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, type state, 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 tables, aggregation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

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 a 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 in a transparent way.

Spreadsheets made data exploration accessible to a large number of people, but operations performed on spreadsheets 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, making spreadsheets error-prone [X]. 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.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces; Article No. 23; pp. 23:1–23:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
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 finding the operation names (groupby, sort_values, head, ...) often requires using internet search. 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 programming mechanism. Finding top 8 athletes by the number of gold medals from Rio 2016 can be written as (in source 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 enables auto-completion that provides a list of available members when writing and modifying code. The members (such as «by Gold descending») are generated by a type provier based on the knowledge of the data source and transformations applied so far – thus only valid and meaningful operations are offered. The rest of the paper gives a detailed analysis and description of the mechanism.

Contributions. This paper explores an interesting particular corner of the programming language design space. We support it by a detailed analysis (Section 3), formal treatment (Section 6) and an implementation with a case study (Section 8). Our 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 6) 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 3.2).
- Our mechanism illustrates how to embed fancy types [X] into a simple nominally-typed programming language (Section 4). We track names and types of available columns of the manipulated data set (using a mechanism akin to row types), but the same mechanism can be used for embedding other advanced typing schemes into any Java-like language.
- We implement the language (`github.com/the-gamma`), make it available as a JavaScript component that can be used to build transparent data-driven visualizations (`thegamma.net`) and discuss a case study visualizing fun facts about Olympic medals (Section 8).
- We formalize the language (Section 5) and the type provider for data exploration (Section 6) and show that queries constructed using the type provider are valid (Section 7). Our formalization also covers the laziness of type providers, which is an important aspect not covered in the existing literature.

2 Using type providers in a novel way

The work presented in this paper consists of a simple nominally-typed host language and the pivot type provider, which generates 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. A type provider is a compile-time component that imports external information source into a programming language [X]. It provides two things to the compiler or editor hosting it: a type signature that models the external source using structures understood by the host language (e.g. types) and 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. The following accesses CO2 emissions by country in 2010:

```
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 obtained from 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, the code looks as follows:

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

Here, a runtime library consists of a data series type (mapping from keys to values) and the getByYear function that downloads data for a specified indicator represented by an ID. 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 of the type provider 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 importing external data sources (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 5, 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 5.

3 Simplifying data scripting languages

In Section 1, we contrasted a data exploration script written using a popular Python library pandas with a script written using the pivot type provider. In this section, we analyze what makes the Python code complex (Section 3.1) and how our design simplifies it.

3.1 What makes data exploration scripts complex

We consider the Python example from Section 1 for concreteness, but the following four points are shared with other commonly used libraries and languages. We use them to guide our alternative design discussed in the rest of this section.

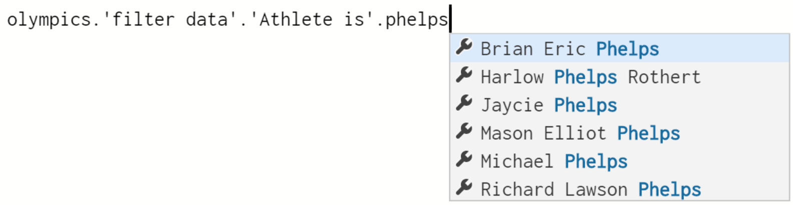
- The filtering operation is written using indexing [. . .] while all other operations are written using member invocation with (optionally named) parameters. In the first case, we write an expression `olympics["Games"] == "Rio (2016)"` returning a vector of Booleans while in the other, we specify a parameter value using `by = "Gold"`. In other languges, a parameter can also be a lambda function specifying a predicate or a transformation.
- 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 similar way of specifying multiple operations or results is common in other languages. For example, anonymous types in LINQ play the same role.
- 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].
- 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 difficult. For example, `"Gold"` is a valid column name when calling `sort_values`, but we only know that because it is a key of the dictionary passed to `agg` before.

In our design, we unify many distinct languages constructs by making member access the primary operation (Section 3.2); we use simple nominal typing to enable auto-completion (Section 3.3); we use operation-chaining via member access for constructing dictionaries (Section 3.4) and we track column names using the pivot type provider (Section 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.



```
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 Unifying language constructs with member access

LISP is perhaps the best example of a language that unifies many distinct 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
```

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 3.3). 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».then            ❶
data.«filter data».«Games is».«Rio (2016)».then       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 3.3) 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 4) 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 3.3).

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 5) that discovers members lazily. This is why approaches based on code generation or pre-processors would not be viable.

[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.

```
«drop columns»                    «group data»
  → «drop Athlete»                  → «by Athlete»
  → «drop Discipline»                   → «distinct Discipline»
  → «drop Year»                         → «average Year»
                                        → «sum Year»
«sort data»
  → «by Athlete»                    → «by Year»
  → «by Athlete descending»             → «distinct Athlete»
  → «by Discipline»                     → «distinct Discipline»
```

Figure 2 Subset of members provided by the pivot type provider

Using descriptive member names is only viable when the names are discoverable. The above code could be executed in a dynamically-typed languge 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 3.3.

Expressivity of members. Using member access as the primary mechanism for programming reduces the expressivity of the language – our aim is to create a domain-specific language for data exploration, rather than a general purpose language[5]. For this purpose, the sequential nature of member accesses matches well with the sequential nature of data transformations.

The members provided, for example, for filtering limit the number of conditions that can be written, because the user is limited to choosing one of the provided members. As discussed in the case study based on our implementation (Section 8), this appears sufficient for many common data exploration tasks. The mechanism could be made more expressive, but we leave this for future work – for example, the type provider could accept or reject member names written by the user (as in internet search) rather than providing names from which the user can choose (as in web directories).

3.3 Tooling and dot-driven development

Source code editors for object-based languages with nominal type sytems often provide auto-completion for members of objects. This combination appear to work extremely well; the member list is a complete list of what might follow after typing "dot" and it can be easily obtained for an instance of known type. The fact that developers can often rely on just typing "dot" and choosing an appropriate member led to a semi-serious phrase dot-driven development, that we (equally semi-seriously) adopt in this paper.

Type providers in F# rely on dot-driven development when navigating through data. When writing code to access current temperature Ldn.Main.Temp in Section 2, the auto-completion offers various available properties, such as Wind and Clouds once "dot" is typed after Ldn.Main. Other type providers [X] follow a similar pattern. It is worth noting that despite the use of nominal typing, the names of types rarely explicitly appear in code – we do not need to know the name of the type of Ldn.Main, but we need to know its members. Thus the type name can be arbitrary [X] and is used more as a lookup key.

[5] Designing a general purpose language based on member access is a separate interesting problem.

The pivot type provider presented in this paper uses dot-driven development for suggesting transformations as well as possible values of parameters. This is illustrated in Figure 1 where the user wants to obtain medals of a specific athlete and is offered a list of possible names. The editor filters the list as the user starts typing the required name.

In Figure 2, we list a subset of the provided members that were used in the example in Section 1. After choosing «sort data», the user is offered the possible sorting keys and directions. After choosing «group data», the user first selects the grouping key and then can choose one or more aggregations that can be applied on other columns of the group. Thus an enitre data transformation (such as choosing top 8 athletes by the number of gold medals) can be constructed using dot-driven development.

Values vs. types. As the Figure 1 illustrates, the pivot type provider sometimes blurs the distinction between values and (members of) types. In the example in Section 1, "Rio (2016)" is a string value in Python, but a statically-typed member «Rio (2016)» when using pivot type provider. This is a recurring theme in type provider development[6].

Our language supports method calls such as head(8) and an alternative design for filtering would be to provide a method such as «Games is»("Rio (2016)"). However, the fact that we can offer possible values largely simplifies writing of the script for the most common case when the user is interested in one of the known values.

Unlike in traditional development, a data scientist doing data exploration often has the entire data set available. The pivot type provider uses this when offering possible values for filtering (Section X), but all other operations (Section Y) require only meta-data (names and types of columns). Following the example of type providers for structured data formats [X], the schema could be inferred from a representative sample.

3.4 Expressing structured logic using members

In the motivating example, the agg method takes a dictionary that specifies one or more aggregates to be calculated over a group. We sum the number of gold medals, but we could also sum the number of silver and bronze medals, concatenate names of teams for the athlete and perform other aggregations. In this case, we provide a nested structure (list of aggregations) as a parameter of a single operation (grouping).

This is an interesting case, because when encoding program as a sequence of member accesses, there is no built-in support for nesting. In the pivot type provider, we use "then" design pattern to provide operations that require nesting. The following example specifies multiple aggregations and then sorts data by multiple keys:

```
olympics.
  «group data».«by Athlete».
    .«sum Gold».«sum Silver».«concat Team».then      ❶
  .«sort data».
    .«by Gold descending».«and Silver descending».then      ❷
```

When grouping, we sum the number of gold and silver medals and concatenates team names ❶. Then we sorts the grouped data using multiple sorting keys ❷ – first by number of gold medals and then by silver medals (within a group with the same number of gold medals).

[6] The Individuals property in the Freebase type provider [X] imports values into types in a similar way.

$$(\textit{drop-start})\ \frac{\Gamma \vdash e : [f_1\!:\!\tau_1, \ldots, f_n\!:\!\tau_n]}{\Gamma \vdash e.\text{«drop columns»} : [f_1\!:\!\tau_1, \ldots, f_n\!:\!\tau_n]_{\mathsf{drop}}}$$

$$(\textit{drop-col})\ \frac{\Gamma \vdash e : [f_1\!:\!\tau_1, \ldots, f_n\!:\!\tau_n]_{\mathsf{drop}}}{\Gamma \vdash e.\text{«drop } f_i\text{»} : [f_1\!:\!\tau_1, \ldots, f_{i-1}\!:\!\tau_{i-1}, f_{i+1}\!:\!\tau_{i+1}, \ldots, f_n\!:\!\tau_n]_{\mathsf{drop}}}$$

$$(\textit{drop-then})\ \frac{\Gamma \vdash e : [f_1\!:\!\tau_1, \ldots, f_n\!:\!\tau_n]_{\mathsf{drop}}}{\Gamma \vdash e.\text{«then»} : [f_1\!:\!\tau_1, \ldots, f_n\!:\!\tau_n]}$$

Figure 3 Tracking available column names with row types and type state

The “then” pattern. Nesting is no doubt essential programming construct and it may very well be desirable to support it directly in the language, but the “then” pattern lets us express it without explicit langauge support. In both cases, the nested structure is specified by selecting one or more members and then ending the nested structure using the then member.

In case of grouping, we choose aggregations («sum Gold», «concat Team», etc.) after we specify grouping key using «by Athlete». In case of sorting, we specify the first key using «by Gold descending» and then add more nested keys using «and Silver descending». Thanks to the dot-driven development and the “then” pattern, the user is offered possible parameter values (aggregations or sorting keys) even when creating a nested structure. This is harder to do with general-purpose nesting as ordinary languages (even domain-specific languages) often provide many different ways for creating a nested structure.

Renaming columns. The pivot type provider automatically chooses names for the columns obtained as the result of aggregation. In the above example ❶, the resulting data set will have columns Athlete (the grouping key) together with Gold, Silver and Team (based on the aggregated columns). The user cannot currently rename the columns.

In F# type providers, this could be done using methods with static parameters [X] by writing, for example, g.«sum Gold as»⟨"Total Gold"⟩(). In F#, the value of the static parameter (here, "Total Gold") is passed to the type provider, which can use it to generate the type signature of the method and the return type with member name according to the value of the static parameter.

4 Tracking column names

The last difficulty with data scripting discussed in Section 3.1 is that pandas (and other data exploration libraries, even for statically-typed languages) track column names as strings at runtime, making code error-prone and auto-complete on column names difficult to provie. Proponents of static typing would correctly point out that column names and their types can be tracked by a more sophisticated type system.

In this section, we discuss our approach – we track column names statically using a mechanism that is inspired by row types and type state (Section 4.1), however we embed the mechanism using type providers into a simple nominal type system (Section 4.2). This way, the host language for the pivot type provider can be extremely simple – and indeed, the mechanism could be added to languages such as Java or TypeScript with minimal effort.

4.1 Using row types and type state

There are several common data transformations that modify the structure of the data set and affect what columns (and of what types) are available. When grouping and aggregating data, the resulting data set has columns depending on the aggregates calculated. Another simpler operation is adding or removing a column from the data set. For example, given the Olympic medals data set, we can drop games and year as follows:

```
olympics.«drop columns».«drop Games».«drop Year».then
```

Operations on the type of rows in the data set can be captured using row types [X]. In addition, we need to annotate type with a form of type state [Y] to restrict what operations are available. When dropping columns, we first access the «drop columns» member, which sets the state to a state where we can drop individual columns using «drop f». The then member can then be used to complete the operation and choose another transformation.

To illustrate tracking of columns using row types and type state, consider a simple language with variables (representing external data sources) and member access. Types can be either primitive types α, types annotated with a type state lbl or row type with fields f:

$$\begin{array}{rcl} e & = & v \mid e.N \\ \tau & = & \alpha \mid \tau_{\mathsf{lbl}} \mid [f_1 : \tau_1, \ldots, f_n : \tau_n] \end{array}$$

Typing rules for members that are used to drop columns are shown in Figure 3. When «drop columns» is invoked on a record, the type is annotated with a state drop (*drop-start*) indicating that individual columns may be dropped. The then operation (*drop-then*) removes the state label. Individual members can be removed using «drop f_i» and the (*drop-col*) rule ensures the dropped column is available in the input row type and removes it.

Other data transformations could be type checked in a similar way, but there are two drawbacks. First, row types and type state (although relatively straightforward) make the host language more complex. Second, rules such as (*drop-col*) make auto-completion more difficult, because the editor needs to understand the rules and calculate what members may be invoked. This is a distinct operation from type checking and type infernece (which operate on complete programs) that needs to be formalized and implemented.

4.2 Using the pivot type provider

In our approach, the information about available fields is used by a type provider to provide types with appropriate members. This is hidden from the host language, which only sees class types with members. Provided class definitions consist of a constructor and members:

$$\begin{array}{rcl} l & = & \mathsf{type}\ C(x : \tau) = \overline{m} \\ m & = & \mathsf{member}\ N : \tau = e \end{array}$$

During type-checking, the type system keeps track of a list of provided class definitions L. Checking member access is then just a matter of finding the corresponding class definition and finding the member type:

$$(\mathit{member})\ \frac{L;\Gamma \vdash e : C \qquad \mathsf{type}\ C(x : \tau) = ..\ \mathsf{member}\ N_i : \tau_i = e_i\ .. \in L}{L;\Gamma \vdash e.N_i : \tau_i}$$

The rule, adapted from [X], does not capture laziness of type providers that is important for the pivot type provider (where the number of provided classes is potentially infinite). We discuss this aspect in Section 5.

$$
\begin{array}{rcll}
D & = & \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r}\rangle\} & \\
R & = & \Pi_{f_1,\ldots,f_n}(R) & \text{Projection – select specified column names} \\
 & | & \sigma_\varphi(R) & \text{Selection – filter rows by given predicate} \\
 & | & \tau_{f_1,\ldots,f_n}(R) & \text{Sorting – sort by specified columns} \\
 & | & \Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(R) & \text{Grouping – group by and calculate aggregates} \\
\rho & = & \mathsf{count} & \text{Count number of rows in the group} \\
 & | & \mathsf{sum}\ f & \text{Sum numerical values of the column } f \\
 & | & \mathsf{dist}\ f & \text{Count number of distinct values of the column } f \\
 & | & \mathsf{conc}\ f & \text{Concatenat string values of the column } f
\end{array}
$$

Figure 4 Relational algebra with values, sorting and aggregation

Using type providers and nominal type system seemingly hides knowledge about fields available in the data set. However, for types constructed by the pivot type provider, we can define a mapping fields that returns the fields available in the data set represented by the class. The type provider encodes the logic expressed in Section 4.1 in the following sense:

▶ **Remark 1** (Encoding of fancy types). If $\Gamma \vdash e : [f_1 : \tau_1, \ldots, f_n : \tau_n]$ using a type system defined in Figure 3 and $\Gamma \vdash e : C$ using nominal typing and C is a type provided by the pivot type provider then $\mathsf{fields}(C) = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$.

In the following two sections, we focus on formalizing the pivot type provider and the nominally typed host language. We define the fields predicate in Section 7 and use it to prove properties of the pivot type provider.

We do not fully develop the type system based on fancy types sketched in Section 4.1. However, the remark illustrates one interesting aspect of our work – the type provider mechanism makes it possible to express safety guarantees that would normally require row types and type state in a simple nominally typed language. In a similar way, type providers have been used to encode session types [X], suggesting that this is a generally useful approach.

5 Formalising the host language

Type providers often provide a thin type-safe layer over a richer untyped runtime component. In case of providers for data access (Section 2), the untyped runtime component performs operations over an external data source. In case of the pivot type provider, the untyped runtime component is a relational algebra, which we use for modelling data transformations. We formalize the relational algebra in Section 5.1, together with the object-based host language in Section 5.2.

5.1 Relational algebra with vector semantics

The focus of our work is on data aggregation and so we use a form of relational algebra with extensions for grouping and sorting [X,Y,Z]. The syntax is defined in Figure 4. We write f for column (field) names and we include definition of a data value D, which maps columns to vectors of length r storing the data (values v are defined below). Aside from standard projection Π and selection σ, our algebra includes sorting τ which takes one or more column names forming the sort key and aggregation Φ, which requires a single grouping key and several aggregations together with names of the new columns to be returned.

$$\begin{array}{l}
\Pi_{f_{p(1)},\ldots,f_{p(m)}}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\
\quad \{f_{p(1)} \mapsto \langle v_{p(1),1},\ldots,v_{p(1),r}\rangle,\ldots,f_{p(m)} \mapsto \langle v_{p(m),1},\ldots,v_{p(m),r}\rangle\}
\end{array}$$

$$\begin{array}{l}
\sigma_{\varphi}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\
\quad \{f_1 \mapsto \langle \ldots,v_{1,j},\ldots\rangle,\ldots,f_n \mapsto \langle \ldots,v_{n,j},\ldots\rangle\} \qquad (\forall j.\ \varphi\,\{f_1 \mapsto v_{1,j},\ldots,f_n \mapsto v_{n,j}\})
\end{array}$$

$$\begin{array}{l}
\tau_{f_{p(1)},\ldots,f_{p(m)}}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\
\quad \{f_1 \mapsto \langle v_{1,q(1)},\ldots,v_{1,q(r)}\rangle,\ldots,f_n \mapsto \langle v_{n,q(1)},\ldots,v_{n,q(r)}\rangle\} \quad \text{where } q \text{ permutation} \\
\qquad \text{such that } i \leq j \implies (v_{p(1),q(i)},\ldots,v_{p(m),q(i)}) \leq (v_{p(1),q(j)},\ldots,v_{p(m),q(j)})
\end{array}$$

$$\begin{array}{l}
\Phi_{f_i,\rho_1/f'_1,\ldots,\rho_m/f'_m}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\
\quad \{f'_1 \mapsto a_1,\ldots,f'_m \mapsto a_m\} \quad \text{where} \\
\qquad \{g_1,\ldots,g_s\} = \{\{l \mid k \in 1\ldots r,\ v_{i,l} = v_{i,k}\},\ l \in 1\ldots r\} \\
\qquad\quad \begin{array}{ll}
a_i = \langle |g_1|,\ldots,|g_s|\rangle & \text{when } \rho_i = \mathsf{count} \\
a_i = \langle \Sigma_{k\in g_1} v_{j,k},\ldots,\Sigma_{k\in g_s} v_{j,k}\rangle & \text{when } \rho_i = \mathsf{sum}\ f_j \\
a_i = \langle \Pi_{k\in g_1} v_{j,k},\ldots,\Sigma_{k\in g_s} v_{j,k}\rangle & \text{when } \rho_i = \mathsf{conc}\ f_j \\
a_i = \langle |\{v_{j,k}, k \in g_1\}|,\ldots,|\{v_{j,k}, k \in g_s\}|\rangle & \text{when } \rho_i = \mathsf{dist}\ f_j
\end{array}
\end{array}$$

Figure 5 Vector-based semantics for operations of the extended relational algebra

The semantics of the algbra is given in Figure 5. We use vector-based semantics to support sorting and duplicate entries, but otherwise the formalization captures the usual behaviour. In projection and sorting, we write $f_{p(1)},\ldots,f_{p(m)}$ to refer to a selection of fields from $f_1,\ldots,f_n$. Assuming $m \leq n$, p can be seen as a mapping from $\{1\ldots m\}$ to a subset of $\{1\ldots n\}$. In selection, φ is a predicate applied to a mapping from column names to values. In sorting, we assume that there is a permutation on row indices q such that the tuples obtained by selecting values according to the sort key are ordered.

The most complex operation is grouping. We need to group data by the value of the column f_i and then apply aggregations $\rho_1,\ldots,\rho_m$. To do this, we first obtain a set of groups $g_1,\ldots,g_s$ where each group represents a set of indicies of rows belonging to each group. For a given group g_i we can then obtain values of column j for rows in the group as $\{v_{j,k}, k \in g_i\}$. This is used when performing the aggregations – $|g_i|$ gives the number of rows in the group, Σ sums numerical values and Π (a slight notation abuse) concatenates string values.

5.2 Foo calculus

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

$$\begin{array}{rcl}
v & = & d \mid \mathsf{None} \mid \mathsf{Some}(v) \mid \mathsf{new}\ C(\overline{v}) \mid v_1 :: v_2 \\
e & = & d \mid op \mid e_1\ e_2 \mid \lambda x.e \mid e.N \mid \mathsf{new}\ C(\overline{e}) \\
& \mid & \mathsf{None} \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{Some}(x) \to e_1 \,|\, \mathsf{None} \to e_2 \\
& \mid & \mathsf{Some}(e) \mid e_1 = e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{nil} \\
& \mid & e_1 :: e_2 \mid \mathsf{match}\ e\ \mathsf{with}\ x_1 :: x_2 \to e_1 \,|\, \mathsf{nil} \to e_2
\end{array}$$

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{main}(F) = C \mapsto (l, L_1 \cup \ldots \cup L_4)} \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \\
\qquad\quad \mathsf{member}\ \text{«drop columns»} : C_1 = C_1(x) & \text{where } C_1, L_1 = \mathsf{drop}(F) \\
\qquad\quad \mathsf{member}\ \text{«sort data»} : C_2 = C_2(x) & \text{where } C_2, L_2 = \mathsf{sort}(F) \\
\qquad\quad \mathsf{member}\ \text{«group data»} : C_3 = C_3(x) & \text{where } C_3, L_3 = \mathsf{group}(F) \\
\qquad\quad \mathsf{member}\ \text{«get series»} : C_4 = C_4(x) & \text{where } C_4, L_4 = \mathsf{get\text{-}key}(F) \\
& \\
\multicolumn{2}{l}{\mathsf{drop}(F) = C, \{l\} \cup \bigcup L_f} \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \\
\qquad\quad \mathsf{member}\ \mathsf{then} : C = C(x) & \text{where } C, L = \mathsf{main}(F) \\
\qquad\quad \mathsf{member}\ \text{«drop } f\text{»} : C_f = & \forall f \in \mathsf{dom}(F) \text{ where } C_f, L_f = \mathsf{drop}(F') \\
\qquad\qquad\quad C_f(\Pi_{\mathsf{dom}(F')}(x)) & \quad \text{and } F' = \{f' \mapsto \tau' \in F, f' \neq f\}
\end{array}
$$

Figure 6 Foo

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

Figure 7 Foo

6 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 $(\mathsf{new}\ C(R)).\mathsf{series}$.

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.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{sort}(F) = C, \{l\} \cup \bigcup L_f \cup \bigcup L'_f} \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«by } f \text{ descending»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, \{f \mapsto \mathsf{desc}\}) \\
\qquad\qquad \mathsf{member}\ \text{«by } f \text{ ascending»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, \{f \mapsto \mathsf{asc}\})
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{sort\text{-}and}(F, S) = C, \{l\} \cup \bigcup L_f \cup \bigcup L'_f} \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«and } f \text{ descending»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, S \cup \{f \mapsto \mathsf{desc}\}) \\
\qquad\qquad \mathsf{member}\ \text{«and } f \text{ ascending»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, S \cup \{f \mapsto \mathsf{asc}\}) \\
\qquad\qquad \mathsf{member}\ \mathsf{then} : C = C(\tau_S(x)) & \text{where } C, L = \mathsf{main}(F)
\end{array}$$

Figure 8 Foo bar

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{group}(F) = C, \{l\} \cup \bigcup L_f} \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«by } f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{agg}(F, \{f \mapsto F(f)\}, \emptyset)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{agg}(F, S, G) = C, \{l\} \cup \bigcup L_f \cup \bigcup L'_f} \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S) \text{ where} \\
\qquad\qquad \mathsf{member}\ \text{«distinct } f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{agg}(F, S \cup \{f \mapsto \mathsf{num}, G \cup \{\mathsf{dist}\ f\}\}) \\
\qquad\qquad \mathsf{member}\ \text{«sum } f\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{agg}(F, S \cup \{f \mapsto \mathsf{num}\}, G \cup \{\mathsf{sum}\ f\}) \\
\qquad\qquad \mathsf{member}\ \text{«concat } f\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{agg}(F, S \cup \{f \mapsto \mathsf{string}\}, G \cup \{\mathsf{conc}\ f\}) \\
\qquad\qquad \mathsf{member}\ \text{«count all»} : C = C(\tau_S(x)) & \text{where } C, L = \mathsf{main}(F) \\
\qquad\qquad \mathsf{member}\ \mathsf{then} : C = C(\tau_S(x)) & \text{where } C, L = \mathsf{main}(F)
\end{array}$$

Figure 9 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

7 Properties

8 Implementation

Rio case study

```
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»
```

Acknowledgements. I want to thank . . .

A Sample of the Olympic medals data set

```
Games,Year,Discipline,Athlete,Team,Gender,Event,Medal,Gold,Silver,Bronze
Athens (1896), 1896, Swimming, Alfred Hajos, HUN, Men, 100m freestyle men, Gold, 1, 0, 0
Athens (1896), 1896, Swimming, Otto Herschmann, AUT, Men, 100m freestyle men, Silver, 0, 1, 0
Athens (1896), 1896, Swimming, Dimitrios Drivas, GRE, Men, 100m freestyle for sailors men, Bronze, 0, 0, 1
Athens (1896), 1896, Swimming, Ioannis Malokinis, GRE, Men, 100m freestyle for sailors men, Gold, 1, 0, 0
Athens (1896), 1896, Swimming, Spiridon Chasapis, GRE, Men, 100m freestyle for sailors men, Silver, 0, 1, 0
```