Data exploration through dot-driven development*

Tomas Petricek[1]

1 The Alan Turing Institute, London, UK
tomas@tomasp.net

Abstract

Data literacy is becoming increasingly important in the modern world. While spreadsheets make simple data analytics accessible to a large number of people, creating transparent scripts that can be checked, modified, reproduced and formally analyzed requires expert programming skills. In this paper, we describe the design of a data exploration language that makes the task more accessible by embedding advanced programming concepts in a simple core language.

The core language uses type providers, but we employ them in a novel way – rather than providing types with members for accessing data, we provide types with members that allow the user to access data, but also compose complex queries using only member access ("dot"). This lets us recreate functionality that usually requires complex type systems (row polymporphism, dependent typing) in an extremely simple object-based language.

We formalize our approach using a minimal object-based calculus and prove that programs constructed using the provided types represent valid data transformations. We then discuss a prototype implementation of the language, together with a simple editor that bridges some of the gaps between programming and spreadsheets. We believe that this provides a pathway towards democratizing data science – our use of type providers significantly reduce the complexity of languages that one needs to understand in order to write scripts for exploring data.

1998 ACM Subject Classification D.3.2 Very high-level languages

Keywords and phrases data science, type providers, pivot, aggregation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction – Simplifying data exploration

The rise of big data and open data initiatives means that there is an increasing amount of raw data available. At the same time, the fact that "post-truth" was chosen as the word of 2016 suggests that there has never been greater need for increasing data literacy and building tools that let anyone – including journalists and interested citizens – explore such data and use it to support facts.

Spreadsheet tools made data exploration accessible to a large number of people, but operations performed in a spreadsheet application cannot be reproduced or replicated with different input parameters – the manual mode of interaction is not repeatable and it breaks the link with the original data source. The answer to this problem is to explore data programmatically. A program can be run repeatedly and its parameters can be modified.

However, even with the programming tools generally accepted as simple, exploring data is surprisingly difficult. For example, consider the following Python program (using the pandas library), which reads a list of all Olympic medals ever awarded (see Appendix A) and finds top 8 athletes by the number of gold medals they won in Rio 2016:

* This work was supported by Google Digital News Initiative.

```
olympics = pd.read_csv("olympics.csv")

olympics[olympics["Games"] == "Rio (2016)"]
    .groupby("Athlete")
    .agg({"Gold" : sum})
    .sort_values(by = "Gold", ascending = False)
    .head(8)
```

The code is short and easy to understand, but writing or modifying it requires the user to understand intricate details of Python and be well aware of the structure of the data source.

The short example specifies operation parameters in three different ways – indexing [...] is used for filtering; aggregation takes a dictionary {...} and sorting uses optional parameters. The dynamic nature of Python makes the code simple, but it also means that auto-completion on member names (after typing dot) is not commonplace and so fining the operation names (groupby, sort_values, head, ...) often requires using a search engine. Furthermore, column names are specified as strings and so the user often needs to refer back to the structure of the data source and be careful to avoid typos.

The language presented in this paper reduces the number of language features by making member access the primary mechanism. For example, finding top 8 athletes by the number of gold medals in Rio 2016 can be written as (in code «...» is written as '...'):

```
olympics
  .«filter data».«Games is».«Rio (2016)».then
  .«group data».«by Athlete».«sum Gold».then
  .«sort data».«by Gold descending».then
  .«paging».take(8)
```

The language is object-based with nominal typing. This means that auto-completion can provide list of available members when writing and modifying code. The members (such as «by Gold descending») are provided by a type provier based on the knowledge of the data source and transformations applied so far – thus only valid and meaningful operations are provided. The rest of the paper gives a detailed description of the mechanism.

Contributions. This paper explores an interesting corner in the programming language design space. We support it by a practical implementation and a formalization with a correctness proof. Our specific contributions are:

- We use type providers in a new way (Section 2). Previous work focused on providing members for direct data access. In contrast, our pivot type provider (Section 5) lazily provides types with members that can be used for composing queries, making it possible to perform entire date exploration through single programming mechanism (Section 3.1).
- Our mechanism shows how to embed fancy types into a simple nominally-typed programming language (Section 3.3). Our proof-of-concept language tracks names and types of available columns in the data set (inspired by row types), but the same mechanism can be used to implement other features in any Java-like language.
- We implement the language outlined in this paper (`github.com/the-gamma`) and make it available as a JavaScript component that can be used to build open and transparent data-driven visualizations (`thegamma.net`) and discuss the implementation (Section 7).
- We formalize the language (Section 4) and the type provider for data exploration (Section 5) and show that queries constructed using the type provider are valid (Section 6). Our formalization also covers the laziness of type providers, which is an important aspect not covered in the existing literature.

2 Background – Type providers

The work presented in this paper consists of a simple nominally-typed host language and the pivot type provider, which provides types with members that can be used to construct and execute queries against an external data source. This section briefly reviews the existing work on type providers and explains what is new about the pivot type provider.

Information-rich programming. Type providers were first presented as a mechanism for providing type-safe access to rich information sources. According to Syme et al. [X], a type provider is a compile-time component that imports external information source into a programming language. It provides two things to the compiler (or editor) hosting it. First, it provides a type signature that models the external source using structures understood by the host language (e.g. types with members). Second, it provides an implementation for the signatures which accesses data from the external source.

For example, the World Bank type provider [X] provides a fine-grained access to development indicators about countries of the world. The following accesses CO2 emissions in 2010 for all available countries:

```
world.byYear.«2010».«Climate Change».«CO2 emissions (kt)»
```

The provided schema consists of types with members such as «CO2 emissions (kt)» and «2010». The members are generated by the type provider based on the meta-data provided by the World Bank. The second part provided by the type provider is code that is executed when the above code is run. For the example above:

```
series.create("CO2 emissions (kt)", "Year", "Value",
    world.getByYear(2010, "EN.ATM.CO2E.KT"))
```

Here, the provided code relies on runtime library that provides a representation of a series (mapping from keys to values) and the getByYear function that downloads data for a specified indicator represented by a code. The type provider provides a type-safe access to known indicators, which exist only as strings in the compiled code, increasing safety and making data access easier thanks to auto-completion (which offers a list of available indicators).

Types from data. Recent work on the F# Data library [X] uses type providers for accessing data in structured formats such as XML, CSV and JSON. This is done by inferring the structure of the data from a sample document, provided as a static parameter to a type provider. In the following example (adapted from [X]), a sample URL is passed to JsonProvider:

```
type Weather = JsonProvider⟨"http://api.owm.org/?q=London"⟩

let ldn = Weather.GetSample()
printfn "The temperature in London is %f" ldn.Main.Temp
```

As in the World Bank example, the JSON type provider generates types with members that let us access data in the external data source – here, we access the temperature using ldn.Main.Temp. The provided code attempts to access the corresponding nested field and convert it to a number. The relative safety property guarantees that this will not fail if the sample is representative of the actual data loaded at runtime.

Pivot type provider. The pivot type provider presented in this paper follows the same general mechanism as the F# type providers discussed above, although it is embedded in a simple language that runs in a web browser.

The main difference between our work and the type providers discussed above is that we do not use type providers for mapping external data sources into the type system (by providing members that correspond to parts of the data). Instead, we use type providers for (lazily) generating types with members that let users compose type-safe queries over the data source. As discussed in Section 4, we model the provided code by a relational algebra.

This means that our use of type providers is more akin to meta-programming or code generation with one important difference – the schema provided by the pivot type provider is potentially infinite (as there are always more operations that can be applied). To support this, we use the fact that type providers are integrated into the type system and types can be provided lazilly. This is also a new aspect of our formalization in Section 4.

3 Analysis – What makes data exploration scripts complex

We started by contrasting a data exploration script using a popular Python library pandas with a script that can be written using the pivot type provider (Section 1). In this section, we analyze what makes the Python code complex and we discuss how our alternative design aims to simplify it. We use Python as a concrete example, but other commonly used tools (mentioned throughout) share the same properties:

1. The filtering operation is written using indexing [. . .] while all other operations are written using member invocation with (optionally named) parameters. In the first case, we write an expression olympics["Games"] == "Rio (2016)" returning a vector of Booleans while in the other, we specify a parameter value using by = "Gold". In other languges, a parameter can also be a lambda function specifying a predicate or a transformation.
2. The aggregation operation takes a dictionary {. . .}, which is yet another concept the user needs to understand. Here, it lets us specify one or more aggregations to be applied over a group. A way of specifying multiple operations or results is common in other languages. For example, anonymous types in LINQ play the same role.
3. The editor tooling available for Python is limited – editors that provide auto-completion rely on a mix of advanced static analysis and simple (not always correct) hints and often fail for chained operations such as the one in our example[1]. Statically-typed languages provide better tooling, but at the cost of higher complexity[2].
4. In the Python example (as well as in most other data manipulation libraries[3]), column names are specified as strings. This makes static checking of column names and auto-completion help difficult. For example, "Gold" is a valid column name when calling sort_values – we know that because it is a key of the dictionary passed to agg before.

We use the above four points as guiding principles for the design discussed in the rest of this section. We unify as many distinct languages constructs as possible by making member access the primary operation (Section 3.1); we use simple nominal typing to enable auto-completion (Section 3.2); we track column names using the pivot type provider (Section 3.3) and we use operation-chaining via member access for constructing dictionaries (Section 3.4).

[1] For an anecdotal evidence, see for example: `stackoverflow.com/questions/25801246/`
[2] Again, See `fslab.org/Deedle` and `extremeoptimization.com/Documentation/Data_Frame/Data_Frames.aspx`
[3] deedle, etc.

3.1 Unifying language constructs with member access

LISP is the best example of a language that unifies many language constructs using a single form. In LISP, everything is an s-experssion. That is, either a list or a symbol. In contrast, a typical data processing language uses a number of distinct constructs including indexers (for range selection and filtering), method calls (for transformations) and named parameters (for further configuration). Consider filtering and sorting:

```
data[data["Games"] == "Rio (2016)"]       ❶
data.filter(fun row → row.Games == "Rio (2016)")     ❷
data.sort_values(by = "Gold", ascending = False)     ❸
```

Python and pandas uses indexers for filtering ❶ which can alternatively be written (e.g. in LINQ) using a method taking a predicate as a lambda function ❷. Operations that are parameterized only by column name, such as sorting in pandas ❸ are often methods with named parameters.

We aim to unify the above examples using a single language construct that offers a high-level programming model[4] and can be supported by modern tooling (as discussed in Section 3.2). Member access provides an extremely simple programming construct that is, in conjunction with the type provider mechanism capable of expressing the above data transformations in a uniform way:

```
data.«sort data».«by Gold descending»     ❶
data.«filter data».«Games is».«Rio (2016)»     ❷
```

The member names tend to be longer and descriptive and we write them using «...». The names are not usually typed by the user (see Section 3.2) and so the length is not an issue when writing code. The above two examples illustrate two interesting aspects of our approach.

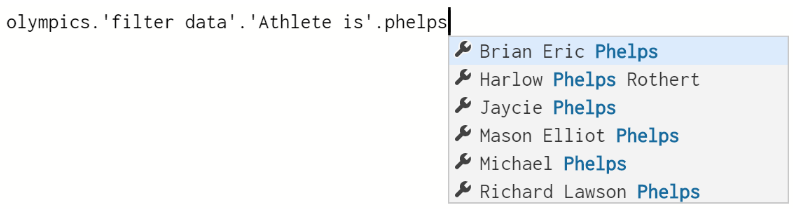
Members, type providers, discoverability. When sorting ❶ the member that specifies how sorting is done includes the name of the column. This is possible because the pivot type provider tracks the column names (see Section 3.3) and provides members based on the available columns suitable for use as sort keys. When filtering ❷, the member «Rio (2016)» is provided based on the values in the data source (we discuss this furter in Section 3.2).

These two examples illustrate that member access can be expressive, but it requires huge number of types with huge number of members. Type providers address this by integration with the type system (formalized in Section 4) that discovers members lazily. This is why approaches based on code generation or pre-processors would not be viable.

Using descriptive member names is only viable when the names are discoverable. The above code could be executed in a dynamically-typed langue that allows custom message-not-understood handlers, but it would be impossible to get the name right when writing it. Our approach relies on discovering names through auto-completion as discussed in Section 3.2.

Are members expressive enough? no... but it works fine! See Section 7. there are ways to make it more expressive too.

[4] In contrast, s-expressions in LISP are typically used as a lower-level encoding of higher-level constructs. For example lambda symbol encodes a lambda function as an s-expression.



```
olympics.'filter data'.'Athlete is'.phelps
  Brian Eric Phelps
  Harlow Phelps Rothert
  Jaycie Phelps
  Mason Elliot Phelps
  Michael Phelps
  Richard Lawson Phelps
```

Figure 1 Auto-completion offering the available values of the athlete name column

3.2 Dot-driven development

Autocompletion - see figure.

Abstraction and data. - Needs concrete data set or a sample, but that's fine

Values vs. types. - provide values? Could be «Games is»("Rio (2016)")

3.3 Column names

The last two points are more interesting – proponents of static typing would correctly point out that both member names and column names can be tracked by a more sophisticated type system. However, powerful types usually increase the complexity of the programming language.

We use type providers to provide available options (e.g. something)

Considerations

- renaming columns

3.4 Dictionary syntax

We create the "and blah. and blah. then" pattern (e.g. grouping and sorting)

```
olympics
  .«filter data».«Games is».«Rio (2016)».then
  .«group data».«by Athlete».«sum Gold».then
  .«sort data».«by Gold descending».then
  .«paging».take(8)
  .«get series».«with key Athlete».«and value Gold»
```

Introduction

```
olympics
  .«filter data».«Games is».«Rio (2016)».then
  .«group data».«by Athlete».«sum Gold».then
  .«sort data».«by Gold descending».then
  .«paging».take(8)
  .«get series».«with key Athlete».«and value Gold»
```

Details - Grouping - Filtering - Joining

4 Formalising The Gamma script

Target language is super simple OO language, but we can use it to encode very fancy types!!

4.1 Relational algebra

$$\begin{array}{rcll} R & = & \Pi_{f_1,\ldots,f_n}(R) & \text{get columns} \\ & | & \sigma_{\varphi}(R) & \text{filter by predicate} \\ & | & \tau_{f_1,\ldots,f_n}(R) & \text{sort by fields} \\ & | & \Phi_{f,\rho_1,\ldots,\rho_n}(R) & \text{group by } f \text{ and aggregate} \end{array}$$

$$\begin{array}{rcl} \rho & = & \mathsf{count} \\ & | & \mathsf{sum}\ f \\ & | & \mathsf{dist}\ f \\ & | & \mathsf{conc}\ f \end{array}$$

4.2 Foo calculus

$$\begin{array}{rcl} \tau & = & \mathsf{num} \mid \mathsf{string} \mid \mathsf{series}\langle \tau_1, \tau_2 \rangle \mid \mathsf{Query} \\ l & = & \mathsf{type}\ C(\overline{x : \tau}) = \overline{m} \\ m & = & \mathsf{member}\ N : \tau = e \end{array}$$

$$(\text{foo})\,\frac{L_1; \Gamma \vdash e : C; L_2 \qquad ((\mathsf{type}\ C(\overline{x:\tau}) = ..\ \mathsf{member}\ N_i : \tau_i = e_i\ ..), L) \in L_2}{L_1; \Gamma \vdash e.N_i : \tau_i; L_2 \cup L}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{main}(F) = C \mapsto (l, L_1 \cup \ldots \cup L_4)} \\
\quad l = \textsf{type}\ C(x : \mathsf{Query}) = & \\
\qquad \textsf{member}\ \text{«drop fields»} : C_1 = C_1(x) & \text{where } C_1, L_1 = \mathsf{drop}(F) \\
\qquad \textsf{member}\ \text{«sort data»} : C_2 = C_2(x) & \text{where } C_2, L_2 = \mathsf{sort}(F) \\
\qquad \textsf{member}\ \text{«group data»} : C_3 = C_3(x) & \text{where } C_3, L_3 = \mathsf{group}(F) \\
\qquad \textsf{member}\ \text{«get series»} : C_4 = C_4(x) & \text{where } C_4, L_4 = \mathsf{get\text{-}key}(F)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{drop}(F) = C, \{l\} \cup \bigcup L_f} \\
\quad l = \textsf{type}\ C(x : \mathsf{Query}) = & \\
\qquad \textsf{member}\ \mathsf{then} : C = C(x) & \text{where } C, L = \mathsf{main}(F) \\
\qquad \textsf{member}\ \text{«drop } f\text{»} : C_f = & \forall f \in \mathsf{dom}(F) \text{ where } C_f, L_f = \mathsf{drop}(F') \\
\qquad\quad C_f(\Pi_{\mathsf{dom}(F')}(x)) & \quad \text{and } F' = \{f' \mapsto \tau' \in F, f' \neq f\}
\end{array}$$

Figure 2 Foo

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{get\text{-}key}(F) = C, \{l\} \cup \bigcup L_f} \\
\quad l = \textsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\
\qquad \textsf{member}\ \text{«with key } f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{get\text{-}val}(F, f)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{get\text{-}val}(F, f_k) = C, \{l\} \cup \bigcup L_f} \\
\quad l = \textsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \{f\} \text{ where} \\
\qquad \textsf{member}\ \text{«and value } f\text{»} : C_f = C_f(\Pi_{f_k, f_v}(x)) & \quad C_f \text{ defined as below} \\
\quad l_f = \textsf{type}\ C_f(x : \mathsf{Query}) = & \quad \text{and } \tau_k, \tau_v \text{ such that} \\
\qquad \textsf{member}\ \mathsf{series} : \mathsf{series}\langle \tau_k, \tau_v \rangle = x & \quad \tau_k = F(f_k), \tau_v = F(f)
\end{array}$$

Figure 3 Foo

5 Pivot type provider

Omit paging, because it's boring

Our formalization of the pivot type provider follows the style used by Petricek et al. [?] in their formalization of F# Data. In our case, a type provider is a function that takes a schema F (mapping from field names to field types) and produces a class C together with collection of other class defintins L that are used by C:

$$\mathsf{provider}(F) = C, L \quad \text{where } F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$$

The provided class C has a constructor taking Query. It may be a class with further members that allow refining the query or a class with a single member series that returns a value of type $\mathsf{series}\langle\tau_1, \tau_2\rangle$. In the second case, we can evaluate the provided transformation on input R using an expression ($\textsf{new}\ C(R)$).series.

Top-level type. The top-level type allows the user to select a transformation. The members return objects with members that allow specifying further properties of each transformation.

$$\begin{array}{ll}
\mathsf{sort}(F) = C, \{l\} \cup \bigcup L_f \cup \bigcup L'_f & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{by}\ f\ \mathsf{descending}\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, \{f \mapsto \mathsf{desc}\}) \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{by}\ f\ \mathsf{ascending}\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, \{f \mapsto \mathsf{asc}\})
\end{array}$$

$$\begin{array}{ll}
\mathsf{sort\text{-}and}(F, S) = C, \{l\} \cup \bigcup L_f \cup \bigcup L'_f & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{and}\ f\ \mathsf{descending}\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, S \cup \{f \mapsto \mathsf{desc}\}) \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{and}\ f\ \mathsf{ascending}\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, S \cup \{f \mapsto \mathsf{asc}\}) \\
\qquad\qquad \mathsf{member}\ \mathsf{then} : C = C(\tau_S(x)) & \text{where } C, L = \mathsf{main}(F)
\end{array}$$

Figure 4 Foo bar

$$\begin{array}{ll}
\mathsf{group}(F) = C, \{l\} \cup \bigcup L_f & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{by}\ f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{agg}(F, \{f \mapsto F(f)\}, \emptyset)
\end{array}$$

$$\begin{array}{ll}
\mathsf{agg}(F, S, G) = C, \{l\} \cup \bigcup L_f \cup \bigcup L'_f & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{distinct}\ f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{agg}(F, S \cup \{f \mapsto \mathsf{num}, G \cup \{\mathsf{dist}\ f\}\}) \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{sum}\ f\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{agg}(F, S \cup \{f \mapsto \mathsf{num}\}, G \cup \{\mathsf{sum}\ f\}) \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{concat}\ f\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{agg}(F, S \cup \{f \mapsto \mathsf{string}\}, G \cup \{\mathsf{conc}\ f\}) \\
\qquad\qquad \mathsf{member}\ \text{«}\mathsf{count\ all}\text{»} : C = C(\tau_S(x)) & \text{where } C, L = \mathsf{main}(F) \\
\qquad\qquad \mathsf{member}\ \mathsf{then} : C = C(\tau_S(x)) & \text{where } C, L = \mathsf{main}(F)
\end{array}$$

Figure 5 Foo bar 2

Sorting. yadda
!!
TODO: Somehow delay L
!!
We only generate sorting methods for fields not used yet
Helper sort-and also takes sorting keys already used

6 Properties

7 Implementation

Rio case study

Acknowledgements. I want to thank . . .

A Morbi eros magna

Morbi eros magna, vestibulum non posuere non, porta eu quam. Maecenas vitae orci risus, eget imperdiet mauris. Donec massa mauris, pellentesque vel lobortis eu, molestie ac turpis. Sed condimentum convallis dolor, a dignissim est ultrices eu. Donec consectetur volutpat eros, et ornare dui ultricies id. Vivamus eu augue eget dolor euismod ultrices et sit amet nisi. Vivamus malesuada leo ac leo ullamcorper tempor. Donec justo mi, tempor vitae aliquet non, faucibus eu lacus. Donec dictum gravida neque, non porta turpis imperdiet eget. Curabitur quis euismod ligula.