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 into 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 also compose rich and correct queries using just member access ("dot"). This way, we recreate functionality that usually requires complex type systems (row polymorphism, type state and dependent typing) in an extremely simple object-based language.

We formalize our approach using an object-based calculus and prove that programs constructed using the provided types represent valid data transformations. We discuss a case study developed using the language, together with additional editor tooling that bridges some of the gaps between programming and spreadsheets. We believe that this work 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.DRAFT.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 [10] 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 make transparent factual claims.

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 [13, 21]. One solution is to explore data programmatically, as programs can be run repeatedly and their 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 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:

```
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 the pivot type provider based on the knowledge of the data source and transformations applied so far – 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 new area of the programming language design space. We support our design 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” [32] 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 our mechanism can be used for embedding other advanced typing schemes into any Java-like language.
- We formalize the language (Section 5) and the pivot type provider (Section 6) and show that queries for exploring data constructed using the type provider are correct (Section 6.2). Our formalization also covers the laziness of type providers, which is an important aspect not covered in the existing literature.
- We implement the language (`github.com/the-gamma`), make it available as a JavaScript component (`thegamma.net`) that can be used to build transparent data-driven visualizations and discuss a case study visualizing facts about Olympic medalists (Section 7).

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 [30]. 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 [23] 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 indicators exist only as strings in compiled code, but the type provider provides a type-safe access to known indicators, 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 [22] 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 [22], 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 converts 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 host 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 to lazily generate types with members that let users compose type-safe queries over the data source.

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). The implementation relies on the fact that type providers are integrated into the type system and types can be provided lazily. 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 the popular Python library pandas [17] 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 the four points to inform our alternative design as discussed in the rest of this section.

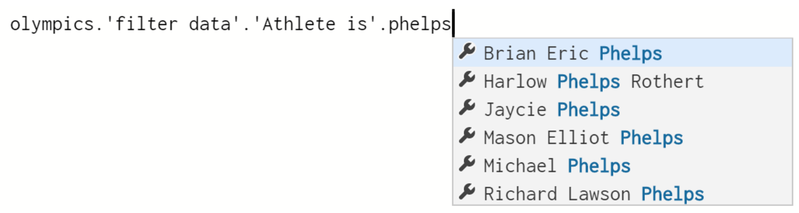
- The filtering operation is written using indexing [. . .] while all other operations are written using member invocation with (optionally named) parameters. In the first case, we write an expression `olympics["Games"] == "Rio (2016)"` returning a vector of Booleans while in the other, we specify a column name using `by = "Gold"`. In other languages, 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 [18] 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), column names are specified as strings[3]. 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 statically in the pivot type provider (Section 4).

[1] For an anecdotal evidence, see for example: `stackoverflow.com/questions/25801246`

[2] A detailed evaluation is out of the scope of this paper, but the reader can compare the Python example with F# code using Deedle (`fslab.org/Deedle`), Haskell Frames library (`acowley.github.io/Frames`) and similar C# project (`extremeoptimization.com/Documentation/Data_Frame`)

[3] This is the case for Deedle and the aforementioned C# library. Haskell Frames [8] tracks column names statically, arguably at the cost of higher code complexity when compared with Python.



```
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-expression, 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 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. Quoted names appear as '...' in code, but we typeset them using «...» for readability. 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 further in Section 6.3).

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.

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

```
«drop columns»
  → «drop Athlete»
  → «drop Discipline»
  → «drop Year»

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

«group data»
  → «by Athlete»
      → «average Year»
      → «sum Year»
  → «by Year»
      → «distinct Athlete»
      → «concat Athlete»
      → «distinct Discipline»
      → «concat Discipline»
```

Figure 2 Subset of members provided by the pivot type provider

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[4]. 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 restricted to choosing one of the provided members. As illustrated by 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 systems often provide auto-completion for members of objects. This combination works extremely well in practice; 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 [30] 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 [22] and is used merely as a lookup key.

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.

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

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 entire 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 Figure 1 illustrates, the pivot type provider sometimes blurs the distinction between values and types. In the example in Section 1, "Rio (2016)" is a string value in Python, but a statically-typed member «Rio (2016)» when using the pivot type provider. This is a recurring theme in type provider development[5].

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 6.3), but all other operations (Section 6.1) require only meta-data (names and types of columns). Following the example of type providers for structured data formats [22], 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 the "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 sort the grouped data using multiple sorting keys ❷ – first by the number of gold medals and then silver medals (within a group with the same number of gold medals).

The "then" pattern. Nesting is an essential programming construct and it may be desirable to support it directly in the language, but the "then" pattern lets us express nesting without language support. In both of the cases above, the nested structure is specified by selecting one or more members and then ending the nested structure using the then member.

[5] The Individuals property in the Freebase type provider [30] imports values into types in a similar way.

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. We also use the simple structure of the "then" pattern to automatically generate interactive user interfaces for specifying aggregation and sorting parameters (Section 7).

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 type providers for F#, this could be done using methods with static parameters [29] 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 most 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 support. 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. For simplicity, we consider another operation – removing column from the data set. For example, given the Olympic medals data set, we can drop Games and Year columns as follows:

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

Operations that change the type of rows in the data set can be captured using row types [31]. In addition, we need to annotate type with a form of typestate [28] 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:

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

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

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

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

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

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 typestate (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 inference (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 the pivot type provider to provide types with appropriate members. This is hidden from the host language, which only sees class types. 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 lookup of provided class definitions L. Checking member access is then just a matter of finding the corresponding class definition and finding the member type:

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

The rule, adapted from [22], 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.

Using type providers and nominal type system 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:

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	$=$	$\Pi_{f_1,\ldots,f_n}(e)$	Projection – select specified column names
	$\mid$	$\sigma_\varphi(e)$	Selection – filter rows by given predicate
	$\mid$	$\tau_{f_1 \mapsto \omega_1,\ldots,f_n \mapsto \omega_n}(e)$	Sorting – sort by specified columns
	$\mid$	$\Phi_{f,\rho_1/f_1,\ldots,\rho_n/f_n}(e)$	Grouping – group by and calculate aggregates
ω	$=$	$\mathsf{desc} \mid \mathsf{asc}$	Sort order – descending or ascending
ρ	$=$	count	Count number of rows in the group
	$\mid$	$\mathsf{sum}\ f$	Sum numerical values of the column f
	$\mid$	$\mathsf{dist}\ f$	Count number of distinct values of the column f
	$\mid$	$\mathsf{conc}\ f$	Concatenate string values of the column f

Figure 4 Relational algebra with values, sorting and aggregation

▶ **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 typestate in a simple nominally typed language. In a similar way, type providers have been used to encode session types [2], 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 lookups into 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, followed by 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 [7, 20]. 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 column names 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.

The semantics of the algebra 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

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

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

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

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

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

$\{1\ldots n\}$. 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 indices 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 [22]. 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.

The Foo calculus includes two special types. Query is a type of data and queries constructed using the relational algebra. The type series$\langle\tau_1,\tau_2\rangle$ models a type-safe data series mapping keys of type τ_1 to values of type τ_2 that can be used, for example, as input for a charting library. A series is a 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.

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

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

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

Figure 6 Syntax and remaining reduction rules of the Foo calculus

Reduction rules. The reduction relation $\rightsquigarrow_L$ is parameterized 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 $(\mathit{context})$ 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. The types are provided lazily as the type checker explores parts of the type space used by the program [30]. 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 members 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 [22, 15], but it is necessary for the pivot type provider. The typing rules given are written using the following judgement:

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

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 resulting context obtained by type checking contains all definitions that may be needed when running the program and is passed to the reduction operation $\rightsquigarrow_L$.

The structure of class definitions L is a function mapping a class name C to a pair consisting of the definition and a function that provides definitions of delayed classes:

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

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

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

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

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

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

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

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

Figure 7 Type-checking of Foo expressions with lazy context

The class C may use classes defined in L, 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 [1]. 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 Query input and produce Query results. An untyped Query value can be converted into a series (*series*) of any type, akin to the boundary between static and dynamic typing in gradually typed languages [27]. When provided by the pivot type provider, the operation produces series with values of correct types.

6 Formalising the pivot type provider

A type provider is an executable component called by the compiler and the editor to provide information about types on demand. In our formalization, we follow the style of Petricek et al. [22], 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 refine the model to include filtering Section 6.3. For simplicity we omit paging, which does not affect the shape of data.

$$
\begin{array}{ll}
\mathsf{pivot}(F) = C, \{C \mapsto (l, L_1 \cup \ldots \cup L_4)\} \quad ❶ & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \\
\qquad \mathsf{member}\ \text{«drop columns»} : C_1 = C_1(x) & \text{where } C_1, L_1 = \mathsf{drop}(F) \\
\qquad \mathsf{member}\ \text{«sort data»} : C_2 = C_2(x) & \text{where } C_2, L_2 = \mathsf{sort}(F) \\
\qquad \mathsf{member}\ \text{«group data»} : C_3 = C_3(x) & \text{where } C_3, L_3 = \mathsf{group}(F) \\
\qquad \mathsf{member}\ \text{«get series»} : C_4 = C_4(x) & \text{where } C_4, L_4 = \mathsf{get\text{-}key}(F) \\
 & \\
\mathsf{get\text{-}key}(F) = C, \{C \mapsto (l, \bigcup L_f)\} \quad ❷ & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \text{ where} \\
\qquad \mathsf{member}\ \text{«with key } f\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{get\text{-}val}(F, f) \\
 & \\
\mathsf{get\text{-}val}(F, f_k) = C, \{C \mapsto (l, \{\})\} \quad ❸ & \\
\quad l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \mathsf{dom}(F) \setminus \{f_k\} \text{ where} \\
\qquad \mathsf{member}\ \text{«and value } f\text{»} : \mathsf{series}\langle \tau_k, \tau_v \rangle = & \quad \tau_k = F(f_k),\ \tau_v = F(f) \\
\qquad\quad \mathsf{series}\langle \tau_k, \tau_v \rangle (\Pi_{f_k, f}(x)) \quad ❹ &
\end{array}
$$

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

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 and definitions of delayed 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:

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

The class C provided by the pivot type 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 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 together with a function that computes the definition of the class when the type checker needs to inspect 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 $\mathsf{series}\langle \tau_1, \tau_2 \rangle$. For example, the following expression has a type $\mathsf{series}\langle \mathsf{string}, \mathsf{num} \rangle$:

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

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

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

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

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

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

$$\begin{array}{ll}
l = \mathsf{type}\ C(x : \mathsf{Query}) = & \forall f \in \{f \mid F(f) = \mathsf{num}, \nexists i. s_i = f' \mapsto \omega \wedge f' = f\} \\
\quad \mathsf{member}\ \text{«}\mathsf{and}\ f\ \mathsf{descending}\text{»} : C_f = C_f(x) & \quad C_f, L_f = \mathsf{sort\text{-}and}(F, \langle s_1, .., s_n, f \mapsto \mathsf{desc}\rangle) \\
\quad \mathsf{member}\ \text{«}\mathsf{and}\ f\ \mathsf{ascending}\text{»} : C'_f = C'_f(x) & \quad C'_f, L'_f = \mathsf{sort\text{-}and}(F, \langle s_1, .., s_n, f \mapsto \mathsf{asc}\rangle) \\
\quad \mathsf{member}\ \mathsf{then} : C' = C'(\tau_{s_1, \ldots, s_n}(x)) \quad ❺ & \text{where } C', L' = \mathsf{pivot}(F)
\end{array}$$

Figure 9 Pivot type provider – dropping columns and sorting data

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' 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 $F(f) = \mathsf{num}$). The sort 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 $\nexists i.\, s_i = f' \mapsto \omega$).

Grouping and aggregation. The final part of the pivot type provider is defined in Figure 10. 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

```
$\mathsf{group}(F) = C, \{C \mapsto (l, \bigcup L_f)\}$ ❶
 $l = \mathsf{type}\ C(x : \mathsf{Query}) =$                      $\forall f \in \mathsf{dom}(F)$ where
       $\mathsf{member}$ «by $f$» $: C_f = C_f(x)$                      $C_f, L_f = \mathsf{agg}(F, f, \{f \mapsto F(f)\}, \emptyset)$ 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
 $l = \mathsf{type}\ C(x : \mathsf{Query}) =$                      $\forall f \in \mathsf{dom}(F) \setminus \mathsf{dom}(S)$
       $\mathsf{member}$ «sum $f$» $: C'_f = C'_f(x)$                     when $F(f) = \mathsf{num}$ ❹
       $\mathsf{member}$ «concat $f$» $: C''_f = C''_f(x)$                when $F(f) = \mathsf{string}$ ❺
       $\mathsf{member}$ «count all» $: C' = C'(x)$                    when $\mathsf{Count} \notin G$ ❻
       $\mathsf{member}$ «distinct $f$» $: C_f = C_f(x)$
       $\mathsf{member}$ 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
    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

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 implicitly added 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 formalized in Section 5.1 to construct queries, we can write 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 formalize this, we prove partial soundness of the Foo calculus (Theorem 1), which characterizes 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 formalize 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*) typing guarantees reduction is possible. ◀

▶ **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. Our type system does not check type definitions in L upfront (although this is possible in dependently-typed languages [6]), but we prove that the body of all provided members is well-typed.

Type checking can also 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 $\downarrow_n L$ such that $\downarrow_{n+1} L = \downarrow_n L \cup \bigcup L_n$ where $\downarrow_n L = \{C_0 \mapsto (l_0, L_0), \ldots, C_n \mapsto (l_n, L_n)\}$ and $\downarrow_0 L = L$.

▶ **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:

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

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

The example given in Section 1 obtained top 8 athletes based on the number of gold medals from Rio 2016. It used two operations that were omitted in the formalization in Section 6.1. We omitted paging to keep the host language simple, but we also omitted filtering, which lets us write «filter data».«Games is».«Rio (2016)». This operation is worth further discussion. To support it, the type provider needs not only the schema of the data set, but also sample data set that is used to offer the available values such as «Rio (2016)».

In the revised formalization, the pivot function which models the type provider takes the schema F together with sample data D and provides the type with class context:

$$\begin{array}{l} \mathsf{pivot}(F, D) = C, L \quad \text{where} \\ \qquad F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\} \\ \qquad D = \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,r} \rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,r} \rangle\} \end{array}$$

In prior work [22], the input value is not available when writing the code and so the schema is inferred from a representative sample. In exploratory data analysis, the data set is often available at the time of writing the code and so D can be the actual data set.

The Figure 11 shows a revised version of the pivot function ❶ together with one of the operations discussed before and the newly added filter function. As members members performing data transformations are generated, the provider applies the same transformation on the sample data. For example, the revised drop function ❷ takes the sample data set D; when calling drop recursively to generate nested class after dropping a column ❸, it removes

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

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

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

$\text{filter-val}(F, f, D) = C, \{C \mapsto (l, \cup \bigcup L_v)\}$      where $D = \{f \mapsto \langle v_1, \ldots, v_n \rangle, \ldots\}$ ❺
  $l = \text{type } C(x : \text{Query}) =$      $\forall v \in \{v_1, \ldots, v_n\}$
      $\text{member « } v \text{ »} : C_v =$      where $C_v, L_v = \text{filter}(F, \sigma_{\varphi_v}(D))$
        $C_v(\sigma_{\varphi_v}(x))$      and $\varphi_v(r) = r(f) = v$ ❻
```

Figure 11 Pivot type provider – grouping and aggregation

the column from the schema (as before), but it also removes the column from the sample dataset. This means that as nested types are provided, the sample data used is always representative of data what will be passed to the class at runtime.

After choosing the «filter data» member, the class provided by filter lets the user select one of the columns ❹ based on the schema; filter-val then generates a class with members based on the available values for the specified column in the data D ❺. The predicate that filters data based on the value ❻ is used both in the runtime code and when restricting the sample data set using $\sigma_{\varphi_v}(D)$ in the type provider when recursively calling filter.

The fact that we transform the sample data when providing types is important for two reasons. It makes it possible to apply filtering after aggregation (which changes the format of data) and it means that more appropriate values are provided for faceted data. For example, Figure 12 shows some of the provided members when filtering data by medal, team and individual athlete. Once we refine the team using «Team is».Mongolia and attempt to filter by athlete using «Athlete is», the type provider offers only the names of Mongolian athletes.

7 Case study: Visualizing Olympic medalists

We used The Gamma script with the pivot type provider to build an interactive web site (`rio2016.thegamma.net`) that visualizes a number of facts about Olympic medalists using the data set discussed in Appendix A and used throughout this paper. The web site lets the readers view and modify the source code and we also developed a number of tools that

```
olympics.«filter data».«Medal is».Gold.«Team is»
    → «Czech Republic».«Athlete is»        → Mongolia.«Athlete is»
        → «Barbora Spotakova»                  → «Badar-Uugan Enkhbat»
        → «David Kostelecky»                   → «Tuvshinbayar Naidan»
        → «David Svoboda»
```

Figure 12 Subset of members provided by the filtering operation

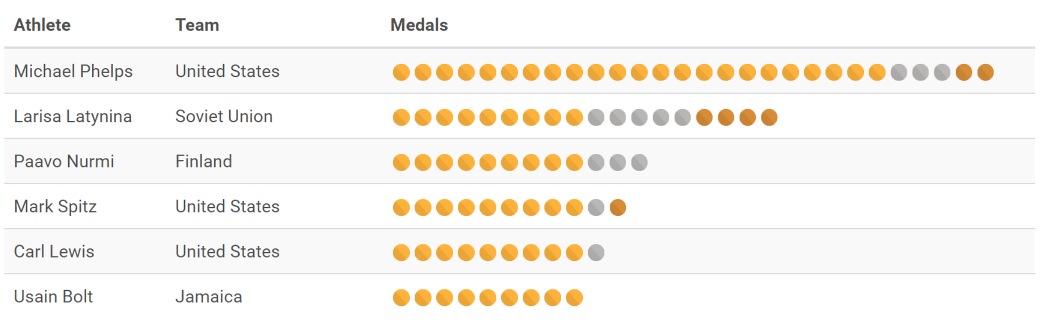
make working with the source code easier, going beyond the basic auto-completion tooling to enable dot-driven development as discussed in Section 3.3. In this section, we review our experience and outline some of the additional tools (available at `github.com/the-gamma`).

Building tables and charts. As part of the case study, we implement functions for building basic visualizations (table, column chart, pie chart and timeline) and we extended the host language with more advanced features that can be used to customize the displays. Building rich visualizations with the simplicity of the pivot type provider is an interesting future work. Figure 13 shows a sample table, listing top athletes over the entire history of Olympic games.

The data transformation used to construct the table include the operations discussed in this paper together with paging functionality and «get the data» which returns the entire data set of type Query, as opposed to extracting a series with keys and values:

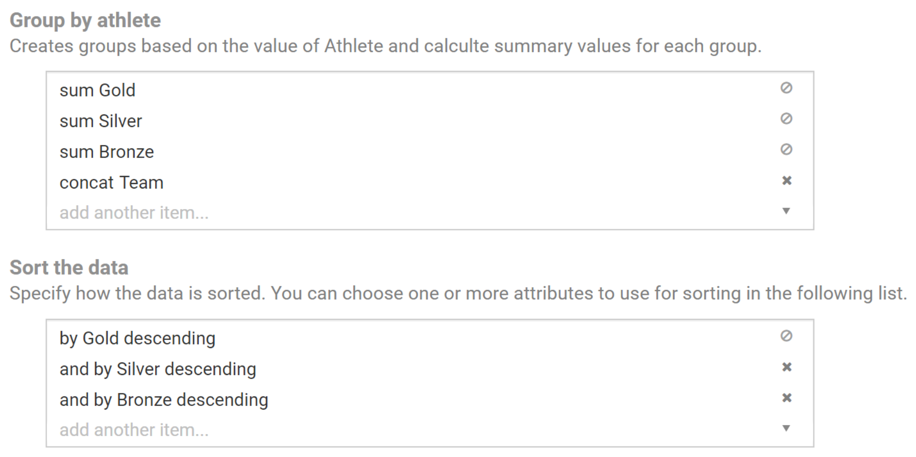
```
let data = olympics
  .«group data».«by Athlete»
    .«sum Gold».«sum Silver».«sum Bronze».«concat Team».then
  .«sort data».«by Gold descending»
    .«and by Silver descending».«and by Bronze descending».then
  .paging.take(10).«get the data»

table.create(data)
```

The table.create operation on the last line generates a table based on the columns available in the data set. We omit the additional customization which specifies that medals should be rendered as images. For most visualizations we built, the pivot type provider was expressive enough to capture the core logic of the operation, but further joining of data was sometimes needed. Possible extensions that would allow capturing those are discussed in Section 8.1.



Athlete	Team	Medals
Michael Phelps	United States	🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡🟡⚪⚪⚪🟤🟤
Larisa Latynina	Soviet Union	🟡🟡🟡🟡🟡🟡🟡🟡🟡⚪⚪⚪⚪⚪🟤🟤🟤🟤
Paavo Nurmi	Finland	🟡🟡🟡🟡🟡🟡🟡🟡🟡⚪⚪⚪
Mark Spitz	United States	🟡🟡🟡🟡🟡🟡🟡🟡🟡⚪🟤
Carl Lewis	United States	🟡🟡🟡🟡🟡🟡🟡🟡🟡⚪
Usain Bolt	Jamaica	🟡🟡🟡🟡🟡🟡🟡🟡🟡

Figure 13 Athletes by the number of medals over the entire history of Olympic games


Group by athlete
Creates groups based on the value of Athlete and calculte summary values for each group.

sum Gold
sum Silver
sum Bronze
concat Team
add another item...

Sort the data
Specify how the data is sorted. You can choose one or more attributes to use for sorting in the following list.

by Gold descending
and by Silver descending
and by Bronze descending
add another item...

Figure 14 User interface with automatically provided grouping and sorting options

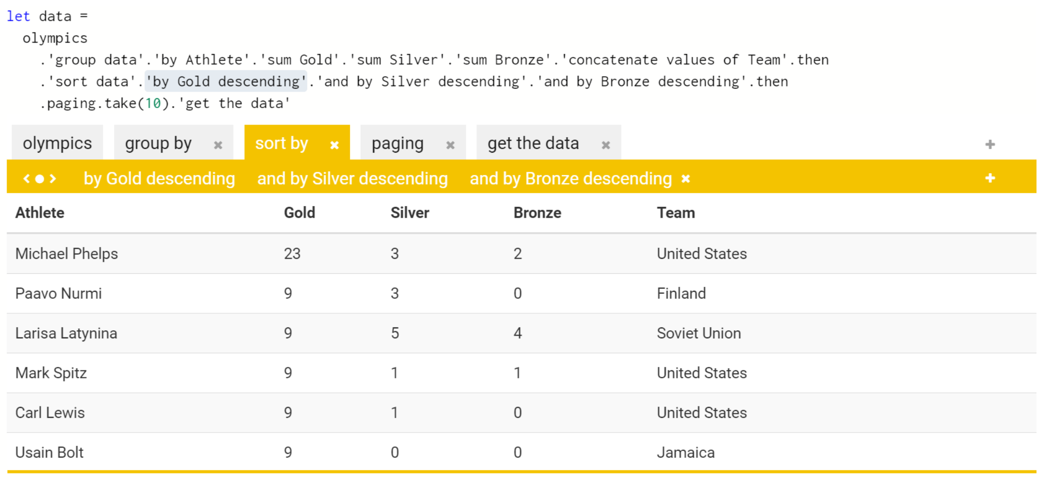
Generating interactive user interfaces. Although the pivot type provider simplifies code needed for data exploration, not everyone will be able to write or modify source code. The simplicity of the host language makes it possible to automatically generate user interface that allows changing of some of the parameters of the program. Figure 14 shows an example for the above code snippet that we implemented as part of the visualization.

The user interface lets the user choose aggregations to be calculate over a group and select columns used for sorting. It is generated automatically by looking for specific pattern in the chain of member accesses – we annotate members with annotations denoting whether a member is start of a list, list item or an end of a list. The editor then looks for parts of the chain of the form «list start».«list item 1».«list item 2».«list end» and generates a component that lets the user remove or add list items. An item cannot be removed if the operation would break the code (e.g. when it adds a member that is needed later) and items to be added are chosen using available members (as in the standard auto-complete). The headers shown in Figure 14 are provided as additional annotations attached to «list start».

Spreadsheet-inspired live editor. The third editor extension that we developed for the pivot type provider aims to bridge the gap between code and user interfaces. This is done through a direct manipulation editor [24] inspired by spreadsheet applications. When exploring data in a spreadsheet, the user can always see the data they work with and the results of an action will be immediately visible. When exploring data using the pivot type provider, the intermediate results can be calculated immediately using the sample data set provided when instantiating the refined version of the type provider with filtering support (Section 6.3).

The Figure 15 shows the sample expression (discussed above) in the live editor[6]. Note that the selected part of code is the «by Gold descending» identifier and so the preview shows results as computed at that point of the query evaluation. Athletes with largest number of gold medals appear first, but silver or bronze medals are not yet used as secondary sorting keys. As the user moves through the code, or writes the code, the live preview is updated accordingly.

[6] The live editor can be tested live as part of the documentation for the JavaScript package at `thegamma.net`



```
let data =
  olympics
    .'group data'.'by Athlete'.'sum Gold'.'sum Silver'.'sum Bronze'.'concatenate values of Team'.then
    .'sort data'.'by Gold descending'.'and by Silver descending'.'and by Bronze descending'.then
    .paging.take(10).'get the data'
```

Athlete	Gold	Silver	Bronze	Team
Michael Phelps	23	3	2	United States
Paavo Nurmi	9	3	0	Finland
Larisa Latynina	9	5	4	Soviet Union
Mark Spitz	9	1	1	United States
Carl Lewis	9	1	0	United States
Usain Bolt	9	0	0	Jamaica

Figure 15 Spreadsheet-inspired live editor for the pivot type provider

Finally, the editor also makes it possible to modify the code through the user interface. The "x" buttons can be used to remove sort keys or transformations and "+" buttons (on the right) can be used to add more transformations or parameters.

Unlike the user interface for modifying lists, the live editor works specifically with the pivot type provider. However, it still relies on the simple structure provided by the fact that entire transformation can be written as a single chain of member accesses. In particular, we identify individual transformations («group by», «sort by», etc.) and generate different user interface for specifying parameters of each transformation. For sorting, as shown in Figure 15, the user can add or remove sort keys. For grouping or paging, the user interface lets the user choose the grouping key and the number of elements to take, respectively.

8 Discussion and related work

Although the technical focus of this paper is on the programming language theory behind the pivot type provider (Section 6), the paper also outlines interesting human-computer interaction aspects (Section 7). We discuss further related directions in this section before concluding.

8.1 Related and further work

Type providers. Type providers have been pioneered in F# [30]. They can be seen as a form of dependent typing [6]; we take the opposite perspective and use type providers as a mechanism for implementing other type system features. Our focus on using type providers for describing computations is different from other type provider work [22, 15, 23], which focuses on mapping of external data into types.

Fancy types. The pivot type provider makes data exploration safer as it does not allow construction of invalid queries. Alternative approach would be to use fancy types, such as those available in Haskell [8, 32]. The approach sketched in Section 4.1 used row types and typestate or phantom types [31, 28, 14]. The idea of using type providers to encode fancy types has also been explored for session types [11, 2] and it would be interesting to see whether our approach can be applied in other areas such as web development [5].

Human-computer interaction. We discussed how the pivot type provider simplifies the programming model (Section 3), but it would be interesting to explore this aspect empirically through the perspective of HCI. The live editor shown in Section 7 offers a form of direct manipulation [24, 25, 26]. Unlike spreadsheets, we construct a transformation rather than actually transforming data, which makes it more related to systems for query construction [16, 4]. Our approach is somewhat different in that we see code as equally important to the direct manipulation interface.

Relational algebra. Our operational semantics used to model data transformations (Section 5) was based on relational algebra [7, 20], although our focus was on aggregation, which has been added to the core algebra in a number of different ways [19, 12, 3, 9]. The pivot type provider does not provide operations for joining data sets, which is an interesting problem for further work as it requires extensions to the type provider mechanism – the join operation is parameterized by two data sets that are being combined.

8.2 Conclusions

In this paper, we presented a simple programming language for data exploration. The language addresses two problems with the current tooling for data science. One one hand, spreadsheets are easy to use, but are error-prone and do not lead to reproducible scripts that could be modified or checked for correctness. On the other hand, even simple data exploration libraries require the user to understand non-trivial programming concepts and offer only little help when writing data exploration code.

We reduce the number of concepts in the language by making member access ("dot") the primary programming mechanism and we implement type provider for data exploration, which offers available transformations and their parameters as members of a provided type. This leads to a simple language that can be well supported by standard tooling such as auto-completion. We also explore other possibilities for tooling enabled by this model ranging from simple interactive user interfaces to direct manipulation tools.

The pivot type provider offers a safe and easy to use layer over an underlying relational algebra that we use to model data transformations. As a key technical contribution of this paper, we formalize the type provider and prove that queries constructed using the types it provides are correct. Achieving this property by other means would require a language with complex type system features such as typestate and row types.

We believe that the simple programming model for data exploration presented in this paper can contribute to democratization of data exploration – you should not need to be an experienced programmer to build a transparent visualization using facts that matter to you!

References

1 Martin Abadi and Luca Cardelli. *A theory of objects*. Springer Science & Business, 2012.

2 Fahd Abdeljallal. Session types with Fahd Abdeljallal. F#unctional Londoners meetup group, 2016. URL: `https://skillsmatter.com/meetups/8459`.

3 Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

4 Eirik Bakke and David R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of International Conference on Management of Data*, SIGMOD '16, pages 1377–1392. ACM, 2016. `doi:10.1145/2882903.2915210`.

5 Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. *SIGPLAN Not.*, 45(6):122–133, June 2010. `doi:10.1145/1809028.1806612`.

6 David Raymond Christiansen. Dependent type providers. In *Proceedings of Workshop on Generic Programming*, WGP '13, pages 25–34. ACM, 2013. `doi:10.1145/2502488.2502495`.

7 E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. `doi:10.1145/362384.362685`.

8 Anthony Cowley. Frames: Data frames for tabular data. Available on GitHub, 2017. URL: `https://github.com/acowley/Frames`.

9 Richard Cyganiak. A relational algebra for sparql. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, page 35, 2005.

10 Oxford Dictionaries. Word of the year 2016 is... Oxford University Press, 2016. URL: `https://en.oxforddictionaries.com/word-of-the-year/word-of-the-year-2016`.

11 Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *European Symposium on Programming*, pages 74–90. Springer, 1999.

12 Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of International Conference on Data Engineering*, ICDE '96, pages 152–159. IEEE Computer Society, 1996.

13 Paul Krugman. The Excel depression. *New York Times*, 18, 2013.

14 Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, December 1999. `doi:10.1145/331963.331977`.

15 Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with LITEQ. In *International Semantic Web Conference*, pages 212–227. Springer, 2014.

16 Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Proceedings of International Conference on Data Engineering*, ICDE '09, pages 417–428. IEEE Computer Society, 2009. `doi:10.1109/ICDE.2009.34`.

17 Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* O'Reilly Media, Inc., 2012.

18 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .net framework. In *Proceedings of the International Conference on Management of Data*, pages 706–706. ACM, 2006.

19 Z. Meral Özsoyoglu and Gultekin Özsoyoglu. An extension of relational algebra for summary tables. In *Proceedings of International Workshop on Statistical Database Management*, SSDBM'83, pages 202–211. Lawrence Berkeley Laboratory, 1983.

20 M. Tamer Ozsu. *Principles of Distributed Database Systems.* Prentice Hall Press, 3rd edition, 2007.

21 Raymond R Panko. What we know about spreadsheet errors. *Journal of Organizational and End User Computing (JOEUC)*, 10(2):15–21, 1998.

22 Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: Making structured data first-class citizens in F#. In *Proceedings of Conference on Programming Language Design and Implementation*, PLDI '16, pages 477–490. ACM, 2016. `doi:10.1145/2908080.2908115`.

23 Tomas Petricek, Don Syme, and Zach Bray. In the age of web: Typed functional-first programming revisited. In *Proceedings ML Family/OCaml Users and Developers workshops*, ML '15. ACM, 2015.

24 Ben Shneiderman. The future of interactive systems and the emergence of direct manipulation. In *Proceedings of the NYU Symposium on User Interfaces on Human Factors and Interactive Computer Systems*, pages 1–28. Ablex Publishing Corp., 1984.

25 Ben Shneiderman. Direct manipulation for comprehensible, predictable and controllable user interfaces. In *Proceedings of International Conference on Intelligent User Interfaces*, pages 33–39. ACM, 1997.

26 Ben Shneiderman, Christopher Williamson, and Christopher Ahlberg. Dynamic queries: database searching by direct manipulation. In *Proceedings of Conference on Human Factors in Computing Systems*, pages 669–670. ACM, 1992.

27 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

28 Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Eng.*, (1):157–171, 1986.

29 Don Syme. F# 4.0 speclet - extending the F# type provider mechanism to allow methods to have static parameters. F# Language Design Proposal, 2016. URL: `https://github.com/fsharp/fslang-design/blob/master/FSharp-4.0/StaticMethodArgumentsDesignAndSpec.md`.

30 Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of Workshop on Data Driven Functional Programming*, DDFP '13, pages 1–4. ACM, 2013. `doi:10.1145/2429376.2429378`.

31 Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93(1):1–15, July 1991. `doi:10.1016/0890-5401(91)90050-C`.

32 Stephanie Weirich. Depending on types. *SIGPLAN Not.*, 49(9):241–241, August 2014. `doi:10.1145/2692915.2631168`.

A Sample of the Olympic medals data set

The data set used as an example in the case study discussed in Section 7 as well as in the examples discussed throughout the paper is a single CSV file listing the entire history of Olympic medals awarded since 1896. The data set can be found at `https://github.com/the-gamma/workyard` together with scripts used to obtain it. The following is a representative example listing the first 5 rows:

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

The column names are the same as the column names used to generate the `olympics` value using the pivot type provider. The script to generate the file de-normalizes the `Medal` column and adds `Gold`, `Silver` and `Bronze` columns which are numerical and can thus be easily summed. When loading the data, we also transform country codes such as `GRE` to full country names.