspective of language and tool design, the notebook approach is problematic. The presentation is completely linear, even when the actual code structure is not. Worse, beginners struggle to manage the statefulness of the notebook, often forgetting to rerun code

blocks as they make changes. Lastly, by working inside the notebook students gain no experience with more traditional text editor applications, which are vital tools of the trade.

Both the Processing and Jupyter environments have been successful as teaching tools in part because they allow the novice programmer to write a simple program (as little as a single line of code) and get visual results immediately. This convenience assumes a very specific modality: not only are the outputs primarily visual, but the act of writing code is presumed to occur inside of the unified environment. In other words, both systems are specialized for visual, interactive programming in a way that favors the beginner's need for simplicity over the expert's need for generality.

Direction

The original premise of the project was to explore a language design dedicated to data journalism, and our initial conversations explored traditional angles on programming language design: beginner-friendly syntax and semantics; removal of quirky, historical baggage; robust error detection and reporting; constraints like assertions;[iii] type systems;[iv] and immutability.[v]

iii. An assertion is a statement in a program that states a condition assumed by the programmer to be true; if at runtime the condition fails, then the program terminates rather than continue under faulty conditions.

iv. Types are essentially formal annotations on code that denote which kinds of values (e.g., strings or integers) are valid in a given context (e.g., a variable or function parameter). A "type checker" is a program (either a standalone tool or part of a compiler) that analyzes a piece of code to verify that all of the type constraints are properly met.

v. A "mutable" value is one that can be altered in-place; an "immutable" value cannot. Most imperative languages allow mutating operations with few constraints. They typically also require the programmer to follow certain immutability rules without fully enforcing them. For example, Python's dictionary type requires that key values be immutable with respect to their `hash` value; otherwise lookups will fail silently. It is chiefly for this reason that the builtin string and tuple types are immutable. Nonetheless, one can easily construct a custom class type that acts as a key, but also allows mutation of its hash value. The problem is analogous to opening a file cabinet of employee records sorted by name, and changing an employee's name in their record without re-sorting it properly. In contrast, so-called "purely functional" languages prohibit all mutation in the imperative sense: "pure" code cannot alter a value in place, only create a copy with alterations. The functional paradigm is gaining popularity but can be very challenging to learn, particularly for programmers already trained in the imperative style. Many new language designs (e.g., Rust and Swift) attempt a hybrid approach.

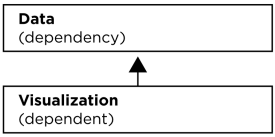
But improving on the state of the art requires a tremendous amount of work. Martin Odersky, a leader in the field of type systems and language design, estimates that designing a good typed language is a multi-person, decade-long effort.[9] Nonetheless, many groups are working on new languages. "Julia" is a recent, successful example in this vein: it is a language designed for the modern data scientist, balancing concerns between correctness, ease of use, and performance.[10]

There are other, wider angles that we considered as well. For example, we spent some time looking at the history of "literate programming," which shifts focus to documenting intent. Donald Knuth (its inventor) describes the paradigm eloquently: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."[11] This is an intriguing approach, but is perhaps easier to justify for sophisticated algorithms than for jobs on a deadline. We can aspire to document our code, but data journalists (as well as most programmers in general) cannot always make it their top priority.

From there we considered a "document-centric" view of data programming, in which the effect of the code is to produce a document. This bears some similarity to literate programming but does away with the latter's dual goals of producing working programs and formatted documentation simultaneously. Instead, we simply narrow our scope to problems in which we are producing some final document. This focus excludes large swathes of conventional software (servers, interactive applications, etc.), but accurately describes the basic goal of data journalism, as well as many other endeavors that fall under the data science umbrella.

Muck: An Overview

Broadly speaking, we characterize the basic programming task of data journalism as a transformation of input data into some output document. For clarity and reproducibility, such transformations should be decomposed into simple, logical steps. Any instance where one step refers to another as a prerequisite is termed a "dependency"; whenever the contents of a dependency changes, its dependent is considered stale or out of date. A whole project can then be described as a network of steps, where each node is a file and the links between them are the dependencies, in the form of references by file name. In computer science this network structure is called a directed acyclic graph, and a graph representing dependency relationships is a dependency graph. A dependency graph is "acyclic" because there can be no circular relationships: it makes no sense to say that a file depends on itself before it can be used, neither directly nor indirectly.



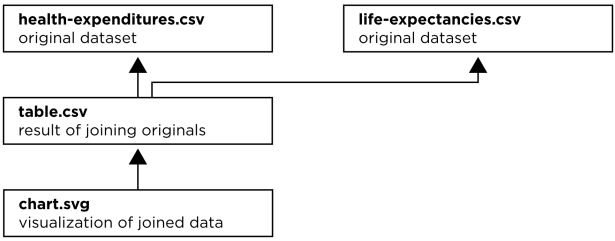
A simple dependency graph: note how the arrow points in the opposite direction of information flow. We choose to draw dependencies as pointing from dependent to dependency because that best represents the notion of a reference: the visualization "knows the name" of the data, but the data has no intrinsic knowledge of or need for the visualization.

The concept of the dependency graph is key to several classes of programming tools, including compilers, reactive and asynchronous programming frameworks (e.g., OCaml async[12]), and build systems. Muck is a build system for data projects. As projects get more complex, the value of taking a modular view of the work increases. This makes describing the system with compelling yet succinct examples something of a challenge!

A Motivating Example

Suppose we want to produce an article like this recent post on *The Guardian* Datablog, which shows life expectancy versus private health care spending per individual across wealthy countries.[13] (A complete walkthrough that reproduces the article using Muck can be found here: https://github.com/gwk/muck-demos/tree/master/oecd-health. The data is downloaded from the Organization for Economic Cooperation and Development (OECD), in the form of two CSV (comma separated values) files, and the outputs are a table and a chart. The basic steps needed to create the article are then:

- download the health expenditures dataset
- download the life expectancies dataset
- extract the relevant data points into a table
- render the chart using the rows in the table

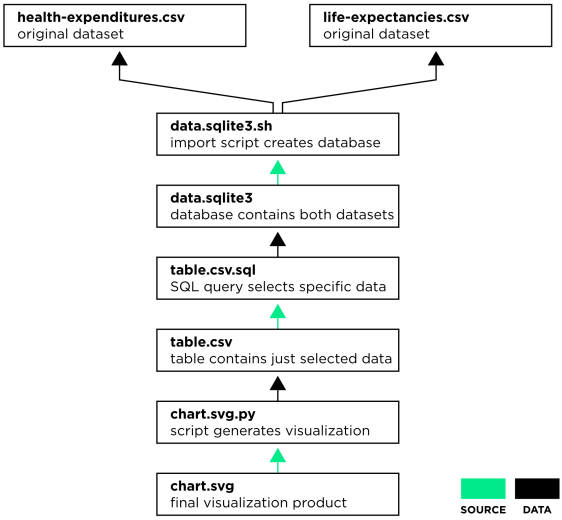


Health costs example: conceptual dependencies.

We could do this work by hand in a spreadsheet, and for simple jobs that is a perfectly reasonable approach. But in this example, the CSV files contain a great deal of extraneous information, and we don't know a priori which rows to select. One good tool for examining structured data like this is SQLite, a free database program that comes preinstalled on MacOS and Linux. If we first load the data as tables into an SQLite database, we can then make investigative queries until we understand which data we

want to select for our table. Once we determine the final query, we will use a Python script to generate the graphic in the SVG (Scalable Vector Graphics) format.

We have now broken our job into several steps. Not including the exploratory queries, our dependency graph looks like this:

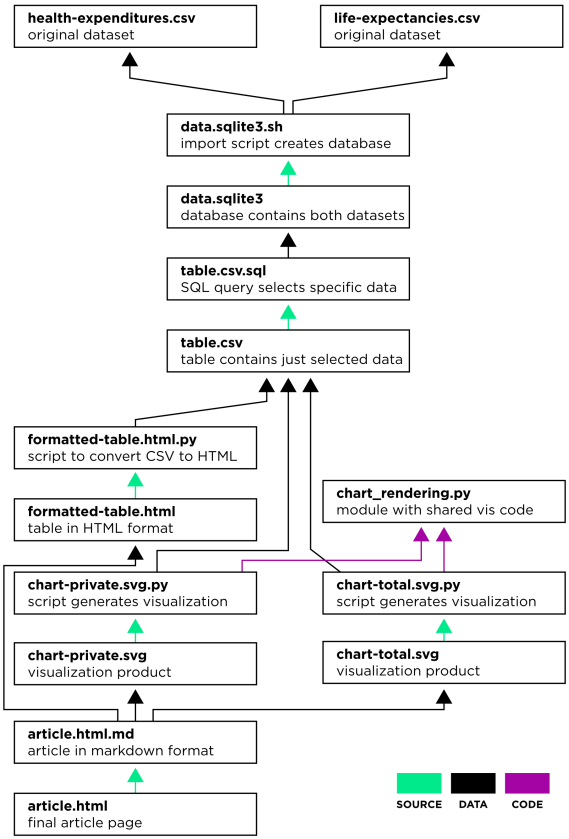


Health costs example: code and data dependencies. "Source" dependencies show that a given product is produced by running the pointed-to source code file. Muck determines these relationships automatically via its naming convention. Each "data" dependency exists because the given source file opens and reads the pointed-to data file. See "Observed and Inferred Relationships" for details on how these dependencies are detected.

The import into the database takes some time, but the downstream steps are instantaneous. A programmer would typically do the import in a shell script by hand on the command line, and then write separate scripts for the table query and the chart rendering (these steps could be combined into a single script, but we chose to write them in different languages). Once we decide to build the product in steps, a subtle challenge emerges: we must remember to update products appropriately as the sources change. This

might sound easy, but as the chains of dependencies get more complex, the opportunities for error multiply.

Let's consider a more elaborate version, where we generate the complete article from a Markdown file, which references the table and two charts ("private expenditures" and "total expenditures"). We also factor out the common code for the charts to a Python module called `chart_rendering.py`.



Health costs example: Muck implementation to build a complete web page. In addition to the source and data dependencies seen in the previous diagram, this implementation also features a code module that is shared by two scripts.

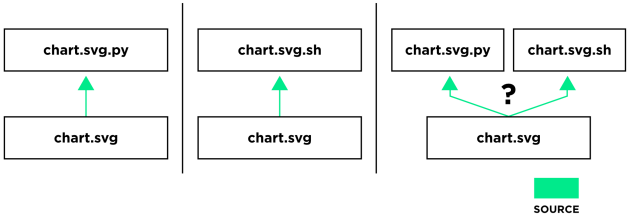
At this point we can appreciate that the relationships between steps are not always linear. For a project of this size, development never really proceeds in a straightforward fashion either—there are several possible starting points, and as we progress we can make changes to any step at any time. We could be working on the final styling of our charts, and then suddenly realize we have a bug in the query. Or perhaps we have been working all week and now that we are done we want to pull the very latest version of the dataset. In the midst of such changes it can be difficult to tell whether or not a given file is actually stale,[vi] and sometimes we just make mistakes. The simplest solution is to rerun everything after any change, but when some steps are slow this is not really feasible. This is exactly where build systems like Muck can help by orchestrating updates both correctly and efficiently.

Technical Description

Muck is invoked from the terminal command line. Given a "target" argument (a file path to be built or updated), Muck first checks to see if the target exists in the project directory (if it does, then no build action needs to be taken). For example, to build our chart we would type: `muck chart.svg`. If the target does not exist in the project source directory, then it is a "product," and Muck tries to find a matching "source" file. The rule is simple: a product's source is any file with the target name, plus an extra file extension. For example, if we wish to build `chart.svg`, a file called `chart.svg.py` (a Python script) could serve as the source, as could `chart.svg.sh` (a shell script). This is the core convention that ties a Muck project together. The source file for any given product must be unambiguous; if both `chart.svg.py` and `chart.svg.sh` are present then Muck will exit with an error.

Once Muck has identified the source, it determines its "static" dependencies (the files that must be present in order to execute the source code),

vi. The "staleness" property is transitive: if A depends on B, and B depends on C, then whenever C is updated, both A and B become stale. This means that a small change to one file can potentially affect many downstream products.



Source file resolution. Either `chart.svg.py` or `chart.svg.sh` is a legitimate source candidate, but if both are present then the source is ambiguous and results in an error.

recursively updates those, and then finally builds the target, updating and recording any "dynamic" dependencies requested by the script as it runs.

For example, here is what the console output looks like when we build our chart for the first time:

Informational output from Muck while building `chart.svg`.

```
muck note: chart.svg: building: ‘python3 chart.svg
    .py‘
muck note: data.sqlite3: building: ‘sh data.
    sqlite3.sh‘
muck note: data.sqlite3: finished: 61.93 seconds (
    via tmp).
muck note: data.sqlite3: product is new; 101.970
    MB.
muck note: table.csv: building: ‘sqlite3 < table.
    csv.sql‘
muck note: table.csv: finished: 1.66 seconds (via
    stdout).
muck note: table.csv: product is new; 1.169 kB.
muck note: chart.svg: finished: 0.12 seconds (via
    stdout).
muck note: chart.svg: product is new; 6.066 kB.
```

Ignoring the slightly convoluted order of build operations for the moment,[vii] we can see that the command caused three products to be built. If we run the same command again, Muck detects that nothing has changed, and so there is no work to be done. When we make a change to one of the sources, Muck will update only those products that are downstream—that is, the products that depend recursively (meaning either directly or indirectly) on the changed file.

Inferred and Observed Relationships

Like most traditional build systems, Muck operates on a file-by-file basis; it has no ability to reason about fine-grained structures like individual functions or rows of data.[viii] However, unlike traditional build systems, Muck determines the dependencies of a given target automatically; there is no "makefile" or other configuration metadata that denotes the relationships between files. While this eliminates the need to explicitly list all required files, it also means that Muck is limited to source languages that it understands, and source code must comply with a few simple requirements.

Muck uses three different techniques to determine dependencies:

- A *Source* dependency is determined by the naming convention (e.g., `chart.svg chart.svg.py`), and denotes a script, markup, or template file, the execution or rendering of which results in some product file (or files).
- An *Inferred* or *Static* dependency is a file (either data or code) referenced by a source in such a way that its name can be detected by Muck prior executing/rendering that source.
- An *Observed* or *Dynamic* dependency is a file (either data or code) opened by a source script during execution and whose name is detected by Muck at the moment the "open" command is issued.

vii. The order of build operations appears interleaved because `table.csv` is an "observed" dependency of `chart.svg`; thus the build process for `chart.svg` blocked while the upstream dependencies were updated. See "Inferred and Observed Relationships."

viii. Dependency inference at the resolution of individual functions, variables, and records seems desirable, but requires much tighter integration with each programming language and thus is beyond the scope of this project. See "Limits of File-Level Dependencies."

Conceptually, we might come up with other categories that make sense for specific languages (e.g., data versus code dependencies as shown in the example diagrams), but Muck's core algorithm uses only these three.

The naming convention for sources is a straightforward formalization of standard practice: Source names should relate to product names. Muck makes the basic purpose of a source script explicit in its name. It specifies the complete product name (including the type extension), plus an extra extension indicating the source's file type. The convention allows Muck to automatically and unambiguously select a single source for a desired product.

Once a source is selected, it is analyzed to extract its dependencies. The exact process is different per source language, and depending on the semantics of the language might not detect all dependencies perfectly.[ix] For example, Python `import` statements are a syntactic feature of the language, and therefore can be detected through static analysis. Conversely, one cannot generally determine which files a Python script opens and reads, because the script might call an external function that does the opening, and the file name to be opened might get passed around through a series of variables. Whenever Muck can detect dependencies ahead of time, it will recursively build those prior to running the source.

The source is then executed, even though it may have dependencies that went undetected. To address this discrepancy, Muck detects whenever the running child process attempts to open a file, blocks it from executing further, updates that dependency (possibly triggering other build processes), and then unblocks the child so that it can open the up-to-date file. Currently, this just-in-time capability works by automatically replacing the `open` system function with a special version. This wrapper function communicates the file name in question to the Muck parent process and then waits for a confirmation before executing the real `open` system call.[x] While

ix. The term "static analysis" comes from the field of compilers and implies any kind of ahead-of-runtime source code analysis that does not actually execute the code outright. Turing-complete languages (a theoretical classification that includes general purpose programming languages) are by definition unpredictable in this regard, so static analysis code behavior cannot be perfectly accurate. For non-executable formats like pure HTML (without JavaScript) though, a static analyzer can accurately extract all dependencies.

x. Muck's version of `open` is injected into the child process using the dynamic linker. On macOS, this is accomplished with the `DYLD_INSERT_LIBRARIES` environment variable;

this trick works for popular tools like Python and Node.js (a version of JavaScript that runs on the command line), it only applies to programs that are dynamically linked to the standard system libraries, and has no effect for statically linked executables. The distinction is quite obscure to the end user, so removing this limitation is a major priority for future work. In the meantime, less than perfect support for languages can be accomplished with simple static analysis.

Outputs

By default, outputting data from a script is simple. Following standard Unix convention, all text written to the standard output stream (`stdout`) is captured as the product for that step. A script can print debugging text separate from the product by using the `stderr` stream. This too follows Unix convention, but depending on the language it requires more code:

A simple python3 script that outputs some HTML and an error message.

```
rom sys import stderrhttps://www.sharelatex.com/
    project/5a39449a225755184fd40964
print('<html><body><p>Text in output document.</p
    ></body></html>')
print('this debug message does not appear in the
    output, file=stderr)
```

If a script needs to create multiple output files then it can simply use the standard `open` function to obtain writable file handles. When a script

Linux uses a similar variable named `LD_PRELOAD`. Muck's `open` looks up a pair of Unix environment variables, `MUCK_DEPS_SEND` and `MUCK_DEPS_RECV`. These are set by the parent `muck` build process, and together provide the child process with a pair of open file descriptors (if they are not set then the script is running as a standalone command and the system `open` is performed immediately). The child writes to the "send" channel the file name it wishes to open. It then reads a confirmation byte back from the parent on the "receive" channel; until Muck writes and flushes that confirmation, the child is blocked from proceeding, per the semantics of the Unix "blocking read" operation. This gives Muck an opportunity to build the dependency in question if it is out of date.

uses this method, it should not write to `stdout`. If it does, Muck raises an ambiguity error. This detection is achieved with the same monitoring of `open` system calls. However, Muck first needs to figure out that the script in question is in fact the source for those targets before running it.

There are a variety of situations where a single script might produce many outputs. For example, suppose we have our CSV table of health expenditures, and we want to split out "private," "public," and "total" expenditures into three separate tables. We can easily do all in one pass:

`expenditures-payer.csv.py` script, which splits the data in `expenditures.csv` into three separate files.

```
import csv
private = csv.writer(open('expenditures-private.
    csv', 'w', newline=''))
public  = csv.writer(open('expenditures-public.csv
    ',  'w', newline=''))
total   = csv.writer(open('expenditures-total.csv'
    ,   'w', newline=''))
for year, private, public, total in load('
    expenditures.csv'):
  private.writerow((year, private))
  public.writerow((year, public))
  total.writerow((year, total))
```

To follow the naming convention, it seems that this script needs three names, one for each output. Instead, we can use Pythons string formatting syntax to indicate a variable in the name, in this case `expenditures-payer.csv.py`.[14] This is like a Unix shell wildcard/glob (`expenditures-*.csv.py`) except that the formatter must be named, and can be constrained with the format syntax, e.g., to contain just numbers, zero-padding, etc. The script's parameterized name tells Muck that it produces any file matching the `expenditures-*.csv` pattern. As soon as the user requests one of those products, the source runs and produces all three CSV files. Because `open`

communicates with Muck, the build process knows that all three files have been produced after a single invocation.

In that example the various product names are hard-coded into the script, but they could also be generated, perhaps from a range of integers or a collection of names. In other cases we want the script's inputs to depend on the name of the output. For example, if we have separate data files for each of the last five years, we might want to generate one chart for each month, one at a time. Muck makes this easy by extracting the strings that match each formatter and passing them as arguments to the script.

Imagine the products are named `chart-2012.svg`, `chart-2013.svg`, etc., and our script is named `chart-year.svg.py`. Here is what happens when the user builds the first chart:

Informational output from Muck while building chart-2012.svg.py. Unlike previous examples, the script's name specifies a named parameter year in braces. Muck identified this script as the source for the requested product using a matching algorithm, and then passes the matching string "2012" as an argument to the script.

```
muck chart-2012.svg.py
muck note: chart-2012.svg: building: 'python3
    chart-{year}.svg.py 2012
```

Looking at the note in the console output, we see that Muck has extracted the matching year "2012" and passed it as an argument to the script. It extracts exactly one argument for each formatter in the source name. This was true in our previous example as well, but that script simply ignored it. For this one, we use the `year` argument to choose which data file to open:

`chart-{year}.svg.py` script that renders a chart for the given yearly data file. The `year` parameter is provided as a command line argument and used to choose the appropriate input file at runtime, creating a dynamic dependency. Because the filename is computed, it would be impossible for Muck to infer it ahead of time.

```
from muck import load
from some_chart_library import render_chart
from sys import argv \# contains the command used
    to invoke this script as an array.

_, year = argv \# unpack the array, ignoring the
    command name at position 0.
with load(year + '.csv') as data: \# use ‘year‘ to
     construct the dependency name.
  render_chart(data) \# this hypothetical function
       would write to stdout.
```

In this script, we use the standard Unix `argv` variable to access the argument and then use it to choose our data file (e.g., `2012.csv`).

Using formatters in file names might seem unconventional, but they communicate concisely what the script intends to produce. Fortunately, Python's format syntax does not conflict with filename restrictions on Mac and Linux, nor do they conflict with the syntax of the popular Bash shell. In summary, command line arguments are specified through the naming convention. This gives the programmer a great deal of flexibility in how they structure their scripts, while maintaining strong dependency relationships that Muck can infer automatically.

Python Client Library: Load, Fetch, and Load_url

As a convenience, Muck provides Python scripts with several utilities. The `load` function calls `open`, and then based on the file's extension dispatches to an appropriate handler (e.g., the standard library `csv.reader` or `json.load`). Compare this traditional implementation to one using Muck's `load`:

Python script using only the Python standard library.

```
import csv
import json
csv_rows = csv.reader(open('a.csv'))
json_object = json.load(open('b.json'))
```

Python script using `muck.load` convenience function.

```
from muck import load
csv_rows = load('a.csv')
json_object = load('b.json')
```

`load` does away with several lines of code in each script. The effect is most noticeable when moving code between scripts or splitting a large script into smaller steps, when the import statements can easily get left out of the refactoring. It also handles several compression formats (gzip, bzip2, xz, and zip) transparently. The result is that loading data into a Python script is very straightforward, requiring only that the Muck library be imported. Muck comes with loaders registered for a variety of common formats, and additional custom loaders can be registered by scripts or library modules at runtime.

Since most data projects involve acquiring data from the internet in some fashion, Muck provides similar conveniences for remote files. To be clear, many data journalism projects involve proprietary or otherwise pri-

vate datasets, but even private data is often accessed via a network. For a project to be reproducible, the first step is for the data to be obtainable in a consistent way. The Muck library provides `fetch` and `load_url` functions analogous to `open` and `load` described above. These functions fetch files at a given URL once and save them locally until the cache is cleared. For data scraping jobs this is especially important, because if a server has usage limits or other anti-scraping measures in place, once these are triggered then work becomes much more difficult. Another benefit of the caching model is that inputs become completely consistent between builds, mitigating the pitfalls of debugging a script when its inputs keep changing. The tradeoff is that the programmer must manually invalidate the cache (by deleting the files in the project's `_fetch` folder) in order to get up-to-date results from an API query.

Another issue that complicates reproducible network fetches is that many servers attempt to block what they perceive as any sort of programmatic access. The simplest implementations simply reject all requests with "user agent" headers that do not match well-known browsers. Muck addresses this by providing a `load_url` function with the option to send a user agent header imitating that of a modern browser; alternatively the programmer can specify any combination of custom headers.

Project Inventory

Muck provides an option for viewing a project's dependency graph in a hierarchical format, which gives the user a quick overview of the project structure that is guaranteed to be correct and up to date.

Output of the `muck deps` command shows a hierarchical listing of dependencies.

```
\$ muck deps chart.svg

chart.svg
chart.svg.py
table.csv
  table.csv.sql
    data.sqlite3
      data.sqlite3.sh
        health-expenditures.csv
        life-expectancies.csv
```

Dependencies can also be printed as a simple list. This provides an accurate means with which to build custom scripts on top of Muck, perhaps to integrate into a larger production pipeline. Dependency listings can also serve as a powerful complement to written documentation.[xi]

Tracking Manual Edits with Patch Files

Reproducible data cleaning does not simply mean doing everything programmatically. Cleaning data effectively often requires a range of manual fixes, which most tools cannot effectively track.

One solution is to use a "patch" or "diff" tool.[15] The idea behind patching is to create a small file containing lines of text, each marked as either "context," "remove," or "add." The context lines are used to unambiguously locate the text to change in the original document; the "remove" lines are present in the original and to be removed; the "add" lines are not present and to be added. This is the basic premise of Git's diff tools, and conceptu-

xi. One obvious addition would be a graphical rendering of the dependency graph, but graph layout is a famously tricky problem, so this would most likely require an external tool like Graphviz.Graphviz - Graph Visualization Software, http://graphviz.org

ally similar to Microsoft Word's "Track Changes" feature. (The distinction between patch and diff is merely colloquial: typically patches are applied to achieve some change, whereas diffs are read to see what has been changed.)

Muck currently uses its own experimental patching format called Pat,[16] because traditional diffs are not designed to be edited. Muck treats patch files as a kind of source file whose sole dependency is the original to be modified. For example, imagine that we have a CSV file `names.csv`, and one of the fields contains an unquoted comma.

The `names.csv` file contains an unquoted comma in the second row: the first name is intended to be "Bobby, Jr" but since the cell contains a comma, CSV readers will interpret that line as having three columns instead of two. Such a cell must be surrounded by quotes in order for the comma to be ignored by the CSV parser.

```
First Name,Last Name
Alice,Anteater
Bobby, Jr,Baboon
Carole,Caterpillar
```

The first name "Bobby, Jr" confuses Python's `csv.reader` and thus breaks the script. Rather than modify the original, we would like to create a new version called `names-fixed.csv`. Detecting and fixing this sort of error programmatically is tricky though, and typically not worth the gymnastics (a real dataset would likely be much longer). Instead, we can ask Muck to create a patch with this command: `muck create-patch names.csv names-fixed.csv`. This creates a new, empty source file `names-fixed.csv.pat`, and a product file `names-fixed.csv` that is identical to the original. We then have two choices.

Our first option is to write the patch by hand that properly quotes the problematic cell, like so:

names-fixed.csv.pat is a patch file that specifies names.csv as its input, followed by "context" lines and then an alteration to the problematic line.

```
pat v0
names.csv
|^
| First Name,Last Name
| Alice,Anteater
- Bobby, Jr,Baboon
+ "Bobby, Jr",Baboon
```

The patch file states the file that it is patching, and then for each alteration shows several lines of context (as many as necessary in order to be unambiguous), followed by removal of the problematic line and insertion of a suitable replacement. Whenever Muck is asked to produce `names-fixed.csv`, it will discover the patch file as the source, which in turn specifies `names.csv` as its sole dependency. `names-fixed.csv` is then produced with the bad line replaced:

The resulting names-fixed.csv product.

```
First Name,Last Name
Alice,Anteater
"Bobby, Jr",Baboon
Carole,Caterpillar
```

The second, recommended option is to simply modify the product file as we see fit, and then have Muck update the patch with the command `muck update-patch names.csv.pat`. Normally, Muck disallows editing of product files via file permissions, but it makes an exception for patch products in order to facilitate this process. As a result, once a patch step has been set up, making fixes is as easy as editing the file, saving it, and asking Muck to

update. Such changes can be accumulated over time into a single patch, or as several patches applied in series.

The advantage of using patches for data cleaning is that they are convenient and self-explanatory: When reviewing a patch, it clearly shows the removed and added lines with context. When a fix is unique, it is much easier to simply change the text than it is to write code that makes the change for you. This becomes even more apparent when returning to code months after it was written; while code often looks more cryptic than we remember it, the intent of a patch is relatively easy to interpret.

Per-Record Transformations

Patching only addresses part of the data cleaning challenge though. While some flaws occur once or a handful of times, others occur thousands of times. Time-efficient strategies for correcting rare versus common flaws tend to be quite different. In early experiments, the diff format proved so convenient when reviewing changes for correctness that it made sense to add similar informational outputs for programmatic cleaning steps as well. The Muck Python library offers a `transform` function which takes in a collection of records (arbitrary text or structured data) and applies a series of transformations to each record in turn. The transformations come in various flavors. Examples include `flag` (apply a predicate function to the record and if it returns `True`, report the record), `drop` (apply a predicate and if it returns `True`, drop the record and report), and `edit` (apply a transformation to the record, and if it makes an alteration report it). For readers familiar with SQL, these operations can be thought of as analogous to `SELECT`, `DELETE`, and `UPDATE` queries, respectively. When a transformation runs, it outputs both the result text and also a diff showing where each transformation was applied.

Document Rendering

Newsrooms typically have their own content management systems (CMS) and processes for getting a graphic properly formatted into a digital article. Nevertheless, it may be useful at times to write a draft (or just an

outline) and then view the computed products composed together with the text. Muck supports HTML, interpreting any relative URLs in a document (links, images, stylesheets, scripts, etc.) as dependencies. But because writing correct HTML by hand is tedious, Muck also supports the popular Markdown format, which translates to simple HTML pages. Here is an example Markdown file that displays a table and two charts along with text:

A markdown file named `index.html.md`, which references the formatted table and charts.

```
# Article Title
This is the introductory text describing our
    health care expenditures investigation.

## The Data
Here is the data used to generate the charts.
<object type="text/html" data="formatted-table.
    html" />

## Private Expenditures
Here is the chart showing life expectancies vs
    private expenditures.
<img src="chart-private.svg" />

## Total Expenditures
Here is a second chart showing life expectancies
    vs total expenditures.
<img src="chart-total.svg" />
```

This Markdown file has three dependencies (the table and two charts). When we build it, Muck updates the charts and renders the Markdown to HTML that can be viewed in a web browser. Since dependency trees can be arbitrarily deep, in principle one could use this approach to build a whole website. While users have no obligation to use Muck in this capacity, it

provides a simple and effective way to publish any sort of computational result as a web page.

Development Web Server

For web pages (including those that execute JavaScript), Muck can run a simple web server during development which treats every request as an update command. In this way, all dependencies are guaranteed to be updated when the page is refreshed.

Prerequisites and System Requirements

Muck is still in development, and most experiments have revolved around the Python language, though it has recently gained some support for SQLite, JavaScript, and Unix shell scripts. For a data journalist to use Muck effectively requires competency in at least one of these languages, as well as familiarity with the terminal, a basic understanding of the Unix file system and the ability to use a separate code editor.

Muck itself is implemented in Python 3.6 and will not run on older versions. It is freely available via Github[17] and the Python Package Index (`pip install muck`). It depends only on a utility library which is developed in tandem with the main tool.

Comparison to Existing Tools

Due to the design of Muck's dependency inference system, I am not aware of any equivalent tools. However, there are meaningful comparisons to be drawn with other systems.

Build Systems

Muck is intended to play a role similar to that of Make,[18] the traditional Unix build tool. Some data journalists have embraced Make as an imperfect means of adding structure to their projects.[19] Much has been written about the limitations of Make,[20][21] and a variety of alternatives have been built over the decades that improve upon the concept and implementation (Cmake,[22] Tup,[23][24] Redo,[25][26] and Ninja[27]). Additionally, many languages provide their own dedicated build systems (Rake,[28] Jake,[29] Shake,[30] Ant,[31] Maven[32] SBT,[33] Leiningen,[34] Boot,[35] and Ocamlbuild,[36] and a few have been designed for data programming specifically (e.g., Drake[37]). However, to my knowledge, all of them operate by executing the build steps specified in some dedicated build script or recipe file. In contrast, Muck requires no "makefile";[38] it figures out the relationships between steps and their sources without additional help from the programmer.

Variants of the original Make offer pattern matching features to reduce the burden of writing makefiles, but these do not address the root of the issue: As long as dependencies are written and not inferred, they will be a source of both tedious work and errors. Some compilers have the ability to generate Make-compatible dependency listings,[39][40] but using such a feature can be confusingly circular. Omitting a single dependency can result in a build process that seems to work for a while and then at some point fails mysteriously—or worse, produces a target incorporating stale data. This is an insidious sort of error because it easily goes unnoticed and can be hard to diagnose. Ironically, Make was created in 1976 in response to these sorts of "disasters."[41]

Another family of tools uses operating system functionality to trace file accesses.[42] [43] [44] Like Muck, these tools create the dependency graph automatically by monitoring script execution, and they provided the inspiration for Muck's "observed dependencies" model. However, they use Linux's

`strace` tool,[45] which is not available on macOS,[xii] and they lack Muck's just-in-time capabilities, as well as the naming convention that connects products to sources automatically.

In any case, while Muck is best described as a build system, for the most part it is meant for a different audience. Data journalists have adopted a variety of languages and tools for doing their work, but there are not yet industry-standard, project-level systems of the sort that Muck seeks to provide.

Interactive Notebooks

Muck is most easily positioned as an alternative to the Jupyter Notebook, a browser-based, interactive programming environment that is popular in data journalism curricula. The interactive notebook model was pioneered by Mathematica in 1988[46] and embodies ideas from the older literate programming paradigm. The Jupyter Project page describes the notebook as used to "create and share documents that contain live code, equations, visualizations and explanatory text." The key distinction compared to Muck is that in the notebook, code forms part of the readable document with the output of each block immediately following. In Muck, the code generates output documents but is distinct from those products.

This distinction is not absolute though. It is possible to turn a Jupyter notebook into a presentable document that omits code,[47] but the core metaphor is the scientist's lab book: it is meant to be read by practitioners who are interested in seeing the implementation and results interwoven together. Conversely, with Markdown the output document does not necessarily contain code, although an author can choose to emulate the notebook style.

Another key difference is that the notebook relies on a fundamentally linear metaphor, which becomes increasingly unwieldy as projects grow in size. Muck embraces the file system tree as a more robust means of organizing large projects, relying on file naming rather than order to dictate the overall structure. This requires more upfront understanding on the

xii. macOS has analogous but different tools called `dtrace` and `ktrace`, but these require root privileges to run, which is inappropriate for general use in a build system.

part of the user; while complete beginners can easily press the run button in a notebook and see results, they are unlikely to guess how Muck works without reading the instructions.

A more technical criticism involves the semantics of how code is run in the notebook. Although the blocks are visually separate, they execute in what amounts to a single namespace, and the user interface executes a single block at a time. This leaves the user responsible for managing the runtime state of the entire notebook in an ad-hoc manner, and when an early block is rerun, it leaves later dependent blocks showing stale output. Mutable state is well recognized as a major source of programming errors for beginners and experts alike[48] and the notebook environment makes the actual semantics of the Python language less clear in this regard. In comparison, Muck runs each named step in a separate process, thereby isolating language-level side effects. The Unix process model is quite simple conceptually, and the environment in which steps run is nearly identical to invoking each script manually on the command line.

Finally, on a more pedagogical note, the convenience of the integrated notebook environment has a downside: It isolates beginners from the complexities and rigor of the command line environment. Jupyter has proven very popular with educators, in no small part because it is easy to set up and affords the user many conveniences, with a built-in editor and graphical support. But narrowing the scope of instruction to this convenient interface means that students are deprived of exposure to the command line, file management tasks, and code editors, all crucial skills for working on larger systems.

To be fair, most of the blame lies with the larger programming ecosystem. Educators reach for friendlier tools because more traditional setups assume too much prerequisite knowledge and are very unforgiving. To make matters worse, the landscape is perpetually changing, so advice on how to resolve issues quickly gets out of date. Muck requires the user to be reasonably competent in a command line environment and does not yet provide any integration into code editors. While a good deal of effort has been expended to add meaningful error messages to the tool, the user must have a good mental model of what dependency graphs are in order to understand how the system works, which is more abstract than the notebook metaphor.

Declarative Languages

Muck can also be compared to the declarative programming paradigm, in which programs are specified without explicitly dictating control flow (this broad category covers relational languages like SQL, purely functional programming languages like Haskell, logic programming languages such as Prolog, and more.[49] Although the individual steps in Muck are typically written in imperative languages like Python, the dependency inference via names is essentially a declarative system. The programmer specifies the relationships between nodes in the graph, and then the dependency algorithm determines the order in which to run the steps. This is analogous to the way in which a relational database like SQLite uses a query planner to reduce the high-level query into low-level instructions to the database engine.

Case Studies

So far Muck has been tested with several substantial demonstration projects. These were not chosen as journalism pieces per se, which would have involved non-technical concerns like timeliness, story, and confidentiality (early collaborative attempts were stymied in part by the need to keep datasets private), but rather as data experiments that had similar technical requirements to exemplary data journalism pieces.

Webster's English Dictionary

The first significant experiment with Muck used the Project Gutenberg text of *Webster's English Dictionary*, 1917 edition. This edition is the last to enter into the public domain, and of some cultural interest.[50] The text was transcribed in 1995[51] and contains many errors, as well as irregularities in the formatting. It contains approximately 4.5 million words of text.

The purpose of the project is to convert the raw text into structured data, and then analyze the relationships between English words in terms of how they are used to define each other by constructing a directed graph of words and their definitions. From the perspective of validating Muck, the goal was to parse the text into a list of structured objects (each representing a word definition), in a completely reproducible way. The basic workflow was to write parsing code with error detection, observe the parser fail on some portion of the text, and then add a fix (to the code, not the original text) to enable the parser to continue. Because the text was so messy I broke the cleanup into stages, where each stage of output was more structured than the previous one (lines of text, then blocks of text per definition, then fully parsed definitions). It quickly became apparent that a purely programmatic approach was not tenable, so I implemented the patching and transformation features described above.

The data-cleaning effort ended up being quite elaborate, but for the purpose of validating Muck's usability, it was quite successful. The structural analysis of the content itself proved to be more challenging and is not yet complete.

Wikipedia Scraping

A second experiment collected sentences from Wikipedia's "Did You Know?" archives. "Did You Know?" is a feature of the Wikipedia main page that "showcases new or expanded articles that are selected through an informal review process."[52] The purpose of the project is to collect factual sentences containing historical content to use for a (not yet implemented) chatbot trivia game. The archive is a series of pages containing sentences starting with the phrase "Did you know . . . " The project uses Textblob,[53] a natural language processing tool, to filter the sentenced based on verb tense. The main lesson learned from this experiment was that Muck needs to accommodate scripts that open arbitrary files, rather than just dependencies that can be statically inferred.

Originally, Muck used only static (syntactic) analysis to determine which files a Python script will open. It did so by walking over the program's abstract syntax tree (AST) and finding all of the function calls to the standard `open` function. This imposed a strict limitations on scripts: In order for Muck to properly detect dependencies, not only was `open` the only safe way of reading in data, but the argument to open was required to be a string literal (a file path written in quotes). Passing a variable or other expression to `open` was disallowed by the analyzer because such expressions cannot be interpreted statically. When this approach proved insufficient, I added a `open_many` function that could iterate over a list of strings or a range of integers, but even that was not enough. Up to this point, working with Muck felt like an exercise in puzzle solving. When I discovered that the "Did You Know" pages switched naming conventions from "Year_Month" to numerical indices partway through the archive, it became clear that the static approach could never fully accommodate the messy reality of scraping jobs in the real world. Dynamic dependency communication is the solution. Early versions were implemented in the Muck Python library; more recently the system was made language-agnostic by monitoring system calls.

Census Data

The third experiment revolved around the U.S. Census Bureau's "American Community Survey" (ACS).[54] The code downloads the dataset, selects requested columns, joins the rows with their geographic locations, and then renders the data to an interactive map in an HTML page. For the demonstration it simply selects the total population column, but it could easily show various other columns of interest. This project tested Muck's ability to fetch large datasets, deal with compressed data automatically, and provide a useful workflow when the product is an interactive web page using client-side JavaScript.

Large datasets

Muck's fetching capabilities were originally built using the popular Python "Requests" library.[55] For very large downloads this approach presented problems because the fetched data is held in memory within the Python process. Since the goal of Muck's fetch API is to cache the results locally to the disk, it made more sense to use the Curl program, which runs as an external process and fetches straight to disk. From there, the downloaded file is opened just like any other.

A second problem quickly emerged with large files: A compressed archive might fit easily on disk, but takes up far too much space when expanded. To address this problem, I added additional loader functions to Muck that can handle decompression transparently, letting a script conveniently walk through archives without unpacking them to disk. These operations are computationally intensive, since decompression is performed on the fly as data is read, but for projects facing size limitations this feature makes it possible to work with large files locally.

Interactive pages

Interactive data journalism usually involves JavaScript code running in the browser. The development cycle for an interactive page typically revolves around making changes to the code and then reloading the page. In order for the programmer to realize the benefits of Muck's dependency-driven paradigm in this context, Muck needs to participate in this cycle. As a

proof of concept, I added a mode that runs a local web server. Whenever the browser makes a request, either because a page itself is visited or because a page needs a resource (CSS, image file, etc.), then Muck updates that product prior to serving it. Web developers typically use a different style of automation system,[xiii] so this feature might need some configuration options to appeal to professionals, but as an experiment it shows that Muck is an effective tool for interactive projects.

xiii. Two popular systems are Grunt (Grunt: The JavaScript Task Runner, https://gruntjs.com) and Gulp (gulp.js, https://gulpjs.com). Both operate by watching a project's source files and running tasks whenever changes are detected. Conceptually, the direction of the arrows in the network diagrams are reversed. While Muck works backwards from the desired product to the original sources in order to do as little work as possible, task-runner systems monitor inputs and trigger fixed pipelines of operations.

Results, Limitations, and Future Work

Preliminary workshops have suggested that Muck is not a great choice for complete beginners in data journalism. Learning to program is challenging, and Muck introduces a second layer of tools to worry about. This is not surprising: Beginners are better served by small, clean examples in a single modality. Muck requires a fair amount of context switching between files, data formats, and even languages. But for individuals with a bit of background knowledge, Muck appears to be quite manageable, especially if they have an existing project to work from.

Muck's primary strength is that it enforces a certain correctness in how steps interact. It also improves the clarity of project structure, but this only becomes valuable once projects reach a certain complexity. Still, some of the smaller conveniences like the `load` functions and Markdown support can make Muck a great choice for small projects.

Another advantage to using a build system is that as programmers make changes to various pieces of the code, they can choose to rebuild any downstream product and observe or test results as they see fit. This ability to hop around and observe the various effects of changes is powerful. It allows programmers to maintain focus on the desired final product, but also inspect intermediate results to gain a better understanding of what the code is doing. In the context of data journalism, this has a special appeal. We can work on the production of a fundamentally non-technical document, consisting of prose, graphical layout, and styling, together with our computational results. Changes to any of these aspects propagate through to the final product immediately.

Not having to write a makefile, build script or other configuration files is an immediate convenience, but the greater benefit in my experience is that the absence of the makefile lowers the cognitive barrier to refactoring the project. This is an entirely subjective observation, but I have repeatedly noticed that moving files and code around in Muck projects seems to incur fewer mistakes than in projects built with makefiles, simply because there is one less file to update. The simplicity of the naming convention makes it easy to refine names to be more descriptive as a project evolves. (One detail that would make this even easier would be if the standard find-and-replace feature of code editors would operate on file names as well as their

contents; depending on the editor, this might be achievable with a simple plugin.)

Muck currently handles compression formats in a fairly transparent manner, but this awareness is limited to the input side only, because while the `load` function can choose an appropriate decompression handler based on file extension, the output of a script goes straight to standard output and is entirely controlled by the programmer. It might be possible to add automatic compression of the output stream based on the target file extension, but when introducing features like this care must be taken not to create new problems.[xiv]

Limits of File-Level Dependencies

One major limitation of Muck is that it operates at a per-file granularity. The division of code into one module per file is standard across most programming languages, but this convention is fundamentally an artificial distinction compared to more formal semantic notions of scope and namespaces. Muck's view of the dependency graph is file-oriented because that is the level at which the tool can be language agnostic. A finer granularity of names is available at the language-specific level, which could make update operations more efficient. However, leveraging this would require elaborate static analysis for each supported language, as well as an execution engine that could rerun individual subsections of code. This is the essence of reactive programming; projects such as Eve[56] and D3.Express[57] are good examples of new projects pursuing this approach.

Dynamic Dependency Inference

As described earlier, Muck uses a hybrid strategy to infer dependencies, which cannot guarantee a perfectly accurate graph in all cases. Currently, certain tools bypass Muck's attempts to monitor their file access, which presents a major problem. How to best deal with this challenge is an ongoing inquiry.

xiv. In this case, it is not clear how the script would indicate that it is doing compression itself. This could lead to accidentally compressing the output data twice.

In all likelihood this problem will never be completely solved, and Muck should do a better job of protecting against error when these situations occur. Currently, if a dependency is not detected, then it will not exist when building from a clean state, and Muck will immediately fail with an error. However, build systems by nature can encounter a huge number of possible project states, so there may be tricky cases where further protection is warranted.

The other downside of Muck's just-in-time approach is that it can make the runtime state of the build rather complicated. The Muck process will have many subprocesses suspended at the same time as it builds a long chain of dependencies. Only one subprocess is ever active at one time (no parallelism), but there is a minor risk of concurrent writes to the same file by two different misbehaved child processes.

Compatibility with Databases

Another major area that Muck has yet to address is databases. Currently, Muck conceives of data as being written by one step and then consumed by one or more subsequent steps. The more the programmer breaks their data down into logical pieces, the better Muck is able to perform incremental updates. In contrast, databases are monolithic stores that perform both read and write operations. Muck does support SQLite natively, but only when all of the write operations happen in an initial import step.

In practice, databases are often remote and therefore outside of Muck's control. This is fine so long as the programmer is willing to fetch the data and then work with it locally. But even a local database like SQLite or a simple key-value store is incompatible with Muck's model if it is constantly mutating, because Muck cannot detect database changes as it does with text files (by inspecting the file timestamp and content hash). Database storage is both mutable and opaque, so for Muck to do any sort of dependency analysis on the database, strict update semantics would need to be defined. In the meantime, programmers can interact with remote databases as they do with any other web API, either utilizing Muck's fetch caching or not. For local databases, Muck users must be careful to manage state themselves in a way that is compatible with Muck's update model.

Muck on the Server

In contrast to the limitations regarding remote databases, there are no obvious technical reasons why Muck could not run as a server that updates pages dynamically. Indeed, this capability is already implemented as a convenience feature for local development. But using Muck as a public server would entail all the usual security considerations for the modern web, and while the particulars have not been explored in any great detail, there are obvious and serious concerns. For one, the idea of running a web server that executes a long chain of subprocesses is quite antithetical to basic security practices.

More realistic would be to use a continuous integration server like Jenkins[58] to trigger Muck rebuilds whenever a project's git repository is updated, and then serve the built products appropriately using a separate web server. In this arrangement, Muck plays the role of the build system in a traditional continuous integration setup; the only difference is that instead of serving out a compiled application, the final product is a document.

Future Steps

Muck is currently useful, but much work remains to be done. First and foremost, tutorials and technical documentation are required for it to be a compelling choice. Secondly, better testing and support for various formats and languages would greatly improve the utility of the system. Above all, the tool needs users in order to justify continued development. Like any new developer tool, this is is a bit of a chicken-or-egg problem. At this point I am seeking data journalists who would be willing to try out the system in exchange for my support fixing bugs and adding features to make Muck truly valuable.

Conclusion

In summary, Muck provides a framework for data programmers that improves correctness, organization, and efficiency during development. The tool aims to be simple and practical, and requires minimal setup so that potential users can easily try it out. It stands apart from most tools in the data journalism space in that the design is based on the merits of build tools, which have proven indispensable in the systems software industry over the past forty years. Muck is simple enough that it should be useful to programmers with only basic command line skills. At the same time, by modeling work as a dependency graph it enables programmers to construct arbitrarily complex jobs in an intuitive, maintainable style. Lastly, by imposing minimal requirements on how individual steps are programmed, projects that outgrow Muck's build model can transition into full-fledged, modular applications without drastic changes.

At present, Muck is an ongoing experiment and I would be grateful for feedback from practitioners of all kinds. Professional data journalists, students, teachers, data scientists, and programmers of all persuasions might find it useful or at least thought-provoking to browse through the demos, reproduce the demo pages from their original sources, and perhaps even make improvements to the code, which is open source. Please get in touch through https://github.com/gwk/muck with any questions or feedback.

Citations

1. George King et. al., “Reducing Barriers between Programmers and Non-Programmers in the Newsroom,” Tow Center for Digital Journalism, December 28, 2015, http://towcenter.org/reducing-barriers-between-programmers-and-non-programmers-in-the-newsroom/.

2. Sisi Wei, Olga Pierce, and Marshall Allen, “Surgeon Scorecard,” ProPublica, July 15, 2015, https://projects.propublica.org/surgeons/.

3. Staff, “The Counted: People Killed by Police in the U.S.,” *The Guardian*, 2016, https://www.theguardian.com/us-news/ng-interactive/2015/jun/01/about-the-counted.

4. Sarah Cohen, “The Newest Muckrakers: Investigative Reporting in the Age of Data Science,” Computation+Journalism Symposium, 2016, https://www.youtube.com/watch?v=Z5fAhuaCXlA.

5. Mark Hansen et al., “Artificial Intelligence: Practice and Implications for Journalism,” Tow Center for Digital Journalism, 2017, https://academiccommons.columbia.edu/catalog/ac:gf1vhhmgs8.

6. Dan Kopf, “If You Want to Upgrade Your Data Analysis Skills, Which Programming Language Should You Learn?” Quartz, September 22, 2017, https://qz.com/1063071/the-great-r-versus-python-for-data-science-debate/.

7. Overview. A Short Introduction to the Processing Software and Project from the Community, https://processing.org/overview.

8. Jupyter, http://jupyter.org/.

9. Martin Odersky, “The Trouble with Types,” InfoQ, October 28, 2013, https://www.infoq.com/presentations/data-types-issues.

10. Julia, https://julialang.org/.

11. Donald E. Knuth, “Literate Programming,” *The Computer Journal*, no. 2 (January 1, 1983): 97–111, http://www.literateprogramming.com/knuthweb.pdf.

12. Dummy’s Guide to Async, https://janestreet.github.io/guide-async.html.

13. Mona Chalabi, “America’s Broken Healthcare System—in One Simple Chart,” July 2, 2017, https://www.theguardian.com/us-news/datablog/2017/jul/02/us-healthcare-broken-system-one-chart.

14. Python Software Foundation, “Custom String Formatting,” Python.org Library, https://docs.python.org/3/library/string.html#string-formatting.

15. Diff Utility, https://en.wikipedia.org/wiki/Diff_utility.

16. George King, “Pat: A Patch Utility for Data Applications,” GitHub, 2016, https://github.com/gwk/pat.

17. Muck: A Build Tool for Data Projects, https://github.com/gwk/muck.

18. Overview of Make, https://www.gnu.org/software/make/manual/make.html#Overview.

19. Mike Bostock, “Why Use Make,” bost.ocks.org, February 23, 2013, https://bost.ocks.org/mike/make/.
20. Peter Miller, “Recursive Make Considered Harmful,” *AUUGN Journal of AUUG Inc.* No. 1 (1998): 14–25, http://aegis.sourceforge.net/auug97.pdf.
21. D. J. Bernstein, “Rebuilding Target Files When Source Files Have Changed,” http://cr.yp.to/redo.html.
22. CMake Tutorial, https://cmake.org/cmake-tutorial/.
23. Mike Shal, “Tup,” gitup.org, http://gittup.org/tup/.
24. Mike Shal, “Build System Rules and Algorithms,” gitup.org, 2009, http://gittup.org/tup/build_system_rules_and_algorithms.pdf.
25. Avery Pennarun, “redo,” GitHub, 2010, https://github.com/apenwarr/redo.
26. Alan Grosskurth, “Purely Top-Down Software Rebuilding,” University of Waterloo, 2007, http://grosskurth.ca/papers/mmath-thesis.pdf.
27. Ninja, https://ninja-build.org/.
28. RAKE–Ruby Make, https://github.com/ruby/rake.
29. Jake: JavaScript build tool task runner for NodeJS, http://jakejs.com/.
30. Shake Build System, http://shakebuild.com.
31. Apache Ant - Welcome, http://ant.apache.org/.
32. What Is Maven, https://maven.apache.org/what-is-maven.html.
33. sbt Reference Manual, http://www.scala-sbt.org/1.x/docs/index.html.
34. Leinengen, https://leiningen.org/.
35. Boot: Build Tooling for Clojure, http://boot-clj.com/.
36. OCamlbuild, https://github.com/ocaml/ocamlbuild.
37. Factual: Drake, https://github.com/Factual/drake.
38. GNU Make: An Introduction, https://www.gnu.org/software/make/manual/make.html#Introduction.
39. clang: a C Language Family Frontend for LLVM, https://clang.llvm.org/.
40. Makedepend, https://en.wikipedia.org/wiki/Makedepend.
41. Eric Steven Raymond, “Chapter 15. Tools: make: Automating Your Recipes,” in *The Art of Unix Programming* (Boston: Pearson Education, 2003), http://www.catb.org/esr/writings/taoup/html/ch15s04.html.
42. Bill McCloskey, “memoize.py,” GitHub, 2012, https://github.com/kgaughan/memoize.py.
43. Bill McCloskey, “memoize.py classic,” Berkeley, 2007, https://web.archive.org/web/20070612145920/http://www.eecs.berkeley.edu/~billm/memoize.html.
44. Ben Hoyt, “Fabricate,” GitHub, 2009, https://github.com/SimonAlfie/fabricate.
45. Strace (1)—Linux Man Page, https://linux.die.net/man/1/strace.
46. John Fultz, “Launching Wolfram Player for iOS,” Wolfram, November 16, 2016, http://blog.wolfram.com/2016/11/16/launching-wolfram-player-for-ios/.
47. Chris Said, “How to Make Polished Jupyter Presentations with Optional

Code Visibility," chris-said.io, February 13, 2016, http://chris-said.io/2016/02/13/how-to-make-polished-jupyter-presentations-with-optional-code-visibility/.

48. Max Goldman and Rob Miller, "Reading 9: Mutability & Immutability," MIT, Fall 2015, http://web.mit.edu/6.005/www/fa15/classes/09-immutability/.

49. Declarative Programming, https://en.wikipedia.org/wiki/Declarative_programming.

50. George King, Gabe Stein, and Mark Hansen, "Data Cleaning with Muck," Tow Center for Digital Journalism, July 12, 2016, http://towcenter.org/data-cleaning-with-muck/.

51. Micra, http://www.micra.com.

52. Wikipedia: Did You Know, https://en.wikipedia.org/wiki/Wikipedia:Did_you_know.

53. TextBlob: Simplified Text Processing, 2013, https://textblob.readthedocs.io.

54. U.S. Census Bureau, "American Census Survey," Census.gov, 2016, https://www.census.gov/programs-surveys/acs/.

55. Kenneth Reitz, "Requests: HTTP for Humans," Python.org library, 2017, http://docs.python-requests.org/en/master/.

56. Eve: A Moonshot to Make Programming Accessible to Everyone, http://witheve.com.

57. Mike Bostock, "A Better Way to Code," Medium, Apri 28, 2017, https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0.

58. Jenkins, https://jenkins.io/.