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.

```
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 7). 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 7).
- 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 6.2). 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.

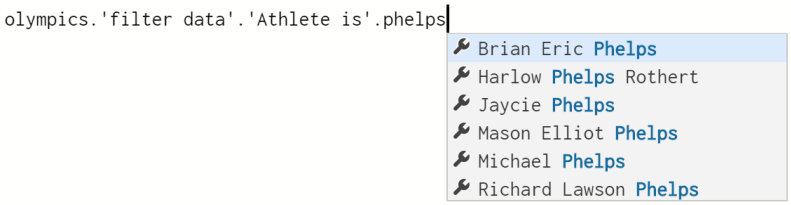

Figure 1 Auto-completion offering the available values of the athlete name column

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»
- → «drop Athlete»
- → «drop Discipline»
- → «drop Year»

«sort data»
- → «by Athlete»
- → «by Athlete descending»
- → «by Discipline»

«group data»
- → «by Athlete»
  - → «distinct Discipline»
  - → «average Year»
  - → «sum Year»
- → «by Year»
  - → «distinct Athlete»
  - → «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 7), 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 `Idn.Main.Temp` in Section 2, the auto-completion offers various available properties, such as `Wind` and `Clouds` once "dot" is typed after `Idn.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 `Idn.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.

$$
\begin{array}{rcll}
D & = & \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r}\rangle\} & \\
e & = & \Pi_{f_1,\ldots,f_n}(e) & \text{Projection – select specified column names} \\
 & | & \sigma_{\varphi}(e) & \text{Selection – filter rows by given predicate} \\
 & | & \tau_{f_1 \mapsto \omega_1,\ldots,f_n \mapsto \omega_n}(e) & \text{Sorting – sort by specified columns} \\
 & | & \Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(e) & \text{Grouping – group by and calculate aggregates} \\
\omega & = & \mathsf{desc} \mid \mathsf{asc} & \text{Sort order – descending or ascending} \\
\rho & = & \mathsf{count} & \text{Count number of rows in the group} \\
 & | & \mathsf{sum}\ f & \text{Sum numerical values of the column } f \\
 & | & \mathsf{dist}\ f & \text{Count number of distinct values of the column } f \\
 & | & \mathsf{conc}\ f & \text{Concatenat string values of the column } f
\end{array}
$$

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 6.2 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 and runtime

Type providers often provide a thin type-safe layer over richer untyped runtime components. In case of providers for data access (Section 2), the untyped runtime component performs operations over external data sources. In case of the pivot type provider, the untyped runtime component is a relational algebra 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 columns forming the sort key (with sort order ω) and aggregation Φ, which requires a single grouping key and several aggregations together with names of the new columns to be returned.

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

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

$$\tau_{f_{p(1)}\mapsto\omega_1,\ldots,f_{p(m)}\mapsto\omega_m}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\ \{f_1 \mapsto \langle v_{1,q(1)},\ldots,v_{1,q(r)}\rangle,\ldots,f_n \mapsto \langle v_{n,q(1)},\ldots,v_{n,q(r)}\rangle\} \quad \text{where } q \text{ permutation} \\ \quad \text{such that } \forall i,j.\ i \le j \implies (u_{1,i},\ldots,v_{m,i}) \le (v_{1,j},\ldots,v_{m,j}) \text{ where} \\ \qquad u_{k,l} = v_{p(k),q(l)} \quad (\text{when } \omega_k = \mathsf{asc}) \\ \qquad u_{k,l} = -v_{p(k),q(l)} \quad (\text{when } \omega_k = \mathsf{desc})$$

$$\Phi_{f_i,\rho_1/f'_1,\ldots,\rho_m/f'_m}\{f_1 \mapsto \langle v_{1,1},\ldots,v_{1,r}\rangle,\ldots,f_n \mapsto \langle v_{n,1},\ldots,v_{n,r}\rangle\} \rightsquigarrow \\ \{f'_1 \mapsto a_1,\ldots,f'_m \mapsto a_m, f_i \mapsto b\} \quad \text{where} \\ \quad \{g_1,\ldots,g_s\} = \{\{l \mid k \in 1\ldots r,\ v_{i,l} = v_{i,k}\},\ l \in 1\ldots r\} \\ \qquad b = \langle v_{i,k_1},\ldots,v_{i,k_s}\rangle \qquad \text{where } k_j \in g_j \\ \qquad a_i = \langle |g_1|,\ldots,|g_s|\rangle \qquad \text{when } \rho_i = \mathsf{count} \\ \qquad a_i = \langle \Sigma_{k\in g_1} v_{j,k},\ldots,\Sigma_{k\in g_s} v_{j,k}\rangle \qquad \text{when } \rho_i = \mathsf{sum}\ f_j \\ \qquad a_i = \langle \Pi_{k\in g_1} v_{j,k},\ldots,\Sigma_{k\in g_s} v_{j,k}\rangle \qquad \text{when } \rho_i = \mathsf{conc}\ f_j \\ \qquad a_i = \langle |\{v_{j,k} \mid k \in g_1\}|,\ldots,|\{v_{j,k} \mid k \in g_s\}|\rangle \qquad \text{when } \rho_i = \mathsf{dist}\ f_j$$

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 $m \le n$, 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 given sort key are ordered. The auxiliary definition $u_{k,l}$ negates the number to reverse the sort order when descending order is required.

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 $\{v_{j,k} \mid k \in g_i\}$. This is used to calculate the resulting data set – the field f_i becomes a new column formed by the group keys (obtained by picking one of the indices from g_j for each group); other fields are calculated by aggregating data in various ways – $|g_i|$ 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 with lazy context

We model the host language using a variant of the Foo calculus [X]. The core of the calculus models a simple object-based language with objects and members. The syntax of the language is shown in Figure 6. The relational algebra defined in Figure 4 is included in the Foo calculus as a model of the runtime components of the pivot type provider – the values include the data value D and the expressions include all the operations of the relational algebra.

$$
\begin{array}{rcl}
v & = & C(v) \mid \mathsf{series}\langle\tau_1,\tau_2\rangle(v) \mid n \mid s \mid D \\
e & = & C(e) \mid \mathsf{series}\langle\tau_1,\tau_2\rangle(e) \mid x \mid v \mid e.N \mid \ldots \\
E & = & C(E) \mid \mathsf{series}\langle\tau_1,\tau_2\rangle(E) \mid E.N \\
 & \mid & \Pi_{f_1,\ldots,f_n}(E) \mid \sigma_{\varphi}(E) \mid \tau_{f_1,\ldots,f_n}(E) \mid \Phi_{f,\rho_1/f_1,\ldots,\rho_m/f_m}(E) \\
 & & \\
\tau & = & C \mid \mathsf{num} \mid \mathsf{string} \mid \mathsf{series}\langle\tau_1,\tau_2\rangle \mid \mathsf{Query} \\
l & = & \mathsf{type}\ C(x:\tau) = \overline{m} \\
m & = & \mathsf{member}\ N : \tau = e
\end{array}
$$

$$
(member)\ \frac{L(C) = (\mathsf{type}\ C(x:\tau) = \ldots\ \mathsf{member}\ N_i : \tau_i = e_i\ \ldots), L'}{(C(v)).N_i \rightsquigarrow_L e_i[x \leftarrow v]}
$$

$$
(context)\ \frac{e \rightsquigarrow_L e'}{E[e] \rightsquigarrow_L E[e']}
$$

Figure 6 Syntax and remaining reduction rules of the Foo calculus

The Foo calculus includes two special types. Query is a type of data and queries constructed using the relational algebra. The type $\mathsf{series}\langle\tau_1,\tau_2\rangle$ models a data series mapping keys of type τ_1 to values of type τ_2. A series is created as a result of a query written using the pivot type provider. In the actual implementation, a series can be passed to a charting library. A series is a statically-typed wrapper over a Query value and the proofs in Section 6.2 show that a series obtained from the pivot type provider contains keys and values of matching types.

Reduction rules. The reduction relation $\rightsquigarrow_L$ is prameterized by a function L that maps class names to class definitions, together with nested classes associated with the class definition (used during type checking as discussed below). The map is not used in the reduction rules for the relational algebra, given in Figure 5 and so it was omitted there.

The remaining reduction rules are given in Figure 6. The (*member*) rule performs lookup using $L(C)$ to find the definition of the member that is being accessed and then it reduces member access by substituting the evaluated constructor argument v for a variable x. We assume standard capture-avoiding substitution $[x \leftarrow v]$. The rule ignores the nested class definitions L'. The (*context*) rule performs reduction in an evaluation context E.

Type checking. One interesting aspect of type checking with type providers is that type providers can provide potentially infinite number of types. In F#, the types are provided lazily as the type checker explores parts of the type space used by the program [X]. Consider:

```
olympics.«group data».«by Athlete».«sum Gold».then
```

The type checker initially knows the type of olympics is a class C_1 with member «group data» and it knows that the type of this member is C_2. However, it only needs to obtain full definition of C_2 when checking the member «by Athlete». Types of other member of C_1 remain unevaluated when they do not appear in the source code. This aspect of type providers have been omitted in previous work [X,Y], but it is necessary for the pivot type provider. The typing rules given are written using the following judgement:

$$L_1; \Gamma \vdash e : \tau; L_2$$

$$(num)\ \frac{}{L;\Gamma \vdash n : \mathsf{num}; L} \qquad (string)\ \frac{}{L;\Gamma \vdash s : \mathsf{string}; L} \qquad (var)\ \frac{}{L;\Gamma, x:\tau \vdash x : \tau; L}$$

$$(data)\ \frac{L;\Gamma \vdash v_{i,j} : \tau; L \quad \tau \in \{\mathsf{num}, \mathsf{string}\}}{L;\Gamma \vdash \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r}\rangle\} : \mathsf{Query}; L}$$

$$(proj)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \Pi_{f_1,\ldots,f_n}(e) : \mathsf{Query}; L_2} \qquad (sort)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \tau_{f_1,\ldots,f_n}(e) : \mathsf{Query}; L_2}$$

$$(sel)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \sigma_{\varphi}(e) : \mathsf{Query}; L_2} \qquad (group)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(e) : \mathsf{Query}; L_2}$$

$$(series)\ \frac{L_1;\Gamma \vdash e : \mathsf{Query}; L_2}{L_1;\Gamma \vdash \mathsf{series}\langle\tau_1,\tau_2\rangle(e) : \mathsf{series}\langle\tau_1,\tau_2\rangle; L_2}$$

$$(new)\ \frac{L_1;\Gamma \vdash e : \tau, L_2 \qquad L_2(C) = (\mathsf{type}\ C(x:\tau) = \ldots), L}{L_1;\Gamma \vdash C(e) : C; L_2 \cup L}$$

$$(member)\ \frac{\begin{array}{c}L_1;\Gamma \vdash e : C; L_2 \qquad L_2 \cup L;\Gamma, x:\tau \vdash e_i : \tau_i; L_3 \\ L_2(C) = (\mathsf{type}\ C(x:\tau) = .. \ \mathsf{member}\ N_i : \tau_i = e_i\ ..), L\end{array}}{L_1;\Gamma \vdash e.N_i : \tau_i; L_3}$$

Figure 7 Type-checking of Foo expressions with lazy context

The judgement states that given class definitions L_1 and a variable context Γ, the type of expression e is τ and the type checking evaluated class definitions that are now included in L_2. The structure of class definitions L_1 is a function mapping a class name C to a pair consisting of the definition and a function that provides definitions of delayed classes:

$$L_1(C) = \mathsf{type}\ C(x:\tau) = \overline{m}, L$$

The class C may use classes defined in L_1, but also delayed classes from L. This models laziness as L is a function that may never be evaluated. Since L is potentially infinite, we cannot check class definitions upfront as in typical object calculi [X]. Instead, we check that that members are well typed as they appear in the source code, which matches the behaviour of F# type providers. In general, this means that L may contain classes with incorrectly typed members. We prove that this is not the case for the pivot type provider (Section 6.2).

The rules that define type checking are shown in Figure 7. The two rules that force the discovery of new classes are (*new*) and (*member*). In (*new*), we find the class definition and delayed classes using $L_2(C)$. We treat functions as sets and join L_2 with delayed classes defined by L using $L_2 \cup L$. In (*member*), we obtain the class definition and discover delayed classes in the same way, but we also check that the body of the member is well-typed.

The rules for primitive types and variables are standard. Input data (*data*) is of type Query and all the operations of relational algebra take a value of type Query and produce a result of type Query. An untyped Query value can be converted into a series type (*series*) of any type – in the pivot type provider, the Query type is always hidden and users can only work with type-safe series type. The bridging of typed and untyped parts of the language resembles the boundary between static and dynamic typing in gradually typed languages.

$$\mathsf{pivot}(F) = C, \{C \mapsto (l, L_1 \cup \ldots \cup L_4)\} \quad ❶$$

$l =$ type $C(x : \mathsf{Query}) =$
- member «drop columns» : $C_1 = C_1(x)$ where $C_1, L_1 = \mathsf{drop}(F)$
- member «sort data» : $C_2 = C_2(x)$ where $C_2, L_2 = \mathsf{sort}(F)$
- member «group data» : $C_3 = C_3(x)$ where $C_3, L_3 = \mathsf{group}(F)$
- member «get series» : $C_4 = C_4(x)$ where $C_4, L_4 = \mathsf{get\text{-}key}(F)$

$$\mathsf{get\text{-}key}(F) = C, \{C \mapsto (l, \bigcup L_f)\} \quad ❷$$

$l =$ type $C(x : \mathsf{Query}) =$
- member «with key f» : $C_f = C_f(x)$ $\forall f \in \mathsf{dom}(F)$ where $C_f, L_f = \mathsf{get\text{-}val}(F, f)$

$$\mathsf{get\text{-}val}(F, f_k) = C, \{C \mapsto (l, \{\})\} \quad ❸$$

$l =$ type $C(x : \mathsf{Query}) =$
- member «and value f» : $\mathsf{series}\langle \tau_k, \tau_v \rangle =$ $\mathsf{series}\langle \tau_k, \tau_v \rangle(\Pi_{f_k, f}(x))$ ❹

$\forall f \in \mathsf{dom}(F) \setminus \{f_k\}$ where $\tau_k = F(f_k),\ \tau_v = F(f)$

Figure 8 Pivot type provider – entry-point type and accessing transformed data

6 Formalising the pivot type provider

In an implementation, a type provider is an executable component that is called by the compiler and the editor to provide information about types on demand. In our formalization, we follow the style used by Petricek et al. [X], but we add laziness as discussed in Section 5.2. We model the core operations (dropping columns, grouping and sorting) in Section 6.1 and then refine the model to include filtering based on values Section 6.3. We omit paging, as it does not affect the shape of data, but modelling it would require richer host language.

6.1 Pivot type provider

A type provider is a function that takes static parameters, such as schema of the input data set, and returns a class name C together with a mapping that defines the body of the class, together with definitions of associated classes L that may be used by the members of the class C. In our case, the schema F is a mapping from field names to field types:

$$\mathsf{pivot}(F) = C, \{C \mapsto (\text{type } C(x : \mathsf{Query}) = \ldots, L)\} \quad \text{where } F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$$

The class C provided by the pivot tpe provider has a constructor taking Query, which represents the, possibly already partly transformed, input data set. It generates members that allow the user to refine the query and access the data. The type provider is defined using several helper functions discussed in the rest of this section.

Entry-point and data access. Figure 8 shows three of the functions defining the pivot type provider. The pivot function ❶ defines the entry-point type, which lets the user choose which operation to perform before specifying parameters of the operation. This is the type of olympics in the examples throughout this paper. The definition generates a new class C with members that wrap the input data in delayed classes generated by other parts of the type provider. The result of pivot is the class name C together with definition of the class and

$$\mathsf{drop}(F) = C, \{C \mapsto (l, L' \cup \bigcup L_f)\} \quad ❶$$

$l =$ type $C(x : \mathsf{Query}) =$
  member «drop f» : $C_f = C_f(\Pi_{\mathsf{dom}(F')}(x))$
  member then : $C' = C'(x)$ ❷

$\forall f \in \mathsf{dom}(F)$ where $C_f, L_f = \mathsf{drop}(F')$
  and $F' = \{f' \mapsto \tau' \in F, f' \neq f\}$
where $C', L' = \mathsf{pivot}(F)$

$$\mathsf{sort}(F) = C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f)\} \quad ❸$$

$l =$ type $C(x : \mathsf{Query}) =$
  member «by f descending» : $C_f = C_f(x)$
  member «by f ascending» : $C'_f = C'_f(x)$

$\forall f \in \{f \mid f \mapsto \mathsf{num} \in F\}$, where
  $C_f, L_f = \mathsf{sort\text{-}and}(F, \langle f \mapsto \mathsf{desc} \rangle)$
  $C'_f, L'_f = \mathsf{sort\text{-}and}(F, \langle f \mapsto \mathsf{asc} \rangle)$

$$\mathsf{sort\text{-}and}(F, \langle s_1, \ldots, s_n \rangle) = C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f \cup L')\} \quad ❹$$

$l =$ type $C(x : \mathsf{Query}) =$
  member «and f descending» : $C_f = C_f(x)$
  member «and f ascending» : $C'_f = C'_f(x)$
  member then : $C' = C'(\tau_{s_1,\ldots,s_n}(x))$ ❺

$\forall f \in \{f \mid f \mapsto \mathsf{num} \in F, \nexists i. s_i = f' \mapsto \omega \wedge f' = f\}$
  $C_f, L_f = \mathsf{sort\text{-}and}(F, \langle s_1, .., s_n, f \mapsto \mathsf{desc} \rangle)$
  $C'_f, L'_f = \mathsf{sort\text{-}and}(F, \langle s_1, .., s_n, f \mapsto \mathsf{asc} \rangle)$
where $C', L' = \mathsf{pivot}(F)$

Figure 9 Pivot type provider – dropping columns and sorting data

delayed generated types. The definition is a function that only needs to be evaluated when a program accesses a member of the class C, modelling the laziness of the type provider. In the implementation, we return the name C, but only need to compute the definition when the type checker needs to access the body.

The get-key ❷ and get-val ❸ functions provide members that can be used to choose two columns from the data set as keys and values and obtain the resulting data set as a value of type series$\langle \tau_1, \tau_2 \rangle$. For example, the following expression has a type series$\langle$string, num$\rangle$:

```
olympics.«get series».«with key Athlete».«and value Year»
```

The get-key function generates a class with one member for each field in the data set. The returned class C_f is generated by get-val and lets the user choose any of the remaining fields as the value. The key and value columns are then selected using $\Pi_{f_k,f}$ ❹. The series is then created with a data set containing only the key and value columns (we assume the order of columns is preserved). Creating a series does not statically enforce that the data set has the right structure, but the properties discussed in Section 6.2 show that series obtained from the pivot type provider is constructed correctly.

Dropping columns and sorting. Functions that provide types for the «drop columns» and «sort data» members are defined in Figure 9. The drop function ❶ builds a new type that lets the user drop any of the available columns. The resulting type C_f is recursively generated by drop so that multiple columns can be dropped before completing the transformation using the then operation ❷, whose return type is generated using the main pivot function. Note that columns removed from the schema F' as the types are generated match the columns removed from the data set at runtime using $\Pi_{\mathsf{dom}(F')}$.

Types for defining the sorting transformation are split between two functions; sort ❸ generates type for choosing the first sorting key and sort-and ❹ lets the user add more keys. The members are restricted to numerical columns (by checking $f \mapsto \mathsf{num} \in F$). The sort

$\mathsf{group}(F) = C, \{C \mapsto (l, \bigcup L_f)\}$ ❶
$l = \text{type } C(x : \mathsf{Query}) =$
  member «by f» : $C_f = C_f(x)$
$\forall f \in \mathsf{dom}(F)$ where
  $C_f, L_f = \mathsf{agg}(F, f, \{f \mapsto F(f)\}, \emptyset)$ ❷

$\mathsf{agg}(F, f, G, S) = C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f \cup \bigcup L''_f \cup L' \cup L'')\}$ ❸
$l = \text{type } C(x : \mathsf{Query}) =$
  member «sum f» : $C'_f = C'_f(x)$ when $F(f) = \mathsf{num}$ ❹
  member «concat f» : $C''_f = C''_f(x)$ when $F(f) = \mathsf{string}$ ❺
  member «count all» : $C' = C'(x)$ when $\mathsf{Count} \notin G$ ❻
  member «distinct f» : $C_f = C_f(x)$
  member then : $C'' = C''(\Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(x))$ where $\{\rho_1/f_1, \ldots, \rho_n/f_n\} = S$ ❼
$\forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S)$

where

$C_f, L_f = \mathsf{agg}(F, f, G \cup \{f \mapsto \mathsf{num}\}, S \cup \{\mathsf{dist}\ f/f\})$
$C'_f, L'_f = \mathsf{agg}(F, f, G \cup \{f \mapsto \mathsf{num}\}, S \cup \{\mathsf{sum}\ f/f\})$
$C''_f, L''_f = \mathsf{agg}(F, f, G \cup \{f \mapsto \mathsf{string}\}, S \cup \{\mathsf{conc}\ f/f\})$
$C', L' = \mathsf{agg}(F, f, G \cup \{\mathsf{Count} \mapsto \mathsf{int}\}, S \cup \mathsf{count}/\mathsf{Count})$
$C'', L'' = \mathsf{pivot}(G)$

Figure 10 Pivot type provider – grouping and aggregation

keys are kept as a vector. The sort operation creates a singleton vector; sort-and appends a new key to the end and the then member ❺ generates code that passes the collected sort keys to the τ operation of the relational algebra. When generating members for adding further sort keys, we exclude the columns that are used already (by checking that the column f does not match column name of any of the existing keys $s_i = f' \mapsto \omega$).

Grouping and aggregation. The final part of the pivot type provider is defined in Figure 11. The group function ❶ generates a class that lets the user select a column to use as the grouping key and agg is used to provide aggregates that can be calculated over grouped data. The agg function ❸ takes the schema of the input data set F, column f to be used as the group key, a schema of the data set that will be produced as the result G and a set of aggregation operations collected so far S. Initially ❷, the resulting schema contains only the column used as the key with its original type (which is always generated by Φ) and the set of aggregations to be calculated is empty.

The agg function is invoked recursively (similarly to drop and sort-and) to add further aggregation operations, or until the user selects the then member ❼, which applies the grouping using Φ and returns a class generated by the entry-point pivot function.

When calculating an aggregate over a specific column, the type provider reuses the column name from the input data set in the resulting data set. Consequently, the agg function offers aggregation operations only using columns that have not been already used. This somewhat limits the expressivity, but it simplifies the programming model. Furthermore, «sum f» ❹ is only provided for columns of type num and «concat f» ❺ is only provided for strings. Finally, the «count all» aggregation ❻ is not related to a specific field and is exposed once, adding a column Count to the schema of the resulting data set.

6.2 Properties of the pivot type provider

If we were using the relational algebra formalised in Section 5.1 to construct queries, we can obtain an invalid program, e.g. by attempting to select a column f using Π_f from a data set that does not contain the column. This is not an issue when using the pivot type provider, because the provided types allow the user to construct only correct data transformations.

To formalise this, we prove partial soundness of the Foo calculus (Theorem 1), which characterises the invalid programs that can be written using the Query-typed expressions and then prove safety of the pivot type provider (Theorem 7), which shows that such errors do not occur when using the provided types.

Foo calculus. The Foo calculus consists of the relational algebra and simple object calculus where objects can be constructed and their members accessed. It permits recursion as a member can invoke itself on a new object instance. To accommodate this, we formalise soundness using progress (Lemma 2) and preservation (Lemma 3).

The soundness is partial because the evaluation can get stuck when an operation of the relational algebra on a given data set is undefined.

▶ **Theorem 1** (Partial soundness). *For all L_0, e, e', if $L_0, \emptyset \vdash e : \tau, L_1$ and $e \rightsquigarrow^*_{L_1} e'$ then either e' is a value, or there exists e'' such that $e' \rightsquigarrow^*_{L_1} e''$, or e' has one of the following forms: $E[\Pi_{f_1,\ldots,f_n}(D)]$, $E[\sigma_\varphi(D)]$, $\tau_{f_1,\ldots,f_n}(D)]$ or $E[\Phi_{f,\rho_1/f_1,\ldots,\rho_m/f_m}(D)]$ for some E, D.*

Proof. Direct consequence of Lemma 2 and Lemma 3. ◀

▶ **Lemma 2** (Partial progress). *For all L_0, e such that $L_0, \emptyset \vdash e : \tau, L_1$ then either, e is a value, there exists e' such that $e \rightsquigarrow_{L_1} e'$ or e has one of the following forms: $E[\Pi_{f_1,\ldots,f_n}(D)]$, $E[\sigma_\varphi(D)]$, $\tau_{f_1,\ldots,f_n}(D)]$ or $E[\Phi_{f,\rho_1/f_1,\ldots,\rho_m/f_m}(D)]$ for some E and D.*

Proof. By induction over $\vdash$. For data, strings and numbers, the expression is always a value. For relational algebra operations, the expression can either be reduced or has one of the required forms. For (*member*) the expression can always be reduced. ◀

▶ **Lemma 3** (Type preservation). *For all L_0, e, e' such that $L_0, \emptyset \vdash e : \tau, L_1$ and $e \rightsquigarrow_{L_1} e'$ then $L_1, \emptyset \vdash e' : \tau, L_2$ for some L_2.*

Proof. By induction over $\rightsquigarrow_{L_1}$. Cases for relational algebra operations and for (*context*) are straightforward. The (*member*) case follows from a standard substitution lemma and the fact that type checking of member access also type checks the body of the member. ◀

Correctness of the pivot provider. The pivot type provider defined by pivot defines an entry-point class and a context L containing delayed classes. The type system cannot check type definitions in L upfront, but we prove that the body of all provided members is well-typed. Furthermore, type checking could fail if a delayed class was not discovered before it is needed in the (*new*) and (*member*) typing rules (Figure 7). We show that this cannot happen for the context constructed by the pivot function. To avoid operating over potentially infinite contexts, we first define an expansion operation $\downarrow_n L$ that evaluates the first n levels of the nested context L and flattens it.

▶ **Definition 4** (Expansion). Given a context L, we define n^{th} expansion of L, written $\downarrow_n L$:

- $\downarrow_n L = \downarrow_{n-1} L \cup \bigcup L_n$ assuming that $\downarrow_{n-1} L = \{C_0 \mapsto (l_0, L_0), \ldots, C_n \mapsto (l_n, L_n)\}$
- $\downarrow_0 L = L$

▶ **Theorem 5** (Correctness of lazy contexts). *Given $C, L = \mathsf{pivot}(F)$ then for any e if there exists i, τ such that $\downarrow_i L; \emptyset \vdash e : \tau; L'$ then also $L; \emptyset \vdash e : \tau; L''$.*

Proof. Assume there exists F, e, i such that $\downarrow_i L; \emptyset \vdash e : \tau; L'$ but not $L; \emptyset \vdash e : \tau; L''$. This is a contradiction as (*new*) and (*member*) typing rules expand L defined by pivot sufficiently to discover all types that may have been used in the type-checking of e using $\downarrow_i L$. ◀

▶ **Theorem 6** (Correctness of provided types). *For all F, n let $C_0, L_0 = \mathsf{pivot}(F)$ and assume that $C \in \mathsf{dom}(\downarrow_n L)$ where $\downarrow_n L(C) = (\mathsf{type}\ C(x : \tau) = ..\ \mathsf{member}\ N_i : \tau_i = e_i\ ..), L'$. It holds that for all i the body of N_i is well-typed, i.e. $L \cup L'; x : \tau \vdash e_i : \tau_i; L''$.*

Proof. By examination of the functions defining the type provider; the expressions e_i are well-typed and use only types defined in $L \cup L'$. ◀

Safety of provided transformations. The two properties discussed above ensure that the types provided by the pivot type provider can be used to type check expressions constructed by the users of the type provider in the expected way. An expression will not fail to type check because of an error in the provided types.

Now we can turn to the key theorem of the paper, which states that any expression constructed using (just) the provided types can be evaluated to a value of correct type. For simplicity, we only assume expressions that access a series using the «get series» member. However, this covers all data transformations that can be constructed using the type provider.

▶ **Theorem 7** (Safety of pivot type provider). *Given a schema $F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$, let $C, L = \mathsf{pivot}(F)$ then for any expression e that does not contain relational algebra operations or* Query*-typed values as sub-expression, if $L; x : C \vdash e : \mathsf{series}\langle\tau_1, \tau_2\rangle; L'$ then for all $D = \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,m}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,m}\rangle\}$ such that $\vdash v_{i,j} : \tau_i$ it holds that $e[x \leftarrow C(D)] \rightsquigarrow^*_{L'} \mathsf{series}\langle\tau_k, \tau_v\rangle(\{f_k \mapsto k_1, \ldots, k_r, f_v \mapsto v_1, \ldots, v_r\})$ such that for all j $\vdash k_j : \tau_k$ and $\vdash v_j : \tau_v$.*

Proof. Define a mapping $\mathsf{fields}(C)$ that returns the fields expected in the data set passed to a class C provided by the pivot type provider. Let $\mathsf{fields}(C) = F$ for C provided using:

$\mathsf{pivot}(F) = C, L$	$\mathsf{get\text{-}key}(F) = C, L$	$\mathsf{sort}(F) = C, L$
$\mathsf{drop}(F) = C, L$	$\mathsf{get\text{-}val}(F, f_k) = C, L$	$\mathsf{sort\text{-}and}(F, \langle s_1, \ldots, s_n\rangle) = C, L$
$\mathsf{group}(F) = C, L$	$\mathsf{agg}(F, f, G, S) = C, L$	

By induction over $\rightsquigarrow_{L'}$, show that when $C(v).N_i$ is reduced using (*member*) then v is a value $\{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,m}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,m}\rangle\}$ s.t. $\mathsf{fields}(C) = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$ and $\vdash v_{i,j} : \tau_i$. Thus the class provided by get-val is constructed with a data set containing the required columns of corresponding types. ◀

6.3 Adding the filtering operation

Add D representing input data (in reality, we do not need all of it; just a function)

zz

$\mathsf{pivot}(F, D) = C, \{C \mapsto (l, L_1 \cup L_2 \cup \ldots\}$ ❶

$l = \text{type } C(x : \mathsf{Query}) =$
  member «drop columns» : $C_1 = C_1(x)$ where $C_1, L_1 = \mathsf{drop}(F, D)$
  member «filter data» : $C_2 = C_2(x)$ where $C_2, L_2 = \mathsf{filter}(F, D)$
  $(\ldots)$

$\mathsf{drop}(F, D) = C, \{C \mapsto (l, L' \cup \bigcup L_f)\}$

$l = \text{type } C(x : \mathsf{Query}) =$ $\forall f \in \mathsf{dom}(F)$
  member «drop f» : $C_f =$ where $F' = \{f' \mapsto \tau' \in F, f' \neq f\}$
  $C_f(\Pi_{\mathsf{dom}(F')}(x))$ and $C_f, L_f = \mathsf{drop}(F', \Pi_{\mathsf{dom}(F')}(D))$ ❷
  member then : $C' = C'(x)$ where $C', L' = \mathsf{pivot}(F, D)$

$\mathsf{filter}(F, D) = C, \{C \mapsto (l, L' \cup \bigcup L_f)\}$

$l = \text{type } C(x : \mathsf{Query}) =$ $\forall f \in \mathsf{dom}(F)$
  member «f is» : $C_f = C_f(x)$ where $C_f, L_f = \mathsf{filter\text{-}val}(F, f, D)$ ❸
  member then : $C' = C'(x)$ where $C', L' = \mathsf{pivot}(F, D)$

$\mathsf{filter\text{-}val}(F, f, D) = C, \{C \mapsto (l, \cup \bigcup L_v)\}$ where $D = \{f \mapsto \langle v_1, \ldots, v_n \rangle, \ldots\}$ ❹

$l = \text{type } C(x : \mathsf{Query}) =$ $\forall v \in \{v_1, \ldots, v_n\}$
  member «v» : $C_v =$ where $C_v, L_v = \mathsf{filter}(F, \sigma_{\varphi_v}(D))$
  $C_v(\sigma_{\varphi_v}(x))$ and $\varphi_v(r) = r(f) = v$ ❺

Figure 11 Pivot type provider – grouping and aggregation

7 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