The Gamma: Programing tools for transparent data journalism

Tomas Petricek
Computer Laboratory, University of Cambridge
Cambridge, United Kingdom
tomas@tomasp.net

ABSTRACT

Data journalism encourages reader interaction. This is often done through simple user interfaces. For more advanced readers, there is typically a download with the raw data behind the visualization. However, there is an interesting gap between the two. What if the reader wants to change a parameter of a visualization that is not exposed through the user interface? What if they want to re-create the same visualization, but using data from a different source?

We believe that the fundamental reason for this inflexibility is the fact that accessing data and building interactive visualizations is a difficult programming problem. As a result, data journalists use a wide range of tools, often involving manual steps, which makes it hard to publish the entire process as a reproducible program.

In this paper, we present The Gamma, a project that reduces the number of steps needed to link a data source to an end-user visualization. The Gamma uses programming language techniques to make data sources easier to access and to automatically build user interfaces that let readers modify parameters of visualizations. But behind the visualization and the user interface, there is full source code, which makes reports transparent, more reproducible and more accountable.

Categories and Subject Descriptors

D.2.6 [*Programming Environments*]: Interactive environments

General Terns

Languages, Documentation, Experimentation

1. INTRODUCTION

Let us start with an example. Consider a simple data-driven report that uses data from the World Bank to compare CO2 emissions of different countries. The typical way to build such report is to download data for the indicator by hand or using a simple script that calls the World Bank API, save it to a file, share it as a Google Sheet and use a charting library to produce a map with an overlay. In a more interactive example, we could download data for multiple years and let the reader choose two years for comparison.

This approach has a number of issues. First of all, we can easily make a mistake in any of the manual steps and get data for a wrong indicator or a wrong year. The reader has no way to verify this. Second, the reader has only very limited options for interacting with the report. They can compare the pre-defined years (using our user interface) or they can download the data and build different visualizations (using our data). However, what if the reader wanted to get data for a different year or compare different indicator also exposed by the World Bank (e.g. CO2 per capita rather than total CO2)?

For simple changes, the reader should be able to use the user interface. For more complex changes, the reader might need be able to modify the source code, but they should still be able to reproduce the report and obtain a similar interactive visualization as the result. By examining the source code, the reader should be also able to verify what data sources are used by the visualization.

The goal of The Gamma project is to make the above scenarios possible. This is done by taking the code behind data journalism as the fundamental aspect and building programming tools on top of it. The key ideas presented in this paper are:

- A report is just text with source code that can be executed to produce visualizations (Section 2.1). The reader initially sees the final product, but they can change options through user interface or even view the underlying source code.
- The data sources are mapped into the programming environment using *type providers* (Section 2.2). This makes it possible to access data with complex structure directly from the code.
- When editing the code, a rich editor provides auto-completion and other help (Section 2.3), hiding much of the usual complexity behind data access and making basic changes easy.
- The Gamma automatically generates user interface that lets the reader change parameters of the visualization based on the source code that was used to build it (Section 2.4).

The Gamma prototype is created using F# and is available as open source: https://github.com/tpetricek/TheGamma. A sample report on carbon emissions can be found at http://thegamma.net/carbon.

2. THE GAMMA PROJECT

Data journalism not only adds a new perspective to reporting, but it also changes the media formats. The Gamma project takes this development further and treats data-driven article not as text with additional content such as photographs or visualizations, but as a *literate program* [6]. That is, a mix of plain text and program code. The article that the reader sees then becomes just (one possible) view or rendering of the program execution.

This is perhaps just a philosophical shift, but it has a number of important implications. We can provide multiple different views of the same source – the raw text and the article with the final visualizations are just two of those. A program should be self-contained and have clear dependencies. That is, we know exactly what is needed to re-run the code and obtain the report. Finally, seeing a report as program also lets us use a number of interesting programming language research techniques that can make reporting easier, faster and less error-prone.

Computation+Journalism Symposium 2015, New York
This work is licensed under the Creative Commons Attribution License.

2.1 Data-driven article as a literate program

Let us return to the example of visualizing CO2 emissions using data from the World Bank. The following small (but complete) literate program produces two charts with brief commentary, as shown in Figure 1.

```
Carbon emissions today
----------------------

If we look at CO2 emissions for the whole world we can
see that China and US are the largest polluters
followed by India, Russia and rich western countries:

    let co2 = world.byYear
      .``2010``.``Climate Change``
      .``CO2 emissions (kt)``

    let colorScale = [ "#6CC627"; (...) ]
    chart.geo(co2).colorAxis(colors=colorScale).show()

If we look at the share of CO2 emissions for indivi-
dual countries, we can see that China, US and India
are responsible for a half of the world's CO2:

    let co2 = world.byYear
      .``2010``.``Climate Change``
      .``CO2 emissions (kt)``.sortValues(reverse=true)

    let sumRest = co2.skip(6).sum()
    let topAndRest = co2.take(6)
      .append("Other countries", sumRest)
    chart.pie(topAndRest).show()
```

The Gamma uses the F# language [10] for embedded code snippets and the Markdown format for text. F# is statically typed, so for example, the type of `co2` is `series<string, float>`, which represents a collection of numerical values indexed by strings (country names). The only F# keyword in the example is `let`, which defines a variable. Also note that types are not written explicitly in the code and are inferred automatically.

The first interesting aspect of the example is the data access. The `world` identifier is a global value that provides access to World Bank data. In F#, the "." operator is used (as usual) to access a member of an object and `` is used to escape identifiers that contain characters such as spaces and numbers.

When writing the code, authors or readers do not need to know full name of indicators and countries, because the editor provides auto-completion on the names (Section 2.3). The `world` object is also not a predefined construct that would have to be created manually. Instead, it is provided (Section 2.2) based on the schema information exposed by the World Bank.

Although the example used here is quite simple, it shows that the code can do more than just link an indicator to a chart. The first example also specifies a color scale (part of the list is omitted here). The second example partitions the data taking the 6 largest polluters and adding sum of all other countries as another item.

As discussed in Section 2.4, the produced visualization is not a static chart. The Gamma understands the source code and generates user interface for some of the parameters (in this example, letting the user change the year and the indicator). However, the reader can see the full source code, make changes that are not exposed through the user interface (say, change the color scheme or the number of countries shown in the pie chart), refresh the view and immediately see the new visualization.

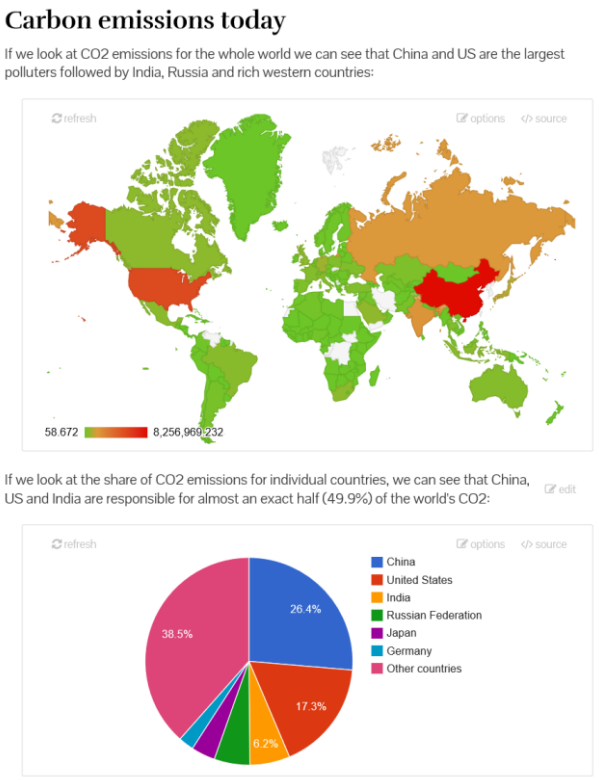

Figure 1. Two visualizations produced by running a sample program

2.2 Data access with type providers

The Gamma makes data available through type providers [9, 2]. The World Bank type provider (accessible via `world`) is a good example of the mechanism, but it is just one specific case. Type providers can similarly be used to access RESTful services, Excel spreadsheets, CSV files and other sources. Section 3.1 discusses a number of other possible uses within The Gamma.

F# type providers. A type provider is a compiler extension that generates types (with members) based on an external data source. To do so, a type provider can run arbitrary code (in case of World Bank, it calls the API to get information about the schema of the data source). The generated types define the structure (i.e. the chain of members that can be used to access the data) and the runtime code to be executed to get the data. In The Gamma, the runtime code is then translated to JavaScript and executed in the web browser when a report is opened or when the reader makes a change to the source code.

The most interesting aspect of a type provider is how it maps the external data source to members visible to the F# code. This is easy for a CSV file, but data sources like the World Bank have rich structure that can be mapped in a number of ways.

World Bank type provider. The World Development Indicators exposed by World Bank store data indexed along three axes – indicators, countries and years. In the above example, we choose a *year* together with an *indicator* and obtain data for all *countries*. This is one possible projection. For other visualizations, we may want to obtain data for all *years* (given a specific *country* and *indicator*) to show the change over time.

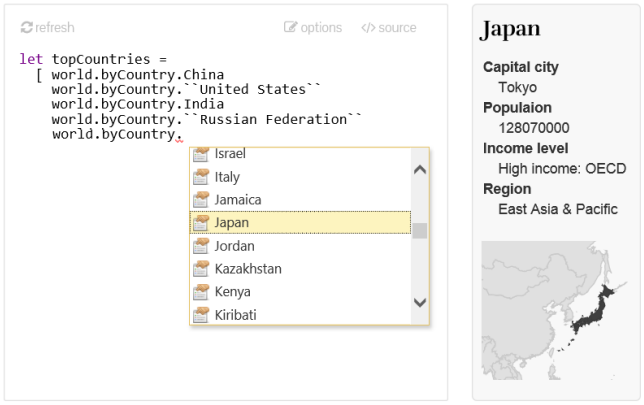

Figure 2. Choosing countries with auto-complete and information side-panel

The `world` type provider available in The Gamma prototype supports the following two ways of accessing data:

```
world.byYear.[Year]
  .[Indicator group].[Indicator name]

world.byCountry.[Country name]
  .[Indicator group].[Indicator name]
```

In both cases, we specify two of the three possible keys and obtain a series that returns values indexed by the third key (by country in the first case and by year in the second case).

The `[Year]` and `[Country name]` placeholders stand for one of the 65 years (1950 … 2015) available in the data source and one of the 230 countries. The World Bank exposes over 3000 indicators and to make navigation easier, the type provider groups them by indicator category (although there is also ``All indicators`` member that provides access to all of them in a flat structure). As mentioned before, navigating through the data source is possible thanks to the rich editor support, which we discuss next.

2.3 Editor support for data navigation

In the reading view of a report (Figure 1), there is a small "source" button attached to every visualization. This is where the more advanced readers can access the source code shown in the previous section and edit it or create their own visualizations.

The editor makes writing and changing code easier by providing tools known from IDEs for (mostly) statically-typed object-oriented languages, but adapted to the data-focused environment. The Figure 2 shows two of the tools using a code sample that creates a list of countries (e.g. to appear in a line chart that compares the top 5 polluters over the last 50 years).

The completion list is using the F# Compiler Service[1] to parse the source code, infer the types and offer members that are available on the type of the expression being typed. In the example, the type of `world.byCountry` is provided by a type provider and has a member for each country tracked by the World Bank.

The panel on the right shows information about the currently selected completion. The information is, again, obtained from the type. Each member can have a documentation and The Gamma supports showing rich HTML in the documentation. The World Bank type provider shows documentation for countries (containing basic statistics and a map) and for indicators (containing full description of the indicator). Note that this is not a built-in functionality of The Gamma project, but rather a feature of the World Bank type provider. Other data sources that can be integrated into The Gamma can provide different information (or even interactive content) using the exact same mechanism.

Finally, it is worth adding that the two uses of the static types to provide better tooling described here are just scratching the surface of what is possible. Section 3.2 discusses future directions.

2.4 Generating interactive user interfaces

The editor tooling makes modifications to the source code easier, but we do not expect that every reader who wants to interact with data-driven articles should be able to modify the source code.

The Gamma also provides a simple user interface that lets the readers change parameters of the visualization. This does not require additional input from the author – the user interface is generated automatically using the information from type providers and from the source code.

To see how the mechanism work, consider the following code snippet that obtains CO2 emissions for 3 countries specified explicitly in a list and creates a line chart comparing them (the highlighted parts are discussed below):

```
let topCountries =
  [ world.byCountry.China
    world.byCountry.India
    world.byCountry.``United States`` ]

let growths =
  topCountries.map(fun p ->
    p.``Climate Change``.``CO2 emissions (kt)``
      .set(seriesName=p.name) )

chart.line(growths).show()
```

Figure 3 shows the result of the visualization together with the user interface that appears when the reader clicks the "options" button. It includes two elements that correspond to specific patterns in the source code (highlighted above). A drop-down lets the reader change the indicator (highlighted in green) in order to view, for example, CO2 emissions per capita or Energy consumption. A multi-select lets the reader add and remove countries in the list (highlighted in orange). The user interface is linked to the source code (even when it is hidden), meaning that changes in the user interface change the source code (and vice versa), which is then re-run to obtain a modified visualization.

Before discussing the two specific patterns, it is worth noting that the underlying mechanism relies on the same underlying information as the code editor. This can be seen in Figure 3, where the user is choosing countries and sees the same information panel with country details as in Figure 2.

Choosing one of several properties. In case of the drop-down, we look for a simple code pattern that looks as follows:

```
<ident1>.(...).<identn-1>.<identn>
```

[1] A standard component used in other F# tools available at: http://fsharp.github.io/FSharp.Compiler.Service

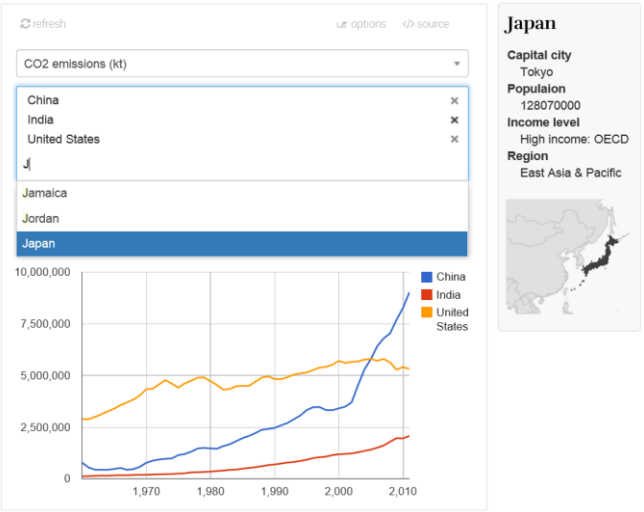

Figure 3. User interface for configuring a sample visualization.

Next, we obtain the inferred type of `<ident1>.(...).<identn-1>`. This is our *parent type*. The parent type has `<identn>` as one of its members. We also obtain the type of this member and we find all other members of the parent type that have the same type as the one member. Those are then made available in the drop-down. Since all the offered identifiers are members of the parent type and have the same type, they can be safely substituted in the source code.

In the example shown here, the parent type is the object returned by `p.``Climate Change``, which represents a category of indicators and has specific indicators as its members. The type of ``CO2 emissions (kt)`` is then `series<int, float>` representing a time-series. The user interface thus exposes all other numerical indicators belonging to the climate change category.

Choosing a list of items. The second user interface component works in a very similar way. It looks for a more complex pattern:

```
[ <ident1>.(...).<identn>.<option1>
  <ident1>.(...).<identn>.<option2>
  (...)
  <ident1>.(...).<identn>.<optionm> ]
```

Note that the elements of the list would also match the pattern discussed above. For this reason, editors for lists are always preferred. As above, we check that all the options are members of the same parent type and generate an editor that lets the reader choose one or more elements from the available members (again, we also offer only members of a single type).

The two editors discussed here are just two examples of a more general mechanism. It is easy to imagine other editors that could be built in the same way. For example, a specialized UI element could be built for a list of colors or for numerical constants.

The two examples discussed here should be a sufficient demonstration of the general mechanism. The key idea is that if we treat the report as a program, we can use (mostly standard) programming language techniques to build powerful user interfaces that are not tied to a specific visualization, but instead work with any report created using The Gamma.

3. FUTURE AND RELATED WORK

The Gamma project combines ideas from programming language research with the perspective of data journalism (cf. [4]). Important prior work has been on both sides, but to our best knowledge, the idea of treating report as a program (and building programming tools on top of that) is new in The Gamma.

The project is currently an early prototype and there is much more that could be done on both the data-access side (creating new type providers) and the tooling side (using the source code in other interesting ways).

3.1 Type providers and data sources

The examples discussed in this paper use World Bank as the data source. Creating similar single-purpose type providers for other data sources with a REST API requires advanced F# programming skills, but it brings no fundamental difficulty and we intend to add additional data sources in the future.

Another approach for creating type providers is to build a provider for a *format* rather than a specific *source*. The Gamma supports this style for the JSON format. The following can be used to get country codes from the World Bank using the API directly:

```
type countryResponse =
  json<"http://api.worldbank.org/country?format=json">

let data = countryResponse.wrap(json)
let codes = data.array
  .filter(fun a -> a.region.id <> "NA")
  .map(fun a -> a.iso2Code)
```

The first line invokes the JSON type provider with a sample URL. This produces a type with members based on the sample document which is then used to read a `json` value and access its members such as `id` and `iso2code`. The important fact is that the members are known to the editor – and so it can provide similar help as when choosing countries or indicators.

The same approach can be used for other formats. The F# Data library [7, 8] supports CSV, JSON, XML and embedded HTML tables. In the context of data journalism, this approach could be used to provide access to Excel, Google Sheets and other frequently used data formats. This approach can also be useful for data that are either obtained offline, or require significant amount of manual preprocessing that cannot be easily captured in reproducible code.

Finally, building a type provider for large-scale data sources such as http://data.gov and http://data.gov.uk is an interesting open problem. There has been some work done on adding more structure to the data [3], but more research is needed before it can be accessed with the same ease as the World Bank data sets used here.

3.2 Getting more from the program

In The Gamma prototype, we use code analysis of the source program to automatically generate user interfaces for visualizations (and to provide usual editor tooling). However, there is a number of other techniques from programming language research that could be adapted and used for data journalism. To list a few:

- **Automatic data citations.** Analyzing the source code and type providers used makes it possible to automatically find out what data sources were used in the article. Programming languages can also automatically track *data provenance* [1] to asses what aspects of the report depend on what data sources (for example, what remains valid if one unverified data source is incorrect?)

- **Numerical values in context.** When a report shows a numerical value (CO2 emissions of Czech Republic are 111,168 kt), it is hard to understand what the number means without context. As The Gamma knows the source of the number, it can automatically generate comparison and, for example, display an information panel with other countries in the same region. Tracking such information is closely related to the tracking of physical units that is available in F# [5].

3.3 Background and related work

The Gamma prototype uses many of the common tools and libraries from the F# community including F# Compiler Service[2] for code analysis; the source code editor is based on the F# Web IntelliSense[3] project and the browser execution uses FunScript[4] to produce JavaScript visualizations. The idea of rich information panel (e.g. showing country details) is inspired by Tsunami[5].

The work on type providers for JSON and the World Bank is inspired by earlier work of the author on F# Data library. Finally, the idea of mixing source code with text goes back to literate programming of Donald Knuth [6]. The model presented here is simple, but other work on literate programming suggests a number of possible extensions.

5. CONCLUSIONS

In this paper, we present The Gamma – a tool that aims to make data-driven articles more transparent, reproducible and accountable. The key design principle behind The Gamma is that the media format of data-driven articles is no longer text with embedded photographs and visualizations, but instead a *program*.

This changes a number of basic assumptions about data-driven reports. First, programs should be self-contained and recreating the report should thus require no manual steps. Second, the program can be viewed and interacted with in multiple ways. Third, we can use programming language techniques to build a number of tools on top of such data-driven articles.

When interacting with The Gamma, the reader initially sees the final text with rendered visualizations. Next, they can change parameters of the visualizations (through an automatically generated user interface). Finally, the reader also has access to the source code behind the report. This makes it possible to change remaining parameters, but also verify what data sources are accessed and how.

REFERENCES

[1] J. Cheney, et al. *Provenance: a future history.* Proceedings of OOPSLA. ACM, 2009.

[2] D. R. Christiansen. *Dependent type providers.* In Proceedings of Workshop on Generic Programming, 2013.

[3] Li Ding, et al. *TWC data-gov corpus: incrementally generating linked government data from data.gov*. In Proceedings of the 19th conference on World Wide Web. ACM, 2010.

[4] J. Gray, L. Chambers, L. Bounegru. *The data journalism handbook*. ISBN 978-1449330064. O'Reilly, 2012.

[5] A. Kennedy. *Types for units-of-measure: Theory and practice*. Central European Functional Programming School. Springer Berlin Heidelberg, 2010. 268-305.

[6] D. Knuth. *Literate programming*. The Computer Journal 27.2, 97-111, 1984.

[7] T. Petricek, G. Guerra, and contributors. *F# Data: Library for data access*, 2015. http://fsharp.github.io/FSharp.Data

[8] T. Petricek, G. Guerra, D. Syme. *F# Data: Making structured data first-class citizens.* Draft available at: http://tomasp.net/academic/drafts/fsharp-data

[9] D. Syme et al. *Themes in information-rich functional programming for internet-scale data sources*. In Proceedings of the DDFP Workshop, 2013

[10] The F# Software Foundation (FSSF). *The F# language*. Available online at http://www.fsharp.org, 2015

[2] http://fsharp.github.io/FSharp.Compiler.Service
[3] https://github.com/BayardRock/FSharpWebIntellisense
[4] http://funscript.info
[5] http://tsunami.io