

Charles University
Faculty of Mathematics and Physics

Habilitation Thesis

2024

Tomas Petricek



CHARLES UNIVERSITY
Faculty of mathematics
and physics

Simple programming tools for data exploration

Tomas Petricek

Habilitation Thesis
Computer Science, Software Systems

Prague

2024

Acknowledgements

This thesis presents a selection of my recent research that focuses on making programming tools for data exploration simpler. The origins of this research direction can be traced to my involvement with the F# programming language. When Don Syme and the F# team at Microsoft developed the first versions of the F# type provider mechanism, I started my first experiments that eventually led to the type providers for structured data presented as one of the contributions in this thesis. However, seeing how easy working with data in a programming language can be led me to a further question: Could we make programmatic data exploration easy enough that it could be done by non-programmers? The rising popularity of data journalism at the time provided a further practical motivation.

The work presented in this thesis has been done at a number of institutions, starting towards the end of my PhD at the University of Cambridge and finishing as I was joining the Department of Distributed and Dependable Systems (D3S) at the Faculty of Mathematics and Physics. In between, I spent time at Microsoft Research in Cambridge, The Alan Turing Institute in London and the University of Kent in Canterbury. I am grateful to all those institutions for enabling me to pursue my research vision.

Although I am the first author of most of the work presented in this thesis, none of it would be possible without the many collaborators that I was fortunate to meet along the way. Don Syme not only provided the initial motivation and collaborated with me on multiple papers but also became my long-term mentor and friend. The work related to F# received a warm welcome from the friendly F# community and early commercial adopters. Gustavo Guerra deserves special credit for turning F# Data from a prototype to a well-engineered (and widely adopted) package.

At The Alan Turing Institute, I was fortunate to meet James Geddes who got me involved in the AI for Data Analytics (AIDA) project. The bridging of different worlds that James made possible resulted in my involvement in research on notebooks and data provenance with Charles Sutton, but also work on automating data wrangling with Gerrit van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz and Chris William. The Alan Turing Institute also provided initial funding for our joint work on data visualization tools with Roly Perera.

In addition to direct collaborators, the work presented in this thesis benefited from numerous discussions with my other colleagues and friends. This includes Dominic Orchard and Stephen Kell first at the University of Cambridge and then at the University of Kent, Kenji Takeda, Jomo Fisher, and Keith Battocchi at Microsoft Research and May Yong, Nick Barlow, Brooks Paige at The Alan Turing Institute. Mathias Brandewinder, Jonathan Edwards, Nour Boulahcen, Luke Church, Clemens Klokmoose, Mariana Marasoiu and Alan Blackwell also provided ideas, insights, technical contributions and valuable feedback on some of the works presented as part of this thesis.

The work presented in this thesis also received valuable support from industrial collaborators. My research focused on data journalism benefited from discussions with Megan Lucero from The Bureau of Investigative Journalism. My work on F# was supported by the wide and enthusiastic F# community and also by Howard Mansell and BlueMountain Capital. I had the pleasure of presenting many of the ideas at multiple industry conferences, including NDC in Oslo and London, LambdaDays and DevDay in Kraków, GOTO in Copenhagen and Chicago, CogX London and, most recently, the F# Data Science conference in Berlin. These presentations were vital not only for enabling industry adoption of some of the systems presented in this thesis but also provided valuable feedback.

Over time, the work has been financially supported in a number of ways. The Google Digital News Initiative provided me with a generous individual grant that allowed me to fully focus on programming tools for data journalism for one and a half years. Microsoft Research, BlueMountain Capital, the University of Kent, and Charles University paid for some of my time over the years. At The Alan Turing Institute, I was supported by The UKRI Strategic Priorities Fund under EPSRC Grant EP/T001569/1, particularly the Tools, Practices and Systems theme within that grant, through the UK Government's Defence & Security Programme and by The Alan Turing Institute under EPSRC grant EP/N510129/1. At Charles University, I was a part of the Department of Distributed and Dependable Systems and I was also supported by the PRIMUS grant PRIMUS/24/SCI/021.

Contents

Acknowledgements	3
Contents	4
I Commentary	8
1 Introduction	9
1.1 How data journalists explore data	10
1.2 Requirements of simple tools for data exploration	11
1.3 Data exploration as a programming problem	12
1.4 Utilised research methodologies	13
1.5 What makes a programming tool simple	14
1.6 Structure of the thesis contributions	15
1.7 Research outlook	18
2 Type providers	19
2.1 Information-rich programming	20
2.2 Type providers for semi-structured data	21
2.2.1 Shape inference and provider structure	23
2.2.2 Relative safety of checked programs	24
2.2.3 Stability of provided types	25
2.3 Type providers for query construction	26
2.3.1 Formalising lazy type provider for data querying	27
2.3.2 Safety of data acquisition programs	28
2.4 Contributions	28
3 Data infrastructure	30
3.1 Notebooks and live programming	31
3.2 Live data exploration environment	32
3.2.1 Data exploration calculus	33
3.2.2 Computing previews using a dependency graph	35
3.3 Live, reproducible, polyglot notebooks	37
3.3.1 Architecture of a novel notebook system	38
3.3.2 Dependency graphs for notebooks	39
3.4 Contributions	40

4	Iterative prompting	42
4.1	Data wrangling and data analytics	43
4.2	Iterative prompting	44
4.2.1	Iterative prompting for data querying	45
4.2.2	Usability of iterative prompting	46
4.3	AI assistants	47
4.3.1	Merging data with Datadiff	48
4.3.2	Formal model of AI assistants	49
4.3.3	Practical AI assistants	50
4.4	Contributions	52
5	Data visualization	54
5.1	Visualisations to encourage critical thinking	55
5.2	Composable data visualisations	56
5.2.1	Declarative chart descriptions	57
5.2.2	Rendering a Compost chart	58
5.2.3	Functional abstraction and interactivity	58
5.3	Automatic linking for data visualizations	59
5.3.1	Creating linked visualizations using Fluid	60
5.3.2	Language-based foundation for explainable charts	62
5.3.3	Bidirectional dependency analyses	62
5.4	Contributions	63
II	Publications: Type providers	65
6	Types from data: Making structured data first-class citizens in F#	66
7	Data exploration through dot-driven development	81
III	Publications: Data infrastructure	109
8	Foundations of a live data exploration environment	110
9	Wrattler: Reproducible, live and polyglot notebooks	147
IV	Publications: Iterative prompting	152
10	The Gamma: Programmatic data exploration for non-programmers	153
11	AI Assistants: A framework for semi-automated data wrangling	161
V	Publications: Data visualization	179
12	Composable data visualisations	180

13	Linked visualizations via Galois dependencies	199
VI	Conclusions	229
14	Contributions and outlook	230
14.1	Contributions to included publications	230
14.2	Open-source software contributions	231
14.3	New look at data exploration	232
14.4	Towards programming systems research	233

Part I

Commentary

Chapter 1

Introduction

The rise of big data, open government data initiatives (Attard et al., 2015),¹ and civic data initiatives mean that there is an increasing amount of raw data available that can be used to understand the world we live in, while increasingly powerful machine learning algorithms give us a way to gain insights from such data. At the same time, the general public increasingly distrusts statistics (Davies, 2017) and the belief that we live in a post-truth era has become widely accepted over the last decade.

While there are complex socio-political reasons for this paradox, from a merely technical perspective, the limited engagement with data-driven insights should perhaps not be a surprise. We lack accessible data exploration technologies that would allow non-programmers such as data journalists, public servants, and analysts to produce transparent data analyses that can be understood, explored, and adapted by a broad range of end-users including educators, the public, and the members of the civic society.

The technology gap is illustrated in Figure 1.1. On the one hand, graphical tools such as spreadsheets are easy to use, but they are limited to small tabular data sets, they are error-prone (Panko, 2015) and they do not aid transparency. On the other hand, programmatic tools for data exploration such as Python and R can tackle complex problems but require expert programming skills for completing even the simplest tasks.

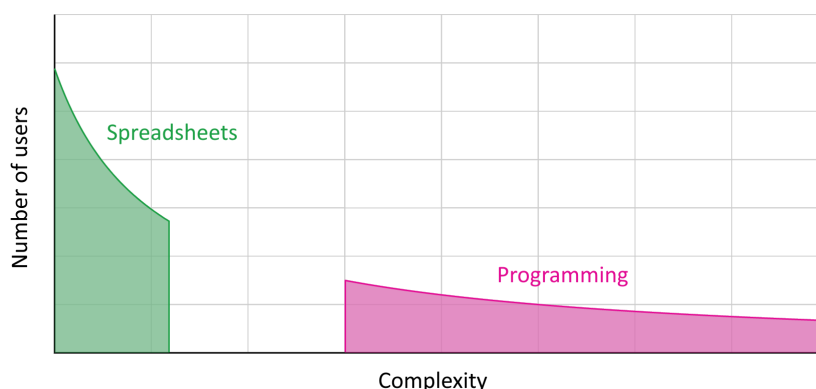


Figure 1.1: The gap between programming and spreadsheets – spreadsheets can be used by many people, but solve problems of a limited complexity. Programming scales arbitrarily, but has a high minimal complexity limiting the number of users. Adapted from Edwards (2015).

¹See <https://data.gov> and <https://data.gov.uk>, but also <https://opendata.gov.cz> as examples.

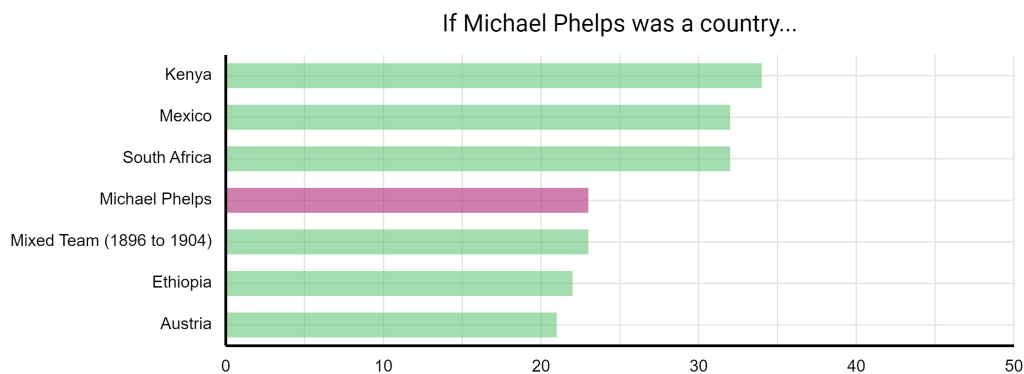


Figure 1.2: A visualization comparing the number of gold Olympic medals won by Michael Phelps with countries that won a close number of gold medals. Inspired by e.g., [Myre \(2016\)](#)

The above illustration should not be taken at face value. Although there is no single accepted solution, there are multiple projects that exist in the gap between spreadsheets and programming tools. However, the gap provides a useful perspective for positioning the contributions presented in this thesis. Some of the work I present develops novel tools that aim to combine the simplicity of spreadsheets with the power of programming for the specific domain of data exploration, aiming to fill the space in the middle of the gap. Some of the work I present focuses on making regular programming with data easier, or making simple programming with data accessible to a greater number of users, reducing the size of the gap on the side of programming.

1.1 How data journalists explore data

To explain the motivation behind this thesis, I use an example data exploration done in the context of data journalism ([Bounegru and Gray, 2021](#)). Following the phenomenal success of the swimmer Michael Phelps at the 2016 Olympic games, many journalists produced charts such as the one in Figure 1.2, which puts Phelps on a chart showing countries with similar numbers of medals. Even such a simple visualization raises multiple questions. Is the table counting Gold medals or all medals? How would it change if we used the other metric? What would it look like if we added more countries or removed the historical “Mixed Team”? How many top countries were skipped?

This simple example illustrates two challenges that I hinted at earlier. First, producing this visualization may not be hard for a programmer, but it involves a number of tricky problems for a non-programmer. The author has to acquire and clean the source data, aggregate medals by country and join two subsets of the data. Doing so manually in a spreadsheet is tedious, error-prone and not reproducible, but using Python or R requires non-trivial programming skills. Second, the non-technical reader of the newspaper article may want to answer the above follow-up questions. Data journalists sometimes offer a download of the original dataset, but the reader would then have to redo the analysis from scratch. If the data analysis was done in Python or R, they could get the source code, but this would likely be too complex to modify.

This thesis presents a range of tools that allow non-programmers, such as data journalists, to clean and explore data, such as the table of Olympic medals, and produce data

```

1 let data = olympics.'group data'.'by Team'.'sum Gold'.then
2   .'sort data'.'by Gold descending'.then
3   .paging.skip(42).take(6)
4   .'get series'.'with key Team'.'and value Gold'
5
6 let phelps = olympics.'filter data'.'Athlete is'.'Michael Phelps'.then
7   .'group data'.'by Athlete'.'sum Gold'.then
8   .'get series'.'with key Athlete'.'and value Gold'
9
10 charts.bar(data.append(phelps).sortValues(true))
11   .setColors(["#94c8a4", "#94c8a4", "#94c8a4", "#e30c94"])

```

Figure 1.3: Source code of the data analysis used to produce the visualization in Figure 1.2. The case study is based on the work presented in Chapter 10.

analyses that are backed by source code in a simple programming language that can be read and understood without sophisticated programming skills. In some cases, the code can be produced interactively, by repeatedly choosing one from a range of options offered by the tool and can then be modified to change the parameters of the visualization.

As an example, the source code of the data analysis used to produce the visualization above is shown in Figure 1.3. The tools that enable non-programmers to create it will be discussed later. The key aspect of the code is that it mostly consists of a sequence of human-readable commands such as 'filter data'.'Athlete is'.'Michael Phelps'. Those are iteratively selected from options offered by the system and so the author of the data analysis can complete most of the analysis without writing code.

The use of a simple programming language also makes it possible to understand the key aspects of the logic. The analysis counts the number of gold medals ('sum Gold'), skips 42 countries before the ones shown in the visualization, and does not filter out any other data. Finally, the code can be easily executed (in a web browser), allowing the reader to easily make small changes, such as picking a different athlete or increasing the number of displayed countries. Such engagement has the potential to aid the reader's positive perception of open, transparent data-driven insights based on facts.

1.2 Requirements of simple tools for data exploration

Although the tools and techniques presented in this thesis are more broadly applicable, the focus of this thesis is on a narrower domain illustrated by the above motivating example. I focus on programmatic data exploration tools that can be used to produce accessible and transparent data analyses that will be of interest to a broader range of readers and allow them to critically engage with the data.

In the subsequent discussion, I thus distinguish between *data analysts* who produce the analyses and *readers* who consume and engage with the results. The former are technically skilled and data-literate, but may not have programming skills. The latter are non-technical domain experts who may nevertheless be interested in understanding and checking the analysis or modifying some of its attributes. This context leads to a number of requirements for the envisioned data exploration tools:


- *Gradual progression from simple to complex.* The system must allow non-programmers with limited resources to easily complete simple tasks in an interface that allows them to later learn more and tackle harder problems. In the technical dimensions of programming systems framework (Jakubovic et al., 2023), this is described as the staged levels of complexity approach to the learnability dimension.
- *Support transparency and openness.* The readers of the resulting data analyses must be able to understand how the analysis was done and question what processing steps and parameters have been used in order to critically engage with the problem.
- *Enable reproduction and learning by percolation.* A reader should be able to see and redo the steps through which a data exploration was conducted. This lets them reproduce the results, but also learn how to use the system. As noted by Sarkar and Gordon (2018), this is how many users learn the spreadsheet formula language.
- *Encourage meaningful reader interaction.* The reader should not be just a passive consumer of the data analyses. They should be able to study the analysis, but also make simple modifications such as changing analysis or visualization parameters, as is often done in interactive visualizations by journalists (Kennedy et al., 2021).

The criteria point to the technology gap illustrated by Figure 1.1 and there are multiple possible approaches to satisfy the criteria. This thesis explores one particular point in the design space, which is to treat data analysis as a program with an open source code, created in a simple programming language with rich tooling.

As I will show, treating data exploration as a programming problem makes it possible to satisfy the above criteria. Gradual progression from simple to complex can be supported by a language that provides very high-level abstractions (or domain-specific languages) for solving simple problems. Transparency, openness, and reproducibility are enabled by the fact that the source code is always available and can be immediately executed while learning by percolation can be supported by structuring the program as a sequence of transformations. Finally, meaningful interaction can be offered by suitable graphical tools that simplify editing of the underlying source code.

1.3 Data exploration as a programming problem

Data exploration is typically done using a combination of tools including spreadsheets, programming tools, online systems, and ad-hoc utilities. Spreadsheets like Excel and business intelligence tools like Tableau (Wesley et al., 2011) are often used for manual data editing, reshaping, and visualization. More complex and automated data analyses are done in programming languages like R and Python using a range of data processing libraries such as pandas and Tidyverse (Wickham et al., 2019). Such analyses are frequently done in a computational notebook environment such as RStudio or Jupyter (Kluyver et al., 2016), which make it possible to interleave documentation, mathematical formulas and code with outputs such as visualizations. Online data processing environments like Trifacta provide myriads of tools for importing and transforming data, which are accessible through different user interfaces or programmatically, but even those have to be complemented with ad-hoc single-purpose tools, often invoked through a command line interface. Finding a unified perspective for thinking about such a hotchpotch of systems and tools is a challenge.

 **Key novel perspective.** In this thesis, I propose to view systems and tools used for data exploration as *programming tools*. This view can offer a unified perspective on a broad range of systems and tools. It also enables us to apply the powerful methodology of programming languages research to the problem of data exploration.

If we look at data exploration tools from the perspective of programming languages research, we can adapt and leverage techniques for ensuring program correctness and compositional design, as well as rich interaction principles. However, the programs that are constructed during data exploration have a number of specific characteristics that distinguish them from programs typically considered in programming language research:

- *Data exists alongside code.* Systems such as spreadsheets often mix data and code in a single environment. In conventional programming, this is done in image-based systems like Smalltalk, but not in the context of programming languages.
- *Concrete inputs are often known.* Moreover, data exploration is typically done on a known collection of concrete input datasets. This means that program analysis can take this data into account rather than assuming arbitrary unknown inputs.
- *Programmers introduce fewer abstractions.* Even in programmatic data exploration using R or Python in a Jupyter notebook, data analysts often write code as a sequence of direct operations on inputs or previously computed results, rather than introducing abstractions such as reusable generic functions.
- *Most libraries are externally defined.* Finally, data exploration is often done using libraries and tools that are implemented outside of the tool that the analysts use. For example, spreadsheet formulas use mostly built-in functions, while data analyses in Python often use libraries implemented in C/C++ for performance reasons.

The above holds for simple data explorations, such as those done by data journalists that this thesis is concerned with. The characteristics do not apply to all programs that work with data. Reusable and parameterized models, general-purpose algorithms and rich data processing pipelines share structure with conventional programs. However, focusing on simple data explorations for which the above criteria are true allows us to narrow the design space and study a range of interesting problems. The narrow focus also makes us rethink a number of accepted assumptions in programming language research, such as what are the key primitives of a programming language (in Chapter 8, an invocation of an external function becomes more important than lambda abstraction).

1.4 Utilised research methodologies

The research presented in this thesis tackles multiple research questions such as: Does a particular language design rule out certain kinds of programming errors? What is an efficient implementation technique for a particular language or a tool? Does a newly developed tool simplify data exploration by reducing the number of manual interventions by the user? What is a suitable interaction mechanism for completing a particular task? And can non-programmers effectively use such interaction mechanism? The diversity of the research questions calls for a corresponding diversity of research methodologies.

Programming language theory. The first methodology used in this thesis is that of theoretical programming language research. When using this methodology, a core aspect of a programming language is described using a small, formally tractable mathematical model that captures the essential properties of the aspect. The model is then used to formally study properties of the given aspect, such as whether a programming language that implements it can be used to write programs that exhibit a certain kind of incorrect behavior.

In this thesis, Part II presents two instances of a programming language extension mechanism called type providers. To show that code written using type providers will never result in a particular error condition, I develop a formal model of type providers and prove a correctness property using the model. The actual system implementation then closely follows the formal model. Theoretical programming language research methods are also used to develop a data visualization language in Chapter 13, to formalize the optimization technique introduced in Chapter 8 and to define the structure of the semi-automatic data wrangling tools developed in Chapter 11.

Programming systems. The theoretical approach is complemented by a range of applied programming systems methods. The work using those methodologies often focuses on designing suitable system architecture, empirical evaluation of measurable characteristics of the system such as efficiency. It should also be complemented with an open-source implementation and/or a reproducible software artifact.

I use the programming systems research methodology primarily in Chapter 9, which presents the architecture and implementation of a novel computational notebook system for data science. Chapter 8 develops an optimized programming assistance tool and evaluates the efficiency empirically. Software systems and libraries presented in this thesis are available as open-source and are listed below.

Human-computer interaction. Finally, answering questions that concern usability requires a human-centric approach offered by the human-computer interaction (HCI) research methodology, which is increasingly used to study programming languages and systems (Chasins et al., 2021). The HCI methods include controlled usability studies, qualitative and quantitative user studies, as well as the development and application of heuristic evaluation frameworks.

I use the HCI methodology in Chapter 10, which introduces the “iterative prompting” interaction mechanism and conducts a usability study with non-programmers to evaluate whether they can use it to complete simple data exploration tasks. Chapter 12, which presents a novel data visualization library, also draws on the HCI methodology, but uses a comprehensive case study instead of a user study to evaluate the design.

1.5 What makes a programming tool simple

The very title of this thesis refers to the aim of creating programming tools for data exploration that are *simple*. However, simplicity is difficult to quantify precisely. It is understood differently by different communities and in different contexts. I thus follow the recommendation of Muller and Ringler (2020) to make explicit how the term should be understood in the context of this thesis. The notion of simplicity is used as a unifying theme in this commentary. In the papers presented as part of this thesis, the notion takes one of several more specific and rigorously evaluated forms:

- In the context of user-centric work, I refer to a system as *simple* if it allows non-programmers to complete tasks that are typically limited to programmers. This is the case when discussing the iterative prompting interaction principle in Chapters 10 and the live programming tools in Chapter 8.
- In the context of programming language or library design, I consider the design *simple* when it allows expressing complex logic using a small set of highly composable primitives that are easy to understand. This applies to the language design in Chapter 7 and visualization library design in Chapter 12.
- In the context of programmer assistance tools, simple indicates that the user does not have to perform a task that they would otherwise have to complete manually. This applies to AI assistants, presented in Chapter 11, which relieve the user from tedious manual setting of parameters by partly automating the task.
- Finally, I also use the term simple when talking about programming systems and libraries that provide a high-level interface designed specifically for a particular task. This is the case for the notebook system presented in Chapter 9, data access library in Chapter 6, and the language for creating visualizations in Chapter 13. Using such high-level abstractions means that programmers have to write less code.

The overarching theme of this thesis is thus the design of programming tools for data exploration that are simple in one or more of the meanings of the term indicated above. The focus on simplicity aims to fill or reduce the technology gap illustrated in Figure 1.1 and, ultimately, make data exploration accessible to a broader range of users.

1.6 Structure of the thesis contributions

The key novel perspective—to view data exploration tools from the perspective of programming language research—can be leveraged for a wide range of different data exploration tools, including tools for data acquisition, data cleaning and data visualization. Correspondingly, the contributions presented in this thesis cover multiple different kinds of tasks that a data analyst faces when they work with data.

To position the contributions in the broader context of data analytical work, it is useful to see where they fit in a typical data science lifecycle. For this thesis, it is useful to consider a variant of the lifecycle that distinguishes between the exploration and production phases as done by [Jain and Kushagra \(2022\)](#) as well as [IBM \(2020\)](#). As shown in Figure 1.4, the contributions of this thesis focus on the work done in the initial data exploration phase. Unlike with the later production phase, the programs used in the exploration phase typically exhibit the unique characteristics discussed in Section 1.3.

The data science lifecycle starts with data acquisition (1), which involves loading data from a range of sources. This is followed by data cleaning (2), where multiple data sources are joined, incomplete data is filled or removed and data structure is recovered. In data exploration (3), the analyst transforms the data to discover interesting patterns and, finally, in data visualization (4) they produce charts to present their insights. In the production phase, the insights are then used to develop a model that becomes a part of a production system. The process can be repeated based on the results of the model evaluation.

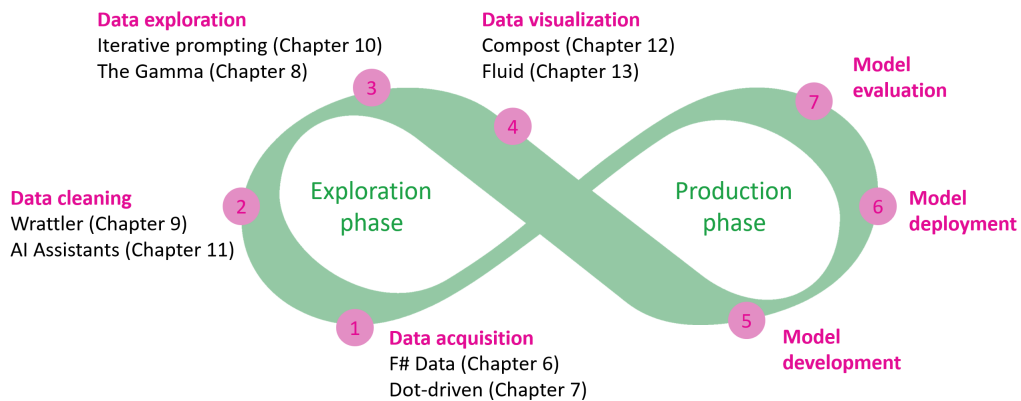


Figure 1.4: Illustration showing the data science lifecycle, as understood by , alongside with the contributions of this thesis to the individual steps of the data exploration phase.

The work that constitutes this thesis contributes to each of the four steps of the data exploration phase. In Part II, I present two papers on type providers, which simplify data acquisition, while Part V consists of two novel data visualization systems. The four publications presented in Part III and Part IV all focus on working with data, including data cleaning and exploration. They are not grouped in parts based on the lifecycle step but based on their research methodology. The publications in Part III use programming systems methods to design new infrastructure, while Part IV introduces a novel interaction principle and applies it to two problems, one from the domain of data exploration and one from the domain of data cleaning. The rest of this section summarises the contributions of the work presented in this thesis in more detail.

Type providers. The *type provider* mechanism (Syme et al., 2012, 2013) makes it possible to integrate external data into a statically-typed programming language. The work presented in Part II presents two new type providers.

Chapter 6 presents a library of type providers that makes it possible to safely access structured data in formats such as CSV, XML, and JSON in the F# programming language. The two key research contributions of the work are, first, a novel inference mechanism that infers a type based on a collection of sample data and, second, a formulation of a *relative safety property* that formally captures the safety guarantees offered by the system.

Chapter 7 takes the idea of type providers further. It uses the mechanism not just for data access, but for the construction of SQL-like queries over tabular data. The research contribution is a novel type provider, implemented in The Gamma system, which generates a type that can be used to group, filter, and sort tabular data. Using a novel formal model, the presented paper shows that all queries constructed using the type provider are valid.

Data infrastructure. Programmatic data exploration is typically done in notebook systems such as Jupyter (Kluyver et al., 2016) that make it possible to combine documentation, formulas, code, and output such as visualizations. Notebook systems are a convenient tool, but they suffer from a number of limitations and issues. The two novel systems presented in Part III address several of those.

Chapter 8 presents a programming environment for The Gamma that makes data exploration easier by providing instant feedback. The research contributions of the work are twofold. First, it builds a practical efficient algorithm for displaying live previews. Second, it develops a formal model of code written to explore data called *data exploration calculus* and uses it to show the correctness of the live preview algorithm.

Chapter 9 tackles more directly the problems of notebook systems. It presents Wrattler, which is a novel notebook system that makes it possible to combine multiple programming languages and tools in a single notebook resolves the reproducibility issues of standard systems and stores computation state in a transparent way, allowing for precise data provenance tracking.

Iterative prompting Treating data analyses as programs makes them transparent and reproducible, but writing code has an unavoidable basic complexity. Part IV presents a novel interaction principle for program construction called *iterative prompting*. The mechanism is rooted in the work on type providers and makes it possible to construct programs by repeatedly choosing from one of several options.

Chapter 10 introduces the iterative prompting mechanism from the human-computer interaction perspective. It shows that the mechanism can be used to construct programs that explore data in multiple input formats including tables, graphs and data cubes. The usability of the mechanism is evaluated through a user study, showing that non-programmers can use it to complete a range of data exploration tasks.

Chapter 11 uses the iterative prompting mechanism as the basis of a range of semi-automatic data cleaning tools. It augments existing AI tools for parsing data, merging data, inferring data types and semantic information with a mechanism that lets the user guide the AI tool. Using iterative prompting, the user can correct mistakes and configure the parameters of the tool. The augmented tools are evaluated empirically, showing that the correct result can typically be obtained with 1 or 2 manual interventions.

Data visualization. Data visualization is the last step in the exploratory phase of the data science lifecycle discussed above. Although standard charts are typically easy to build, creating richer interactive visualizations is a challenging programming task. Part V presents two systems that make it easier to build interactive data visualizations that encourage critical thinking about data.

Chapter 12 presents a functional domain-specific language for creating charts that makes it possible to compose rich interactive charts from basic building blocks (such as lines and shapes) using a small number of combinators (such as overlaying and nesting of scales). The simplicity of the approach is illustrated through a range of examples and confirmed by the publication of the work as a so-called functional pearl (Gibbons, 2010).

Chapter 13 introduces a language-based program analysis technique that makes it possible to automatically build linked data visualizations that show the relationships between parts of charts produced from the same input data. The key research contribution is a novel bidirectional dynamic dependency program analysis, which is formalized and shown to have a desirable formal structure. The technique is used as the basis for a high-level programming language Fluid.

A Key novel perspective. A close look at how data scientists interact with programming tools forces us to rethink how we conceptualize programming. It shows that we need to shift our attention from static *programming languages* to rich, stateful, and interactive *programming systems*. Understanding the theory and practice of those remains an interesting open problem.

1.7 Research outlook

Viewing data exploration from the perspective of programming language research is beneficial in both directions. Most of this thesis is concerned with the novel data exploration tools and systems that become conceivable as a result of this perspective. However, an equally interesting question is whether data exploration forces us to think about (conventional) programming differently. I believe this is the case.

As noted earlier, data scientists often work with concrete data that exists alongside with code. This is an approach that has existed in image-based programming systems since the era of Smalltalk. They also often interleave coding with execution, which is how most modern programs are constructed. Although many programming environments discard any state of the executing program, hot-reloading is increasingly used to make sure programmers do not lose state while editing code.

Most contemporary programming language research focuses solely on languages and ignores such stateful aspects of programming systems, possibly due to the paradigm shift documented by [Gabriel \(2012\)](#). This ignores an important aspect of the reality of modern programming. Moreover, the new capabilities presented in this thesis in the context of data science suggest that the programming systems perspective has the potential to yield fruitful results about programming in a broader sense. This is also an area that I started exploring in recent years in joint work with [Jakubovic et al. \(2023\)](#); [Edwards and Petricek \(2021\)](#); [Edwards et al. \(2025\)](#).

Chapter 2

Type providers

The first step of the data science lifecycle outlined in the previous chapter was data acquisition. This typically involves reading data in semi-structured formats such as CSV, XML, and JSON or retrieving data from a database. The aim of the work on type providers, outlined in this chapter, is to make programmatic data acquisition reliable and simpler.

The lack of reliability arises primarily from the fact that most data access code is written in dynamically-typed scripting languages. This is largely because using such languages is easier. A dynamically-typed language does not need to consider the structure of the input data to check that the program accesses it correctly. If we retrieve a JSON object that represents a record with fields `title` and `link` and parse it into an object `item` in JavaScript, we can then access the fields using just `item.title` and `item.link`. The fields will exist at runtime, but the language does not need to know at compile-time whether they will be available, because member access is not statically checked.

In statically-typed programming languages, the situation is no better. The typical approach, illustrated in Figure 2.1, is equally dynamic, but more verbose. Object fields are accessed using a string-based lookup, which can easily contain fields that do not exist at runtime (indeed, there is an uncaught typo on line 6!) and, moreover, the lookup has to be done using an additional method invocation and may require tedious type conversions. The first challenge we face is how to make accessing data in semi-structured formats, such as JSON, XML, and CSV, as simple as in dynamically-typed languages (a matter of just using a dot), but support checking that will statically guarantee that the accessed fields will be present at runtime.

However, the simplicity of data access in dynamic scripting language also has its limits. It is easy to access individual fields, but the code gets more complicated if we want to perform a simple query over the data. Consider, for example, the query in Figure 2.2.

```
1 var url = "http://dvd.netflix.com/Top100RSS";
2 var rss = XDocument.Load(topRssFeed);
3 var channel = rss.Element("rss").Element("channel");
4
5 foreach(var item in channel.Elements("item")) {
6     Console.WriteLine(item.Element("title").Value);
7 }
```

Figure 2.1: Printing titles of items from an RSS feed in C#. The snippet uses dynamic lookup to find appropriate elements in the XML and extracts and prints the title of each item.

```

1 olympics = pd.read_csv("olympics.csv")
2 olympics[olympics["Games"] == "Rio (2016)"]
3   .groupby("Athlete")
4   .agg({"Gold": sum})
5   .sort_values(by="Gold", ascending=False)
6   .head(8)

```

Figure 2.2: Data transformation written using pandas in Python. The code loads a CSV file with Olympic medal history, gets data for Rio 2016 games, groups the data by the athlete, and sums their number of gold medals and, finally, takes the top 8 athletes.

Despite being widely accepted as simple, the Python code snippet involves a remarkable number of concepts and syntactic elements that the user needs to master:

- *Generalised indexers* (`.[condition]`) are used to filter the data. This is further complicated by the fact that `==` is overloaded to work on a data series and the indexer accepts a Boolean-valued series as an argument.
- *Python dictionaries* (`{"key": value}`) are here used not to specify a lookup table, but to define a list of aggregation operations to apply on individual columns. It also determines the columns of the returned data table.
- *Well-known names*. The user also has to remember the (somewhat inconsistently named) names of operations such as `groupby` and `sort_values` and remember the column names from their data source such as "Athlete".

To make data acquisition simpler, the user should not need this many concepts and they should not need to remember the names of operations or the names of columns in their data source. Moreover, their code should be checked to ensure that it accesses the correct supported operations and applies them to compatible data that exist in the data source. As I will show later, this can be achieved using type providers, a concept that originated in the F# programming language in the early 2010s.

2.1 Information-rich programming

In the 2010s, applications increasingly relied on external data sources and APIs for their function. The typical solution for accessing such data was either to use a scripting language, a dynamic access library (both illustrated above), or a code-generation tool that would generate code for accessing the data source or an API (although only for data sources with small enough schema). This provided the motivation for the type provider mechanism in F# (Syme et al., 2012, 2013), which made it possible to make the type checker in a statically-typed programming language aware of the structure of external data sources.

Technically, a type provider in F# is an extension that is executed by the compiler at compile-time. A type provider can run arbitrary code, such as accessing a database schema or another external data source. It then generates a representation of a type that is passed to the compiler and used to check the user program. For example, the World Bank type provider (Figure 2.3) retrieves the list of known countries and indicators from the World Bank database (by querying the REST API provided by the World Bank) and generates a collection of types. The `WorldBank` type has a `GetDataContext` method, which returns an

```

1 type WorldBank = WorldBankDataProvider<"World Development Indicators">
2 let data = WorldBank.GetDataContext()
3
4 data.Countries.'United Kingdom'.Indicators
5   .'Central government debt, total (% of GDP)''

```

Figure 2.3: The World Bank type provider (Syme et al., 2012) provides access to indicators collected by the World Bank. The countries and indicators are mapped to properties (members) of an F# class that represents the data.

instance of a type with the `Countries` member and the type returned by this member has one member corresponding to each country in the World Bank database. The World Bank type provider, created by the author of this thesis and presented in a report (Syme et al., 2012) not included here, shows two important properties of type providers:

- *Static type provider parameters.* A type provider in F# can take literal values (such as "World Development Indicators") as parameters. They can be used when the provider is executed (at compile-time) to guide how types are generated. Here, the parameter specifies a particular database to use as the source. These can be names of files with schema, connection strings or live URLs.
- *Lazy type generation.* The types generated by a type provider are generated lazily, i.e., the members of a type (and the return types of those members) are only generated when the type checker encounters the type in code. This makes it possible to import very large (potentially infinite) external schema into the type system.

There are other interesting aspects of type providers, but the above two features are crucial for the work included in this thesis. In the following two sections, I review the key contributions to type providers presented in Part II make data acquisition reliable and simpler. The work on type providers included in this thesis develops two kinds of type providers. The type providers for CSV, JSON, and XML packaged in the F# Data library make it possible to access data in a statically-checked way using ordinary member access. The work also makes two theoretical contributions, an algorithm for schema inference from sample data and a programming language theory of type providers.

The pivot type provider, developed for the experimental programming language The Gamma, makes it possible to construct queries such as that shown in Figure 2.2 (and was mentioned briefly in Section 1.1). It adapts the theory developed for the F# Data type providers to show that only correct queries can be constructed when using it. The full account of the work can be found in Chapter 6 and Chapter 7, respectively. The following provides an accessible high-level overview of the contributions.

2.2 Type providers for semi-structured data

The F# Data library implements type providers for accessing data in XML, JSON, and CSV formats. It is based on the premise that most real-world data sources using those formats do not have an explicit schema. The type providers thus infer the schema from a sample (or a collection of samples). The inferred schema is then mapped to F# types through which the user of the type provider can access the data.

```

1 // worldbank.json - a sample response used for schema inference
2 [ { "page": 1, "pages": 1, "per_page": "1000", "total": 53 },
3   [ { "indicator": { "id": "GC.DOD.TOTL.GD.ZS" },
4     "country": { "id": "CZ" },
5     "date": "2011", "value": null },
6     { "indicator": { "id": "GC.DOD.TOTL.GD.ZS" },
7       "country": { "id": "CZ" },
8       "date": "2010", "value": 35.1422970266502 } ] ]

1 // demo.fsx - a data acquisition script using a type provider
2 type WB = JsonProvider<"worldbank.json">
3 let wb = WB.Load("http://api.worldbank.org/.../GC.DOD.TOTL.GD.ZS?json")
4
5 printf "Total: %d" wb.Record.Total
6 for item in wb.Array do
7   match item.Value with
8   | Some v -> printf "%d %f" item.Date v
9   | _ -> ()

```

Figure 2.4: Using the .JSON type provider for accessing data from a REST API. The inference uses a local sample file, while at runtime, data is obtained by calling the live service.

The example shown in Figure 2.4 illustrates one typical use. Here, the user is accessing information from a service that returns data as JSON (incidentally, the service is also the World Bank, but here we treat it as an ordinary REST service). The user stored a local copy of a sample response from the service (`worldbank.json`) and uses it as a static parameter for the JSON type provider (line 2). They then load data from the live service (line 3) and print the total number of items (line 5) as well as each year for which there is a value (line 8). Three aspects of the type provider deserve particular attention:

- *Real-world schema inference is hard.* Here, the response is an array always containing two items, a record with meta-data and an array with individual data points. The data records have a consistent structure, although some values may be `null`.
- *Inference needs to be stable.* The type providers allow adding further samples. If the user adds further examples, the structure of the provided types should change in a predictable (and limited) way so that the user code can be easily updated.
- *Safety guaranteed by static checks is relative.* Static type checking guarantees that only data available in the sample input can be accessed in user code, but if the data loaded at runtime has a different structure, this will not prevent errors. We thus need to specify what exactly can the system guarantee about programs.

The F# Data type providers, presented in full in Chapter 6 offer an answer to all of these three challenges. They can infer the shape of real-world data, infer types with a stable structure, and capture the runtime guarantees formally through the relative safety property. The publication also presented novel programming language theory that made it possible to analyze type providers formally, which I briefly review in the next three sections.

2.2.1 Shape inference and provider structure

When the type provider for semi-structured data is used, it is given a sample of data that can be analyzed at compile time (such as the "worldbank.json" file name above). It uses this to infer the shape of the data. A shape is a structure similar to a type and is composed from primitive shapes, record shapes, collections, and a few other special shapes:

$$\begin{aligned}\hat{\sigma} &= \nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \} \\ &| \text{float} \mid \text{int} \mid \text{bool} \mid \text{string} \\ \sigma &= \mathbf{nullable}\langle \hat{\sigma} \rangle \mid [\sigma] \mid \mathbf{any} \mid \mathbf{null} \mid \perp\end{aligned}$$

The inference distinguishes between non-nullable shapes ($\hat{\sigma}$) and nullable shapes (σ), which can be inferred even when the collection of inputs contains the **null** value. The former consists of primitive shapes (inferred from a corresponding value) and a record shape. The record shape has an (optional) name ν and consists of multiple fields that have their own respective shapes. A record is the shape inferred for JSON objects, but also XML elements containing attributes and child elements. A non-nullable shape can be made nullable by explicitly wrapping it as **nullable** $\langle \hat{\sigma} \rangle$. Other nullable shapes include collections (a **null** value is treated as an empty collection) and shapes that represent any data, only null values, and the bottom shape \perp , representing no information about the shape. The above definition does not include the handling of choice shapes (corresponding to sum types), which is introduced later.

A key technical operation of the shape inference is expressed using the *common preferred shape* function written as $\sigma_1 \nabla \sigma_2 = \sigma$. Given two shapes, the function returns a shape the most specific shape that can be used to represent the values of both of the two given shapes. The details are discussed later, but it is worth illustrating how the function works using two examples.

- $\text{int} \nabla \text{float} = \text{float}$ In this case, the common preferred shape is `float`. This may lead to a loss of precision, but it makes accessing the data easier than if we inferred a shape representing a choice shape. This is one example where the system favors practical usability over formal correctness.
- $\{x : \text{int}\} \nabla \{x : \text{int}, y : \text{int}\} = \{x : \text{int}, y : \mathbf{nullable}\langle \text{int} \rangle\}$ In this case, the common preferred shape is a record where the field that was missing in one of the shapes is marked as **nullable**. In general, the system aims to infer records whenever possible, which is the key for the stability of inferred types discussed below.

When the type provider is used, it receives a sample data value and uses it to infer the expected shape of data. A data value is modeled formally as a value that can be either a primitive value (integer i , floating-point value f , string s , a Boolean or **null**), a collection of values or a record with fields that have other values:

$$\begin{aligned}d &= i \mid f \mid s \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \\ &| [d_1; \dots; d_n] \mid \nu \{ \nu_1 \mapsto d_1, \dots, \nu_n \mapsto d_n \}\end{aligned}$$

The shape inference is then defined as a function $S((d_1, \dots, d_n) = \sigma$ that takes a collection of data values and infers a single shape σ that represents the shape of all the specified values. (Note that this can always be defined. In cases where values are of incompatible shape, the system infers the shape **any**.)

$$\begin{aligned}
S(i) &= \text{int} & S(\mathbf{null}) &= \mathbf{null} & S(\mathbf{true}) &= \text{bool} \\
S(f) &= \text{float} & S(s) &= \text{string} & S(\mathbf{false}) &= \text{bool} \\
S([d_1; \dots; d_n]) &= [S(d_1), \dots, d_n] \\
S(\nu \{ \nu_1 \mapsto d_1, \dots, \nu_n \mapsto d_n \}) &= \nu \{ \nu_1 : S(d_1), \dots, \nu_n : S(d_n) \} \\
S(d_1, \dots, d_n) &= \sigma_n \text{ where } \sigma_0 = \perp, \forall i \in \{1..n\}. \sigma_{i-1} \nabla S(d_i) \vdash \sigma_i
\end{aligned}$$

The shape inference is primarily defined on individual data values. For those, the system infers the shape corresponding to the value. For lists, we infer the shape based on all the values in the list. Finally, the last rule handles multiple sample data values by inferring their individual shapes and combining them using the ∇ function.

The last aspect of the formal programming language model of type providers is the logic that, given an inferred shape, produces the corresponding F# type. To explain the important properties of type providers, we do not need to elaborate on what an F# type is here, but the most important case is a class with members (properties or methods). A type provider takes the inferred shape and produces an F# type τ for the shape, a collection of classes L that may appear in τ (typically as types of members in case τ is a class). The type provider also needs to generate code that turns a raw data value d passed as input at runtime into a value of the provided type τ , which is represented as an expression e :

$$\llbracket \sigma \rrbracket = (\tau, e, L) \quad (\text{where } L, \emptyset \vdash e : \text{Data} \rightarrow \tau)$$

The mapping $\llbracket \sigma \rrbracket$ takes an inferred shape σ and returns a triple consisting of an F# type τ , a function turning a data value into a value of type τ and a collection of classes L .

This brief overview of the formal model of type providers for semi-structured data makes it possible to formulate the two key results about the F# Data type providers. The first describes the relative type safety of programs written using a type provider and is a novel variation on the classic type safety property of programming language research. The second describes the stability of provided types and concerns the usability of the system.

2.2.2 Relative safety of checked programs

The aim of type systems, in general, is to ensure that programs which passed type checking do not contain a certain class of errors. This has been characterised by Milner (1978) using a famous slogan “Well typed programs do not go wrong” (with *wrong* being a formal entity in Milner’s system). A code written using a type provider can go wrong if the input obtained at runtime is of a structure that does not match the structure of the input used as a sample for shape inference at compile time.

However, thanks to the formal model defined above, the property can be specified precisely, and most importantly, we can specify for which inputs the programs written using a type provider will never fail because of invalid data access. The definition relies on a preferred shape relation \sqsubseteq , which captures the fact that one shape is more specific than another (if $\sigma_1 \sqsubseteq \sigma_2$ then $\sigma_1 \nabla \sigma_2 = \sigma_2$). The theorem is also defined in terms of the \rightsquigarrow operation, which captures the operational semantics of the programs of the language used in the formal model. The relation $e_1 \rightsquigarrow e_2$ specifies that an expression e_1 reduces to e_2 in a single step (and \rightsquigarrow^* is the transitive closure of \rightsquigarrow).

Theorem 1 (Relative safety). Assume d_1, \dots, d_n are samples, $\sigma = S(d_1, \dots, d_n)$ is an inferred shape and $\tau, e, L = \llbracket \sigma \rrbracket$ are a type, expression, and class definitions generated by a type provider.

For all inputs d' such that $S(d') \sqsubseteq \sigma$ and all expressions e' (representing the user code) such that e' does not contain any of the dynamic data operations op and any Data values as sub-expressions and $L; y : \tau \vdash e' : \tau'$, it is the case that $L, e[y \leftarrow e' d'] \rightsquigarrow^* v$ for some value v and also $\emptyset; \vdash v : \tau'$.

In other words, the relative safety property specifies that, for any program that the user may write using a type provider (without using low-level functions that are only accessible inside a type provider), if the program is executed with any input whose shape is more specific than the shape inferred from statically known samples, the program will not encounter a data-related runtime error. It is, of course, still possible for runtime errors to happen, but not with a well-chosen sample and, as the wide-ranging adoption of the F# Data library suggests,¹ this is often a sufficient guarantee in practice.

2.2.3 Stability of provided types

When the user of an F# Data type provider gets a runtime error, this is because the data source they use produces an input of a structure not encountered before. A typical example is an input that includes `null` in a field that previously always had a value. Such errors are inevitable (without an explicit schema). The programmer can handle this by adding the new input as a new sample to the collection of samples used for the shape inference.

If they do so, the type provider will provide a new different type. In this case, an important property of the system is that the newly provided type will have the same general structure as the type provided before. This means that the data processing code, written using the provided type, will be easy to adapt. The programmer will need to add handling of a missing value, but they will not have to restructure their code. (A system based on statistical analysis of similarity would not have this property as a small change in the input may affect a decision whether two shapes are sufficiently similar to be unified into a single type.) Using the formal model, we can capture this property (and later prove that it holds for the F# Data type providers).

Theorem 2 (Stability of inference). Assume we have a set of samples d_1, \dots, d_n , a provided type based on the samples $\tau_1, e_1, L_1 = \llbracket S(d_1, \dots, d_n) \rrbracket$ and some user code e written using the provided type, such that $L_1; x : \tau_1 \vdash e : \tau$. Next, we add a new sample d_{n+1} and consider a new provided type $\tau_2, e_2, L_2 = \llbracket S(d_1, \dots, d_n, d_{n+1}) \rrbracket$.

Now there exists e' such that $L_2; x : \tau_2 \vdash e' : \tau$ and if for some d it is the case that $e[x \leftarrow e_1 d] \rightsquigarrow v$ then also $e'[x \leftarrow e_2 d] \rightsquigarrow v$. Such e' is obtained by transforming sub-expressions of e using one of the following translation rules:

- (i) $C[e]$ to $C[\text{match } e \text{ with Some}(v) \rightarrow v \mid \text{None} \rightarrow \text{exn}]$
- (ii) $C[e]$ to $C[e.M]$ where $M = \text{tagof}(\sigma)$ for some σ
- (iii) $C[e]$ to $C[\text{int}(e)]$

The translation rules use a context $C[e]$ to specify that a transformation needs to be done somewhere in the program. Importantly, all the rules are *local* meaning that a change

¹The package is one of the most downloaded F# libraries at the <https://www.nuget.org> package repository and the open-source project at <https://github.com/fsprojects/FSharp.Data> has over 100 contributors.

```

1 olympics
2   .'filter data'.'Games is'.'Rio (2016)'.then
3   .'group data'.'by Athlete'.'sum Gold'.then
4   .'sort data'.'by Gold descending'.then
5   .'paging'.take(8)

```

Figure 2.5: Data transformation constructed using the pivot type provider. This implements the same logic as pandas code in Figure 2.5, computing the top 8 athletes from the Rio 2016 Olympic games based on their number of gold medals.

is done in a particular place in the program. The change can be (i) handling of missing value, (ii) accessing a newly introduced member when the change introduces a new choice type and (iii) adding a conversion of a primitive value.

2.3 Type providers for query construction

The type providers presented in the previous section are designed to allow easy programmatic access to data in semi-structured formats. The focus is on providing typed direct access to the data. The pivot type provider, presented in Chapter 7, builds on the same concepts but focuses on letting users construct queries over tabular data. The user should not just be able to fetch the data in a typed form, but also use the provided types to filter, aggregate, and reshape the data.

The use of the pivot type provider is illustrated in Figure 2.5, which implements the data querying logic written using the pandas Python library in Figure 2.2. The type provider is implemented in the context of The Gamma programming language, which is a simple statically typed programming language with class-based object model and type providers that runs in the web browser.

As the code sample shows, the querying is implemented as a single chain of member accesses. Except for `take`, which is a method with a numerical parameter, all the members are properties that return another object of another class type with further members that can be used to continue constructing the query (the symbol `'` is used to wrap names containing a space). The system has a number of properties:

- *Discoverability of members.* All querying logic is expressed through member accesses. The members are statically known (generated by a type provider). When using the type provider in an editor, the user gets a choice of available members (auto-completion) when they type `"."` and they can thus construct a query simply by repeatedly choosing one of the offered members.
- *Lazy class generation.* The classes used in the code are generated lazily. This is necessary because each operation transforms the set of available fields based on which the subsequent types are generated. For example, calling `'drop Games'` would remove the field `Games` from the schema.
- *Safety of generated types.* Any query constructed using the type provider is correct meaning that it will not attempt to access a field that does not exist in the data. This is a variant of the usual type safety property that is formalized below.

The formalization of the type provider follows the same style as that for F# Data, but it explicitly encodes the laziness of the type provider as illustrated in the next section.

2.3.1 Formalising lazy type provider for data querying

The pivot type provider works on tabular data. In order to generate a type, it needs to have the schema of the input table (names of fields and their types). In the above example, the type provider is imported through a configuration rather than code, and `olympics` refers to a value of the provided type, but the type is generated using a known schema of the input data. In the formal model, the schema is written as (with f ranging over the field names and τ ranging over a small set of primitive types):

$$F = \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\}$$

When the type provider is invoked, it takes the schema and generates a class for querying data of the given schema. The types of members of the class are further classes that allow further querying. As the provided class structure is potentially infinite, it needs to be generated lazily. The structure of the provided class definition, written as L is thus a function mapping a class name C to a pair consisting of the class definition and a function that provides definitions of delayed classes (types used by the members of the class C):

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

Here, $\mathbf{type} C(x : \tau) = \bar{m}$ is a definition of a class C that consists of a sequence of members \bar{m} and has a constructor taking a variable x of type τ as an argument. The structure and evaluation of the resulting object calculus is discussed in Chapter 7 and is loosely modeled after standard object calculi (Igarashi et al., 2001; Abadi and Cardelli, 2012), with the exception that it includes operations for transforming data as primitives.

The classes provided by the pivot type provider can be used to construct a query, which is a value of type `Query`. Expressions of this type are those of relational algebra (projection, sorting, selection, as well as additional grouping). The type provider constructs classes that take the query constructed so far as the constructor argument. The provided members further refine and build the query. A type provider is formally defined as a function $\mathit{pivot}(F)$, which is similar to the function $\llbracket \sigma \rrbracket$ defined for the F# Data type providers:

$$\mathit{pivot}(F) = C, \{C \mapsto (\mathbf{type} C(x : \mathit{Query}) = \dots, L)\} \\ \text{where } F = \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\}$$

The full definition given in Chapter 7 uses a number of auxiliary functions to define the type provider, each of which defines members for specifying a particular query operation. To illustrate the approach, the following excerpt shows the $\mathit{drop}(F)$ function that is used to construct operations that let the user drop any of the columns currently in the schema F . The generated class has a member `'drop f'` for each of the fields and a member `then`, which can be used to complete the selection and return to the choice of other query operations. Each of the `drop` operations returns a class generated for the newly restricted domain and passes it a query that applies the selection Π operation of the relational algebra on the input data:

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

The formalization of the pivot type provider follows a similar style as that of the F# Data, although it differs in that it explicitly represents the laziness of the type generation and also in that the provided types construct more complex code, expressed using a variant of relational algebra, that is executed at runtime. The formalization serves to explain the functioning of the type provider, but also allows us to prove its safety.

2.3.2 Safety of data acquisition programs

The pivot type provider guarantees that the data transformations, which can be constructed using the types it generates will always be correct. They will never result in an undefined runtime behavior that one may otherwise encounter when accidentally accessing a non-existent field. This is an important result because the sequence of operations transforms the fields in interesting ways. Operations like `dropremove` fields from the schema, while `group by` changes the set of fields and their types (e.g., when we count distinct values of a string-typed field f in aggregation, the resulting dataset will contain a numerical field f).

To capture the property formally, we again state that any program written by the programmer using the type provider (without directly accessing the low-level operations of the relational algebra) will always reduce to a value. The evaluation is defined on datasets D which map fields to vectors of values, written as $D = \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,m} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,m} \rangle\}$. A specific kind of data value is a data series $\text{series}\langle\tau_k, \tau_v\rangle(D)$ that contains a vector of keys k and a vector of values v . The evaluation is defined as a reduction operation $e \rightsquigarrow_L^* e'$ which also has access to class definitions L . Similarly, the typing judgment $L_1; \Gamma \vdash e : \tau; L_2$ includes additional handling of lazily generated classes. It states that the expression e has a type τ in a variable context Γ . The typing is provided with (potentially unevaluated) class definitions L_1 . It accesses (and evaluates) some of those definitions and those that are used throughout the typing derivation are represented by L_2 .

Theorem 3 (Safety of pivot type provider). *Given a schema $F = \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\}$, let $C, L = \text{pivot}(F)$ then for any expression e that does not contain relational algebra operations or query-typed values as sub-expression, if $L; x : C \vdash e : \text{series}\langle\tau_1, \tau_2\rangle; L'$ then for all $D = \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,m} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,m} \rangle\}$ such that $\vdash v_{i,j} : \tau_i$ it holds that $e[x \leftarrow C(D)] \rightsquigarrow_{L'}^* \text{series}\langle\tau_k, \tau_v\rangle(\{f_k \mapsto k_1, \dots, k_r, f_v \mapsto v_1, \dots, v_r\})$ such that for all $j \vdash k_j : \tau_k$ and $\vdash v_j : \tau_v$.*

In other words, if a programmer uses the provided types to write a program e that evaluates to a data series and we provide the program with input data D that matches the schema used to invoke the type provider, the program will always evaluate to a data series containing values of the correct type. Although the property is not labeled as *relative type safety* as in the case of the F# Data type providers, it follows the same spirit. A well-typed program will not go wrong, as long as the input has the right structure.

2.4 Contributions

In this chapter, I offered a brief overview of the work on type providers that is included in Part II. The focus of this part is on simplifying programmatic data acquisition, that is on making it easier and safer to write code that reads data from external data sources. It consists of a type provider for semi-structured data in XML, JSON and CSV formats (Chapter 6) and a type provider that makes it possible to express queries over tabular data (Chapter 7).



Key contributions. The publications included in Part II include three main contributions. They introduce the novel notion of *relative type safety* for discussing correctness of programs that rely on external data, they present *type providers for structured data formats* and they a type provider for *querying relational databases* that guarantees relative type safety of the resulting program.

Both of the contributions consist of a practical implementation, as a library for the F# language and as a component of the web-based programming environment The Gamma, respectively. They combine this with a theoretical analysis using the methodology of theoretical programming language research. This makes it possible to precisely capture subtle aspects of how the type providers work (including shape inference, laziness, and generation of types for query construction), but also to capture safety guarantees of the generated types. Given that type providers always access external data, the guarantees are not absolute as in conventional programming language theory. For this reason, my work introduced a novel notion of *relative type safety*, stating that programs will “not go wrong” as long as the input has the correct structure (in a precisely defined sense).

From a broader perspective, the two type providers can be seen as filling a glaring gap in the theoretical work of statically-typed programs. A theoretician who defines a type system always uses a top-level typing rule $\vdash e : \tau$ stating that a program e (closed expression) that does not use any variables has a type τ . While at the top-level, programs may not use any variables, this is misleading because most real-world programs access the outside world in some way, but this is typically done in an unchecked way. Monads and effect systems (Lucassen and Gifford, 1988; Peyton Jones and Wadler, 1993) can be used to track that some external access is made, but they do not help the static type system understand the structure of the outside data. With slight notational creativity, we can say that the static type checking of a program that uses type providers starts with a rule $\pi(\oplus) \vdash e : \tau$ where \oplus (used as the astronomical symbol for the Earth) refers to the entire outside world and π refers to some projection from all the things that exist in the outside world to program variables with static types that a programming language understands.

The two kinds of type providers discussed in this chapter also differ in how they approach the technology gap suggested in Figure 1.1. The F# Data type providers aim to make programming with external data in a statically typed programming language a bit easier. In other words, they extend the area that can be covered by conventional programming, including more users and reducing the complexity. The pivot type provider and The Gamma programming environment tries to fill a particular space within the gap. It lets a relatively large number of users (who are not professional programmers) solve problems that are more complex than simple data wrangling in a spreadsheet system, but much less complex than using a conventional programming tool such as Python and pandas. Its usability is a topic I will revisit in Chapter 4 and the paper included as Chapter 10.

Chapter 3

Data infrastructure

Data scientists use a wide range of tools when working with data. A large part of what makes data cleaning and data exploration challenging is that data scientists often need to switch from one tool to another (Rattenbury et al., 2017). They may use an interactive online tool like Trifacta to do data cleanup, run an ad-hoc command-line tool to transform it, and then import it into a Jupyter notebook to create a visualization. Moreover, data science is an interactive and iterative process. Data scientist need to be able to quickly review the results of the operation they performed in order to see whether the results match their expectations and to detect unexpected problems. The interactivity brings a further challenge, which is the reproducibility of results. If the data scientist quickly tries multiple different approaches, and reverts some of their earlier experiments, they should always be able to know what exact steps led to the final result they see on their screen.

In this chapter, I provide an overview of two contributions to the infrastructure for doing data exploration. The work addresses the three requirements that arise from the typical way data exploration is done as outlined above:

- *Polyglot tooling support.* Data scientist need an easy way of integrating multiple different tools. For example, they should be able to use simple data acquisition tools, such as the pivot type provider implemented in The Gamma, but then pass the data to Python for further processing or to a visual interactive tool.
- *Live preview support.* In order to let data scientists quickly review the results of the operations they perform, the infrastructure should provide immediate live previews without unnecessary recomputation.
- *Reproducibility and correctness.* The results that the data scientist sees on the screen should always match with the code (or reproducible another trace) they have in their data exploration environment. If the operations involved are deterministic, re-running them should produce the same result.

Although each of those challenges has a range of solutions, there are not many systems that address all of them. This chapter provides an overview of work leading towards such a system. It consists of two parts. The first is a data exploration environment for The Gamma that introduces an efficient way of evaluating live previews (presented in full in Chapter 8) using a method based on maintaining a dependency graph. The second part is a notebook system for data science called Wrattler (presented in full in Chapter 9) that follows the same basic approach, but allows integration of multiple languages and tools and also uses the dependency graph to ensure reproducibility of the data explorations.

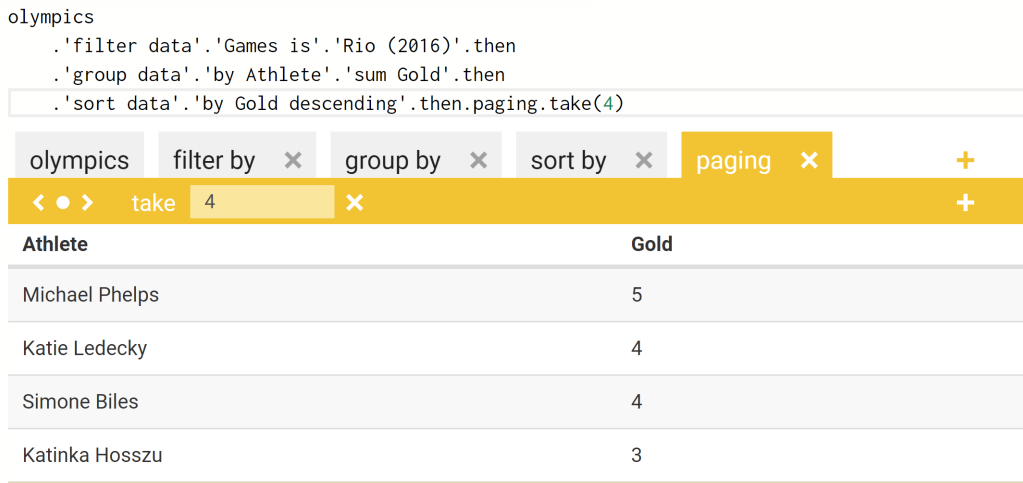


Figure 3.1: A live preview in The Gamma, generated for a code snippet that uses the pivot type provider for data exploration. The interface also lets the user navigate through the steps of the transformation and modify parameters of the query.

Methodologically, the work outlined in this chapter combines the programming systems research methods with programming language theory. Both of the systems are available as open-source projects and they have been evaluated through a range of realistic case studies. The publication on live previews for data exploration environments presents a formal model to explain how live previews are computed using a dependency graph and to show the correctness of this approach, but it also includes a performance evaluation. The main contribution of the work on Wrattler is the novel architecture of the system.

3.1 Notebooks and live programming

As noted above, all three of the challenges have been addressed in isolation. The integration of different tools has been addressed in the context of scientific workflow systems such as Taverna (Oinn et al., 2004) and Kepler (Altintas et al., 2004) that orchestrate complex scientific pipelines, but such tooling is too heavyweight for basic data exploration done for example by data journalists. Scientific workflow systems also tackle the problem of reproducibility as the workflows capture the entire data processing pipeline.

In the context of programming tools, work on live environments that provide immediate feedback and help the programmer better understand relationships between the program code and its outputs have been inspired by the work of Victor (2012b,a). A comprehensive review by Rein et al. (2019) includes programming tools and systems that provide immediate feedback ranging from those for UI development and image processing to live coding tools for music. A chief difficulty with providing live feedback as code is modified lies in identifying what has been changed. This can be done by using a structure editor that keeps track of code edits (Omar et al., 2019). The approach presented below aims to support ordinary text-based editing and is based on the idea of reconstructing a dependency graph from the code.

Finally, the issue of reproducibility has received much attention in the context of notebooks for data science such as Jupyter (Kluyver et al., 2016). Although Jupyter can be used to produce reproducible notebooks, there are practical barriers to this. In particular, it allows execution of cells out-of-order, meaning that one can run code in a way that modifies the global state in an unexpected and non-reproducible way. This has been addressed in multiple systems (Pimentel et al., 2015; Koop and Patel, 2017) and our approach in Wrattler builds on this tradition.

3.2 Live data exploration environment

The Gamma explores a particular point in the design space of data exploration tools. It is built around code written in a simple programming language, leveraging the type provider introduced in Section 2.3. This focus on code makes it easier to guarantee reproducibility and transparency of data analyses. At the same time, the design raises the question of how easy can data exploration be when done through a text-based programmatic environment. I revisit this problem from the human-computer interaction perspective in the next chapter, after discussing the infrastructure that makes using The Gamma easier.

One of the lessons learned from spreadsheets is the value of immediate or live feedback. To make data exploration in The Gamma easier, the work outlined in this section develops an efficient method for displaying live previews for The Gamma as illustrated in Figure 3.1. However, providing live previews in a text-based programming environment is a challenge (McDermid, 2007). There are two difficulties:

- *Live previews and abstractions.* It is difficult to provide live previews for code inside functions or classes because variables in such context cannot be easily linked to concrete values. Even if such abstractions are not used as frequently in data exploration code, abstractions are often the key concern in conventional theoretical thinking about programming language design.
- *Responding to code changes.* Code in a text editor can change in arbitrary ways and so it is unclear how to update the existing live preview when an edit is made. This is easier in structure editors where edits are limited and understood by the system, but live previews for a text-based system need to accommodate large and potentially breaking changes in code.

In the work included as Chapter 8, I tackle the first challenge by arguing that we need a better theoretical model of programming languages for data exploration. When data scientist explore data in a notebook environment, they typically do not introduce new abstractions and most code is first-order. They often use external libraries, some of which provide higher-order functions (projection, filtering, etc.) and so code may use functions and lambda expressions, but those are typically passed directly as arguments to those functions. My work thus introduces the *data exploration calculus*, which is a small formal model of a programming language that corresponds closely to code written to explore data and can be used to formally study problems in programmatic data exploration tools.

The problem of responding to code changes is tackled by constructing a dependency graph and caching its nodes. When the code is edited, the new version is parsed, resulting in a new abstract syntax tree. The nodes of the tree are then analyzed and linked to nodes in a dependency graph. When the node of the tree corresponds to a dependency graph node that has been created previously (with the same dependencies), the graph node is

Programs, commands, terms, expressions, and values

$$\begin{array}{lll}
 p ::= c_1; \dots; c_n & t ::= o & e ::= t \mid \lambda x \rightarrow e \\
 c ::= t & \mid x & v ::= o \mid \lambda x \rightarrow e \\
 \mid \mathbf{let} \ x = t \ t & \mid t.m(e, \dots, e) &
 \end{array}$$

Evaluation contexts of expressions

$$\begin{array}{l}
 C_e[-] = C_e[-].m(e_1, \dots, e_n) \mid o.m(v_1, \dots, v_m, C_e[-], e_1, \dots, e_n) \mid - \\
 C_c[-] = \mathbf{let} \ x = C_e[-] \mid C_e[-] \\
 C_p[-] = o_1; \dots; o_k; C_c[-]; c_1; \dots; c_n
 \end{array}$$

Let elimination and member reduction

$$\begin{array}{l}
 o_1; \dots; o_k; \mathbf{let} \ x = o; c_1; \dots; c_n \rightsquigarrow \\
 o_1; \dots; o_k; o; c_1[x \leftarrow o]; \dots; c_n[x \leftarrow o] \quad (\mathbf{let}) \\
 \\
 \frac{o.m(v_1, \dots, v_n) \rightsquigarrow_\epsilon o'}{C_p[o.m(v_1, \dots, v_n)] \rightsquigarrow C_p[o']} \quad (\mathbf{external})
 \end{array}$$

Figure 3.2: Syntax, contexts and reduction rules of the data exploration calculus

reused. Live previews are then computed (and associated with) dependency graph nodes. As a result, when dependencies of a particular expression do not change, it is linked to the same graph node as before and the associated live preview is reused.

In the following two sections, I provide a brief review of the data exploration calculus and of the dependency graph construction mechanism. In Chapter 8, the data exploration calculus is then used to formalize the graph construction and show that live previews computed based on the graph are the same as previews that would be computed by directly evaluating the data exploration calculus expression. The publication also evaluates the efficiency using live previews, quantifying the reduction in overhead in contrast to two other evaluation strategies.

3.2.1 Data exploration calculus

The data exploration calculus is a small formal language for data exploration. The calculus is intended as a small realistic model of how are programming languages used in data exploration scripts and computational notebooks. The calculus itself is not Turing-complete and models first-order code only, but it supports the notion of external libraries that provide specific data exploration functionality. This may include standard functions for working with collections or data frames that are common in Python, but also libraries based on type providers as in the case of The Gamma.

Figure 3.2 shows the syntax of the calculus. A program p consists of a sequence of commands c . A command can be either a let binding or a term. Let bindings define variables x that can be used in subsequent commands. As noted earlier, lambda functions can only appear as arguments in method calls. To model this, the calculus distinguishes between terms that can appear at the top-level and expressions that can appear as arguments in an invocation. A term t can be a value, variable, or a member access, while an expression e can be a lambda function or a term. Values defined by external libraries are written as o .

The evaluation is defined by a small-step reduction \rightsquigarrow . Fully evaluating a program results in an irreducible sequence of objects $o_1; \dots; o_n$ (one object for each command, including let bindings) which can be displayed as intermediate results of the data analysis. The operational semantics is parameterized by a relation $\rightsquigarrow_\epsilon$ that models the functionality of external libraries. Figure 3.2 defines the reduction rules in terms of $\rightsquigarrow_\epsilon$ and evaluation contexts; C_e specifies left-to-right evaluation of arguments of a method call, C_c specifies evaluation of a command and C_p defines left-to-right evaluation of a program. The rule (external) calls a method provided by an external library in a call-by-value fashion, while (let) substitutes a value of an evaluated variable in all subsequent commands and leaves the result in the list of commands.

Note that our semantics does not define how λ applications are reduced. This is done by external libraries, which will typically supply functions with arguments using standard β -reduction. The result of evaluating an external call is also required to be an object value o . To illustrate how a definition of an external library looks, consider the following script:

```
let l = list.range(0, 10)
    l.map( $\lambda x \rightarrow$  math.mul(x, 10))
```

An external library provides the `list` and `math` objects, as well as numbers n , lists of objects $[o_1, \dots, o_k]$, and failed computations \perp . Next, the external library needs to define the semantics of the `range`, `mul`, and `map` members through the $\rightsquigarrow_\epsilon$ relation. The following shows the rules for the `map` operation on lists:

$$\frac{e[x \leftarrow n_i] \rightsquigarrow o_i \quad (\text{for all } i \in 1 \dots k)}{[n_1, \dots, n_k].\text{map}(\lambda x \rightarrow e) \rightsquigarrow_\epsilon [o_1, \dots, o_k]} \quad \frac{(\text{otherwise})}{[n_1, \dots, n_k].m(v_1, \dots, v_n) \rightsquigarrow_\epsilon \perp}$$

When evaluating `map`, we apply the provided function to all elements of the list using standard β -reduction and return a list of resulting objects. The $\rightsquigarrow_\epsilon$ relation is defined on all member accesses, but non-existent members reduce to the failed computation \perp .

We require that external libraries satisfy two conditions. First, when a method is called with observationally equivalent values as arguments, it should return the same value (compositionality). Second, the evaluation of $o.m(v_1, \dots, v_n)$ should be defined for all o, n and v_i (totality). The above definition satisfies those requirements by using the standard β -reduction for reducing lambda functions and by reducing all invalid calls to the \perp object. Compositionality implies the deterministic behavior of external libraries and is essential for implementing an efficient live preview mechanism. The totality of the definition, in turn, makes it possible to prove the following *normalization* property:

Theorem 4 (Normalization). *For all p , there exists n, o_1, \dots, o_n such that $p \rightsquigarrow^* o_1; \dots; o_n$ where \rightsquigarrow^* is the reflexive, transitive closure of \rightsquigarrow .*

The value of the data exploration calculus is that it can be used to model the functionality of different tools that support data exploration. The work outlined here (and presented in full in Chapter 8) uses the calculus to formalize an efficient mechanism for showing live previews during the editing of data exploration script. As mentioned earlier, the mechanism works by constructing a dependency graph, binding expressions to the graph and associating live previews with the (cached) nodes of the graph. The formal properties of the data exploration calculus make it possible to prove that live previews computed in this way are the same as previews that would be obtained by fully re-evaluating the data exploration script.

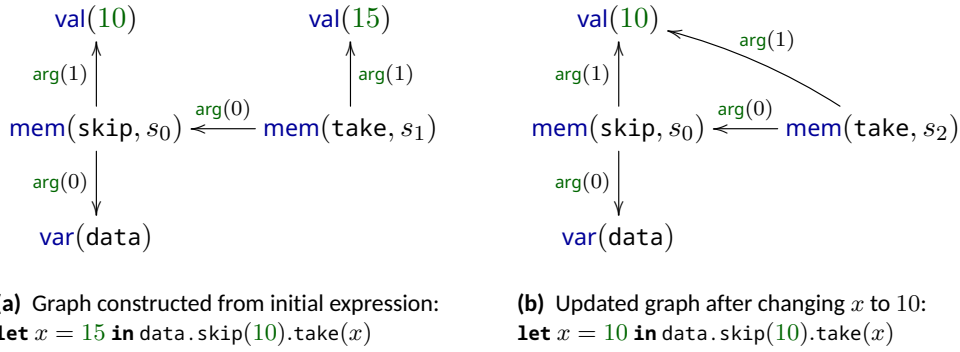


Figure 3.3: Dependency graphs formed by two steps of the live programming process.

3.2.2 Computing previews using a dependency graph

Given a program in the data exploration calculus, I now describe the core of a mechanism that can be used for providing the user with live previews as illustrated in Figure 3.1. The key idea behind our method is to maintain a dependency graph with nodes representing individual operations of the computation that can be evaluated to obtain a preview. Each time the program text is modified, we parse it afresh (using an error-recovering parser) and bind the abstract syntax tree to the dependency graph. When binding a new expression to the graph, we reuse previously created nodes as long as they have the same structure and the same dependencies. For expressions that have a new structure, we create new nodes.

The nodes of the graph serve as unique keys into a lookup table containing previously evaluated parts of the program. When a preview is requested for an expression, we use the graph node bound to the expression to find a preview. If a preview has not been evaluated, we force the evaluation of all dependencies in the graph and then evaluate the operation represented by the current node.

The nodes of the graph represent individual operations of the computation. A node indicates what kind of operation the computation performs and is linked to its dependencies through edges. This makes it possible to define computation not just over expressions of the data exploration calculus, but also over the dependency graph. In order to cache computed previews with the node as the key, some of the nodes need to be annotated with a unique symbol. That way, we can create two unique nodes representing, for example, access to a member named `take` which differ in their dependencies. The graph edges are labeled with labels indicating the kind of dependency. For a method call, the labels are “first argument”, “second argument” and so on. Writing s for symbols and i for integers, nodes (vertices) v and edge labels l are defined as:

$$\begin{array}{ll}
 v & = \text{val}(o) \mid \text{var}(x) \mid \text{mem}(m, s) \mid \text{fun}(x, s) & (\text{Vertices}) \\
 l & = \text{body} \mid \text{arg}(i) & (\text{Edge labels})
 \end{array}$$

The `val` node represents a primitive value and contains the object itself. Multiple occurrences of the same value, such as `10`, will be represented by the same node. Member access `mem` contains the member name, together with a unique symbol s – two member access nodes with different dependencies will contain a different symbol. Dependencies of a member access are labeled with `arg` indicating the index of the argument (0 for the instance and 1, 2, 3, . . . for the arguments). Finally, nodes `fun` and `var` represent function values and variables bound by λ abstraction.

Figure 3.3 illustrates how to build the dependency graph. Node representing $\text{take}(x)$ depends on the argument – the number 15 – and the instance, which is a node representing $\text{skip}(10)$. This, in turn, depends on the instance data and the number 10. Note that variables bound via **let** binding such as x do not appear as **var** nodes. The node using it depends directly on the node representing the expression assigned to x .

After changing the value of x , we create a new graph. The dependencies of the node $\text{mem}(\text{skip}, s_0)$ are unchanged. The symbol s_0 attached to the node remains the same and so the previously computed previews can be reused. This part of the program is not recomputed. The $\text{arg}(1)$ dependency of the take call changed and so we create a new node $\text{mem}(\text{skip}, s_2)$ with a fresh symbol s_2 . The preview for this node is then computed as needed using the already-known values of its dependencies.

The full description of how the dependency graph is constructed can be found in Chapter 8. The construction proceeds recursively over the syntactic structure of the program in the data exploration calculus. For each expression in the program, it recursively obtains graph nodes representing its sub-expressions. It then checks the cache to see if a node representing the current expression with the same dependencies exists already. If so, the node is reused. If no, a new node (possibly with a new symbol) is created.

The construction of the graph makes it possible to compute previews over the nodes of the dependency graph and cache the previously computed previews by using the graph node as the cache key. I illustrate how the evaluation works using two of the reduction rules. For simplicity, I do not discuss the caching here. I will also write p for evaluated previews which can be either primitive objects o or functions $\lambda x.e$ (for which we cannot show a preview directly). Given a dependency graph (V, E) where V is a set of vertices v_1, v_2, \dots, v_n and E is a set of directed labelled edges of the form (v_1, v_2, l) , the evaluation is then defined as a relation $v \Downarrow p$. The following two rules illustrate evaluation for primitive values and for member access:

$$\frac{}{\text{val}(o) \Downarrow o} \text{ (val)}$$

$$\frac{\forall i \in \{0 \dots k\}. (\text{mem}(m, s), v_i, \text{arg}(i)) \in E \quad v_i \Downarrow p_i \quad p_0.m(p_1, \dots, p_k) \rightsquigarrow_{\epsilon} p}{\text{mem}(m, s) \Downarrow p} \text{ (mem-val)}$$

The (val) rule is simple. If a graph node represents a primitive value, it directly reduces to the value. The (mem-val) rule illustrates a more interesting case. To evaluate a member access, we need to find the graph nodes that represent its arguments (by looking for links with an appropriate label), reduce those recursively, and then use the external library reduction $\rightsquigarrow_{\epsilon}$ to reduce the member access.

The sketch presented here omits one interesting aspect of the mechanism. In general, previews can be provided for all sub-expressions that include variables defined by an earlier **let** binding. However, if a sub-expression contains a variable bound by a lambda expression, we have no way of obtaining a suitable value for the variable. In this case, our mechanism evaluates a delayed preview $\llbracket e \rrbracket_{\Gamma}$, which represents a partially-evaluated expression that depends on variables specified by Γ . Delayed previews could still be useful if the user interface allowed the user to specify sample value for the free variables and they also have an interesting theoretical connection to work on Contextual Modal Type Theory (Nanevski et al., 2008) and comonads (Gabbay and Nanevski, 2013).

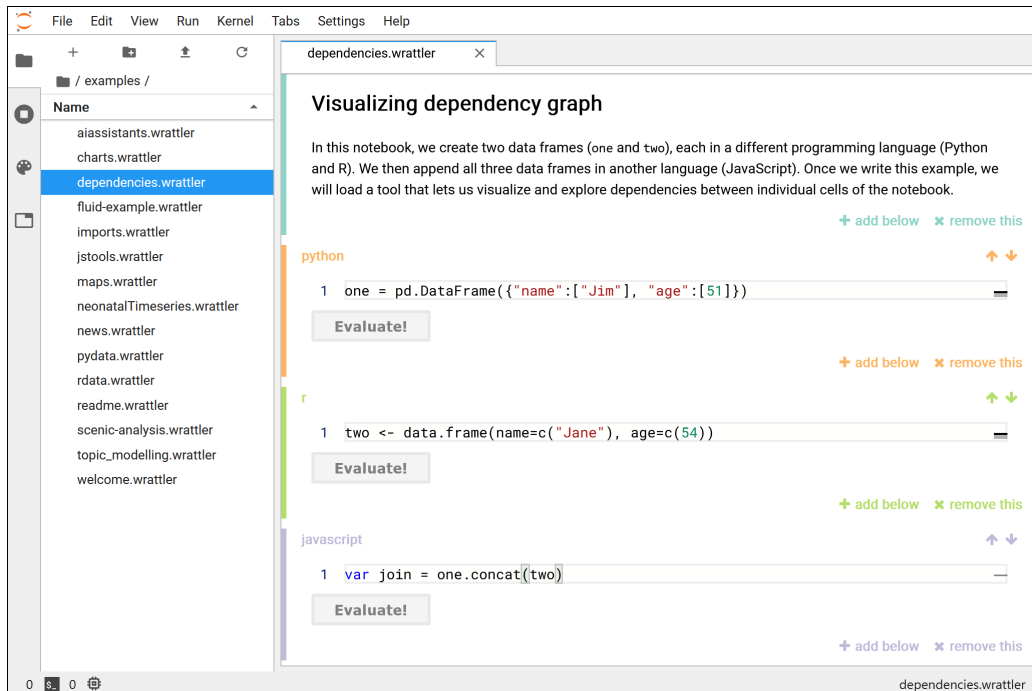


Figure 3.4: Wrattler running inside the JupyterLab system. The opened notebook passes data between cells written in three different programming languages (Python, R and JavaScript).

The full paper, included as Chapter 8, uses two research methodologies to evaluate the work. First, it formalizes how the live preview mechanism works using the model based on the data exploration calculus as sketched above. The formalization is used to show that the previews computed over the dependency graph are *correct*. That is, they are the same as the values we would obtain by evaluating the data exploration calculus expressions directly. The formalization is also used to list a number of common edits to a program that do not invalidate previously computed live previews. Examples of such edits include extracting sub-expression into a let-bound variable, deleting or adding unused code, or changing unrelated parts of the program. The evaluation also employs programming systems research methods to empirically evaluate the efficiency of the live preview evaluation method. The paper contrasts the method with standard call-by-value and lazy evaluation strategies (without caching) and shows the reduction of delays in providing live previews for a sample coding scenario.

3.3 Live, reproducible, polyglot notebooks

The live data exploration environment discussed in the previous section tackles the problem of providing rapid feedback to data scientists during data exploration. The other two challenges that I listed in the opening of this chapter were the need for polyglot tooling support and the need to make data analyses more reproducible.

The two challenges are addressed by the open-source Wrattler notebook system presented in full in Chapter 9. Wrattler is an extension of the industry standard JupyterLab platform. As illustrated in Figure 3.4, Wrattler adds a new type of document format that

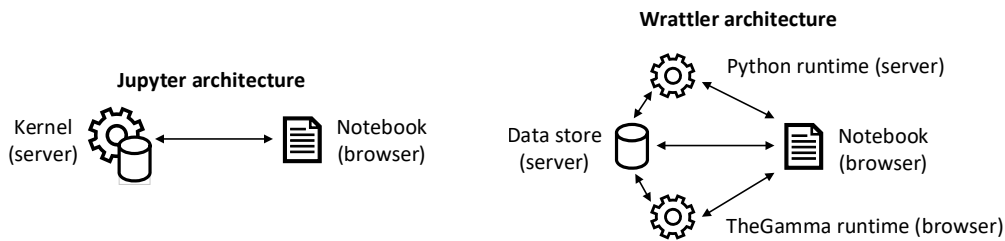


Figure 3.5: In notebook systems such as Jupyter, state and execution are managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Language runtimes can run on the server-side (e.g. Python) or client-side (e.g. The Gamma).

allows programmers to mix cells written in multiple different programming languages in a single notebook. The extensibility model of Wrattler makes it possible to support not only new programming languages but also interactive tools that run directly in the notebook (hosted in a web browser). As a result, it is possible to integrate tools that provide a live preview mechanism such as The Gamma and also interactive AI assistants that I discuss in Part IV. The architecture of the Wrattler system is based on two key principles:

- *Polyglot architecture.* The system is designed to allow the integration of components in different programming languages. This is done by splitting the monolithic architecture of Jupyter into individual components including the central data store and multiple language runtimes.
- *Design for reproducibility.* To guarantee reproducibility and track data provenance, the system represents computation as a dependency graph. The graph is similar to the one discussed in the previous section but uses a coarser granularity with one node for each notebook cell.

The Wrattler system is presented in detail in Chapter 9. The paper follows the programming systems methodology. It focuses on the novel system architecture and documents the capabilities that are enabled by the architecture.

3.3.1 Architecture of a novel notebook system

Standard notebook architecture consists of a *notebook* and a *kernel*. The kernel runs on a server, evaluates code snippets, and maintains the state they use. The notebook runs in a browser and sends commands to the kernel in order to evaluate cells selected by the user. As illustrated in Figure 3.5, Wrattler splits the server functionality into two components:

- *Data store.* Imported external data and results of running scripts are stored in the data store. The data store keeps version history and annotates data with metadata such as types, inferred semantics, and provenance information.
- *Language runtimes.* Code in notebook cells is evaluated by language runtimes. The runtimes read input data from and write results back to the data store. Wrattler supports language runtimes that run code on the server (similar to Jupyter) but also browser-based language runtimes.

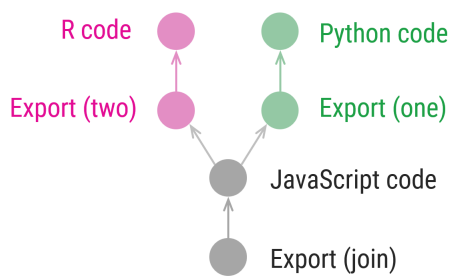


Figure 3.6: Dependency graph of a notebook from Figure 3.4. For each cell, the graph contains a code node and one (or possibly more) export nodes that represent exported data frames. The R and Python cells are independent and map to independent graph nodes. The node corresponding to the final JavaScript cell depends on nodes representing the two variables used in the code.

- *Notebook.* The notebook is displayed in a web browser and orchestrates all other components. The browser builds a dependency graph between cells or individual calls. It invokes language runtimes to evaluate code that has changed and reads data from the data store to display results.

The central component of the system is the data store, which enables communication between individual Wrattler components and provides persistent data storage. Data frames stored in the data store are associated with a hash of a node in a dependency graph constructed from the code in the notebook (using a mechanism discussed below) and are immutable. When the notebook changes, new nodes with new hashes are created and appended to the data store. This means that language runtimes can cache data and avoid fetching them from the data store each time they need to evaluate a code snippet.

External inputs imported into Wrattler notebooks (such as downloaded web pages) are stored as binary blobs. Data frames are stored in either JSON or binary format. The data store also supports a mechanism for annotating data frames with semantic information. Columns can be annotated with primitive data types (date, floating-point number) and semantic annotation indicating their meaning (address or longitude and latitude). Columns, rows, and individual cells of the data frame can also be annotated with custom metadata such as their data source or accuracy.

3.3.2 Dependency graphs for notebooks

At runtime, Wrattler maintains a dependency graph that is remarkably similar to the one used in the live data exploration environment for The Gamma discussed in Section 3.2. As before, the dependency graph is used to cache the results of previous computations. The nodes in the graph have a unique identifier (hash) that is used as the key for caching data in the data store. When code in the notebook is modified, the graph is re-created, reusing previously created nodes where possible.

An example of a dependency graph is shown in Figure 3.6. For every type of cell, Wrattler needs to be able to identify the names of imported and exported variables. In the case of Python, R, and JavaScript, this is done using a lightweight code analysis. In the case of The Gamma, which can also be used in Wrattler, the full parse tree and its associated dependency graph are available. A prototype extension of Wrattler embeds The Gamma graph as a sub-graph of the dependency graph maintained by Wrattler.

An important design choice in the Wrattler design is that cells can only share data in the form of a data frame. The trade-offs of this choice remain to be evaluated. On the one hand, it means that Wrattler fits only certain data analytical scenarios. On the other hand, it makes it possible to easily share data between cells in different languages. In the

example dependency graph, each of the “export” nodes thus corresponds to a data frame that is stored in the data store (using the unique hash of the graph node as the key).

The dependency graph is updated after every code change. This is done using the same mechanism as in the live data exploration environment discussed in Section 3.2. Wrattler invokes individual language runtimes to parse each cell. It then walks over the resulting structure and constructs nodes for each cell or exported variable with edges indicating dependencies. The hash for each node is computed from the data in the node (typically source code or variable name) and the hashes of nodes it depends on. An important property of this process is that, if there is no change in dependencies of a node, the hash of the node will be the same as before. As a result, previously evaluated values attached to nodes in the graph are reused.

When the evaluation of an unevaluated cell is requested, Wrattler recursively evaluates all the nodes that the cell depends on and then evaluates the values exported by the cell. The evaluation is delegated to a language runtime associated with the language of the node. For languages that run on the server-side (Python, R), the language runtime sends the source code, together with its dependencies, to a server that evaluates the code. Note that the request needs to include only hashes of imported variables as the server can obtain those directly from the data store. For nodes that run on the client-side (JavaScript, The Gamma), the evaluation is done directly in the web browser.

3.4 Contributions



Key contributions. The publications included in Part III include three main contributions. They capture the essence of data scripting in the form of *data exploration calculus*, they present the architecture for *polyglot, live and reproducible notebook systems* and they describe an efficient algorithm for *live preview recomputation* based on the construction of a dependency graph.

In this chapter, I outlined two contributions to the data analytics infrastructure that are included in Part III of this thesis. The two contributions describe systems that aim to make data exploration more live and reproducible while supporting the polyglot reality of data processing tools used today.

The work included in Chapter 8 focuses on providing live previews during data exploration. Can we simplify data exploration by efficiently previewing the result of a data transformation while the data analyst is constructing it and tweaking its parameters? The mechanism presented in this thesis provides a possible answer. The work follows primarily the programming language research methodology and so it attempts to capture the core idea behind the approach, using the simple (but adequate) formal model of the data exploration calculus. The implementation of the idea provides live previews for code written in The Gamma, a simple programming language with support for type providers that we encountered already in Section 2.3 and that I will return to once more in the next chapter, but using the perspective of human-computer interaction research.

The work included in Chapter 9 presents a polyglot notebook system Wrattler. The system makes it possible to mix multiple tools in a single notebook. This includes existing programmatic tools, such as those based on Python and R, as well as novel tools like

The Gamma. The sharing is enabled by the design choice of allowing only data frames as the exchange format between cells. The promise of the Wrattler architecture is to enable more research and innovation in the data exploration tooling space. It enables data analysts to use tools they are already familiar with, but use novel tools where appropriate - for example, include a cell in The Gamma that will let consumers of their notebooks explore aggregate data without advanced programming expertise. We will leverage this architecture again in the work on AI assistants (Chapter 11), outlined in the next chapter.

One interesting point that is revealed by putting the two contributions side-by-side is that they both rely on the same implementation technique. They both maintain a dependency graph of code (expressions or cells) and update it as the code is edited. The graph is constructed so that code that remains the same is bound to the same node, making it possible to reuse previously computed results. The technical similarity is rooted in a broader principle. In both cases, the reproducible code is the final trace that produces all relevant outputs. The principle is in contrast with an alternative where code is executed interactively to modify some state as in systems based on Read-Eval-Print Loop (REPLs).

Chapter 4

Iterative prompting

Data wrangling is the tedious process of getting data into the right format for data exploration. It involves parsing data, joining multiple datasets, correcting errors, and recovering semantic information. According to domain experts (Rattenbury et al., 2017), data wrangling takes up to 50-80% of data scientist's time. Unfortunately, there is no easy cure to the problem of data wrangling. The reason for the difficulty is what van den Burg et al. (2019) refer to as the *double Anna Karenina principle*: "every messy dataset is messy in its own way, and every clean dataset is also clean in its own way." In other words, there is no single characterization of a clean dataset that tools could optimize for. Human insight into the data is always needed.

Different research directions approached the problem of data wrangling from different perspectives. Graphical end-user programming tools typically make it easy to complete the most common tasks for the most common kinds of datasets but are incapable of covering the inevitable special cases that are present due to the double Anna Karenina principle. Automatic AI-based tools for data wrangling suffer from the same issue. They work well in a large number of cases, but they can easily confuse interesting outliers for uninteresting noise in cases where a human would immediately spot the difference. This is perhaps why most data wrangling is often done manually and often involves a mix of programmatic and end-user tools. We can make those tools easier to integrate and make tweaking of parameters easier through live previews (as discussed in the previous chapter), but what if we could offer a different way of working with them?

The contributions outlined in this chapter are centered around the question of how to easily enable human data analysts, even if they are not expert programmers, to supply the necessary human insight to programmatic tools when cleaning and analyzing data. The answer presented in the first contribution (Chapter 10) is an interaction principle that I refer to as *iterative prompting*. In a tool that follows the principle, the user is repeatedly asked to choose from a list of offered options. The principle turns the familiar code completion mechanism from a programmer assistance tool into a non-expert programming mechanism. The two contributions included as Part IV use iterative prompting in two ways:

- In Chapter 10, the mechanism is used to allow non-programmers to construct data exploration scripts that query data from a range of different data sources. A key characteristic of the method is that the mechanism allows users to construct only correct scripts and all scripts expressible in the language can be constructed, i.e., the principle is *correct and complete*.

- In Chapter 11, the mechanism is used to guide four different semi-automatic AI data wrangling tools. Here, the tools run automatically, but the user can use iterative prompting to specify constraints in order to correct errors and oversights in the automatically generated solutions. In other words, iterative prompting provides a unified interface through which the analyst can supply human insights to the AI tool.

The primary contribution of the work presented in this chapter is that it develops and validates novel approaches to the problem of data wrangling. To do this, it uses two primary research methods. The work introducing iterative prompting (Chapter 10) is rooted in human-computer interaction research. It motivates the interaction principle, describes a prototype implementation, and shows its effectiveness through a qualitative case study and an empirical user study. The work on AI assistants (Chapter 11) combines programming language theory and programming systems research methods. It describes the architecture of the system using a formal model and validates it by making four existing automatic AI tools interactive and semi-automatic. The novel tools are evaluated empirically. In cases where the fully automatic tool fails, our semi-automatic tool allows the user to correct the solution with a small number (typically 1-2) of simple interactions.

The work in this chapter is best seen as design space exploration. I believe that programming languages and systems provide the right starting point for tackling the problem of data wrangling and data exploration. But in order to fulfill this role, they need to be significantly easier to use. Non-programmers need to be able to create simple data exploration scripts and data analysts need an easy-to-use interface for solving typical problems. Iterative prompting takes the basic auto-completion mechanism leveraged by type providers to a new level, turning it into a simple but powerful unifying interaction principle.

4.1 Data wrangling and data analytics

Data wrangling is most often done manually using a combination of programmatic and graphical tools. Jupyter and RStudio are popular environments used for programmatic data cleaning. They are used alongside libraries that implement specific functionality such as parsing CSV files or merging datasets [van den Burg et al. \(2019\)](#); [Sutton et al. \(2018\)](#) and general data transformation functions provided, e.g., by Pandas and Tidyverse.¹

Graphical data wrangling systems such as Trifacta² consist of myriad tools for importing and transforming data, which are accessible through different user interfaces or through a scriptable programmatic interface. Finally, spreadsheet applications such as Excel and business intelligence tools like Tableau are often used for manual data editing, reshaping, and especially visualization ([Kandel et al., 2011](#)). The above general-purpose systems are frequently complemented by ad-hoc, for example for parsing PDF documents.

Some of the most practical tools along the entire data wrangling pipeline partially automate a specific tedious data wrangling task. To merge datasets, Trifacta and datadiff ([Sutton et al., 2018](#)) find corresponding columns using machine learning. To transform textual data and tables, Excel employs programming-by-example to parse semistructured data and many tools exist to semi-automatically detect duplicate records in databases.

¹<https://pandas.pydata.org> and <https://www.tidyverse.org> (Accessed 12 June 2024)

²<https://www.trifacta.com> (Accessed 12 June 2024)

Interactive and semi-automatic data wrangling tools, allow the analyst to review the current state of the analysis and make changes to it. The interaction between a human and a computer in such data wrangling systems follows a number of common patterns:

- *Onetime interaction*. A tool makes a best guess but allows the analyst to manually edit the proposed data transformation. Examples include dataset merging in Trifacta and datadiff (Sutton et al., 2018).
- *Live previews*. Environments like Jupyter, Trifacta, and The Gamma (Chapter 8) provide live previews, allowing the analyst to check the results and tweak parameters of the operation they are performing before moving on.
- *Iterative*. A tool re-runs inference after each interaction with a human to refine the result. For example, in Predictive Interaction (Heer et al., 2015) the analyst repeatedly selects examples to construct a data transformation.
- *Question-based*. A system repeatedly asks the human questions about data and uses the answers to infer and refine a general data model. Examples include data repair tools such as UGuide (Thirumuruganathan et al., 2017).

For interactive data wrangling tools, the *live previews* pattern is the most common one with a varying degree of liveness. Most semi-automatic data wrangling tools accept only limited forms of human input. The *onetime interaction* pattern is the most common and only a few systems follow the more flexible *iterative* pattern. The iterative prompting principle that I introduce in this chapter implements the *iterative* pattern in a uniform way that is inspired by work on information-rich programming programming (Syme et al., 2013) and type providers (Chapter 2). It is centered around code but reduces the conceptual complexity of coding to a single basic kind of interaction.

4.2 Iterative prompting

Technically speaking, I have already discussed all the components that together make up the first implementation discussed in this chapter. In The Gamma, the iterative prompting principle is implemented through the standard code completion mechanism that is used to select members generated by the type provider outlined in Chapter 2. The main contribution of the paper included as Chapter 10 is that it looks at the design from the perspective of human-computer interaction research.

The key idea behind the principle is that a non-programmer should be able to construct an entire data exploration script only by selecting appropriate members from a list of offered choices. Technically speaking, the script thus becomes a single chain of member accesses. As I discuss below, this also requires a specific type provider design.

The process of data exploration through iterative prompting is illustrated in Figure Figure 4.1, which uses the type provider outlined in Chapter 2 to find the UK House of Lords member from the county of Kent with the most number of days away. The example shows three steps of the process:

1. The user starts by selecting an input data source (not using iterative prompting) and types `.` (dot) to see available querying operations. The system offers a list of (all available) operations including filtering, grouping, and sorting.

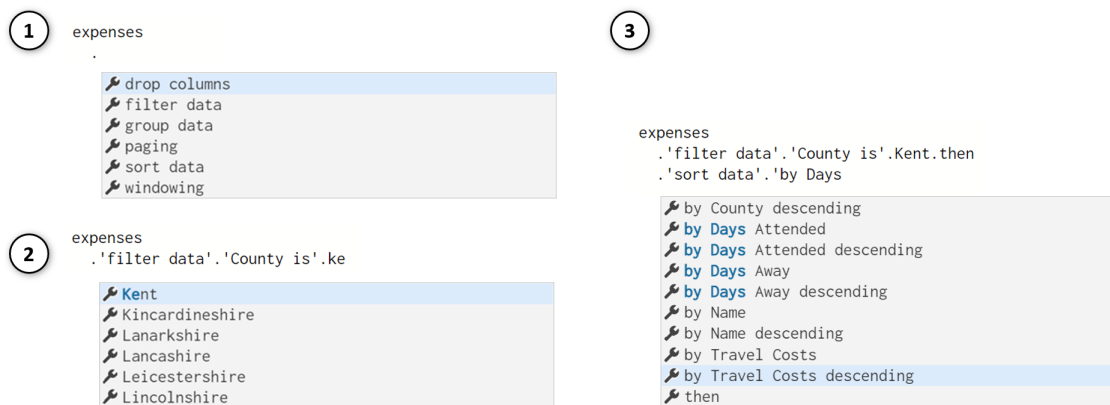


Figure 4.1: Using the iterative prompting interaction principle in The Gamma to explore dataset containing information on UK House of Lords members.

2. The user chooses `filter data`. They are then offered a list of conditions based on the columns in the dataset. The user selects `County is` and is then offered a list of all possible values of the column in the dataset. Thanks to the fact that iterative prompting in The Gamma is embedded in an ordinary text editor, they can start typing to filter the (long) list of possible values.
3. The user chooses `Kent` as the required value. They are then offered a list including further conditions and the `then` member that makes it possible to choose another transformation. They choose `then` and continue to add sorting.

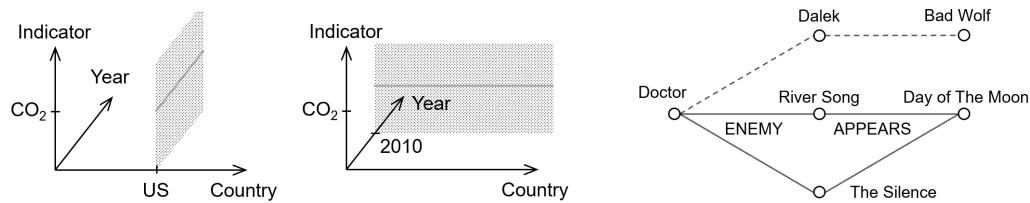
In The Gamma, the iterative prompting principle is used in the context of text-based programming language with type providers. This is a deliberate design choice. The aim of the work is to see whether iterative prompting can make text-based programming accessible to non-programmers. As a programming language, The Gamma is a simple object-oriented language with nominal type system and support for type providers. It allows a couple of constructs in addition to the method chaining shown in Figure 4.1 including `let` binding and method calls such as `expenses.paging.take(10)`. I briefly review the design trade-offs below.

4.2.1 Iterative prompting for data querying

The paper included as Chapter 10 shows that iterative prompting can provide a unified interface for exploring data from a range of different data sources. One of the hypotheses evaluated in the paper is that this aids usability by supporting transfer of knowledge between different kinds of data sources. To evaluate this, we implemented type providers for exploring data cubes (Syme et al., 2013), created by the author of this thesis, tabular data, as outlined in Chapter 2 and discussed in full in Chapter 7, and graph databases.

Data cubes are multi-dimensional arrays of values. For example, the World Bank collects indicators about many countries each year. The type provider makes it possible to select a data series, such as CO₂ emissions of the US over time:

- 1 `worldbank.byCountry.'United States'.`
- 2 `'Climate Change'. 'CO2 emissions (kt)'`



(a) Exploring World Bank data using the data cube type provider, users choose values from two dimensions to obtain a data series.

(b) To query graph data, the user specifies a path through the data, possibly with placeholders to select multiple nodes.

Figure 4.2: Design of type providers for exploring cube and graph data

The dimensions of the `worldbank` cube are countries, years and indicators. Figure 4.2a illustrates how the provider allows users to slice the data cube – byCountry. 'United States', restricts the cube to a plane and 'CO2 emissions (kt)' gives a series with years as keys and emissions as values. Similarly, we could first filter the data by a year or an indicator.

Graph databases store nodes representing entities and relationships between them. The following example explores a database of Doctor Who characters and episodes. It retrieves all enemies of the Doctor that appear in the Day of the Moon episode:

```
1 drwho.Character.Doctor.'ENEMY OF'. '[any]'
```

```
2 . 'APPEARED IN'. 'Day of the Moon'
```

The query is illustrated in Figure 4.2b. We start from the `Doctor` node and then follow two relationships. We use 'ENEMY OF'. '[any]'' to follow links to all enemies of the Doctor and then specify 'APPEARED IN' to select only enemies that appear in a specific episode. The members are generated from the data; 'ENEMY OF' and 'APPEARED IN' are labels of relations and `Doctor` and 'Day of the Moon' are labels of nodes. The [any] member defines a placeholder that can be filled with any node with the specified relationships. The result returned by the provider is a table of properties of all nodes along the specified path, which can be further queried and visualized.

Unlike the graph and data cube providers, the type provider for tabular data does not just allow selecting a subset of the data, but it can be used to construct SQL-like queries. For example, the code constructed in Figure 4.1 filters and sorts the data.

When using the provider, the user specifies a sequence of operations. Members such as 'filter data' or 'sort data' determine the operation type. Those are followed by members that specify operation parameters. For example, when filtering data, we first select the column and then choose a desired value. Unlike SQL, the provider only allows users to choose from pre-defined filtering conditions, but this is sufficient for constructing a range of practical queries.

4.2.2 Usability of iterative prompting

To evaluate the usability of iterative prompting, we conducted a user study for which we recruited 13 participants (5 male, 8 female) from a business team of a research institute working in non-technical roles (project management, communications). Our primary hypothesis was that non-programmers will be able to use iterative prompting to explore data,

but some aspects of the study were also designed to how users learn to use the mechanism and whether knowledge can be transferred between different data sources. The study methodology and detailed discussion of results can be found in Chapter 10. The key observations from the study are:

- *Can non-programmers explore data with The Gamma?* All participants were able to complete, at least partially, a non-trivial data exploration task and only half of them required further guidance. A number of participants shared positive comments in the group interviews. One participant noted that “this is actually pretty simple to use,” while another felt the system makes coding more accessible: “for somebody who does not do coding or programming, this does not feel that daunting.”
- *How users learn The Gamma?* There is some evidence that knowledge can be transferred between different data sources. In two of the tasks, participants were able to complete the work after seeing a demo of using another data source. One participant “found it quite easy to translate what you showed us in the demo to the new dataset.” Once users understood iterative prompting, they were also able to learn from just code samples and do not need to see a live demo of using the tool. One participant noted that “a video would just be this [i.e. a code sample] anyway.”
- *How do users understand complex query languages?* The tabular type provider uses a member then to complete the specification of a current operation, for example when specifying a list of aggregation operations. Two participants initially thought that then is used to split a command over multiple lines, but rejected the idea after experimenting. One participant then correctly concluded that it “allows us to chain together the operations” of the query. While iterative prompting allows users to start exploring new data sources, the structures exposed by more complex data sources have their own further design principles that the users need to understand.
- *What would make The Gamma easier to use?* Three participants struggled to complete a task using the tabular data source because they attempted to use an operation that takes a numerical parameter and thus violates the iterative prompting principle. Most participants had no difficulty navigating around in text editor and some participants used the text editor effectively, e.g. leveraging copy-and-paste. However, two participants struggled with indentation and a syntax error in an unrelated command. This could likely be alleviated through better error reporting.

4.3 AI assistants

Iterative prompting can be used as a mechanism for program construction, as illustrated in the previous section, but it can also be used to guide semi-automatic data wrangling tools. As discussed above, many systems that aim to simplify data wrangling using AI methods support only the *onetime interaction* pattern where the user invokes the tool and gets back a result that they can manually refine if needed. In the paper included as Chapter 11, we use iterative prompting as the basis for the AI assistants framework, which is a common structure for building semi-automatic data wrangling tools that incorporate human feedback. When using an AI assistant, the user invokes the assistant on some input data, but they can then repeatedly use iterative prompting to further constrain the solution.

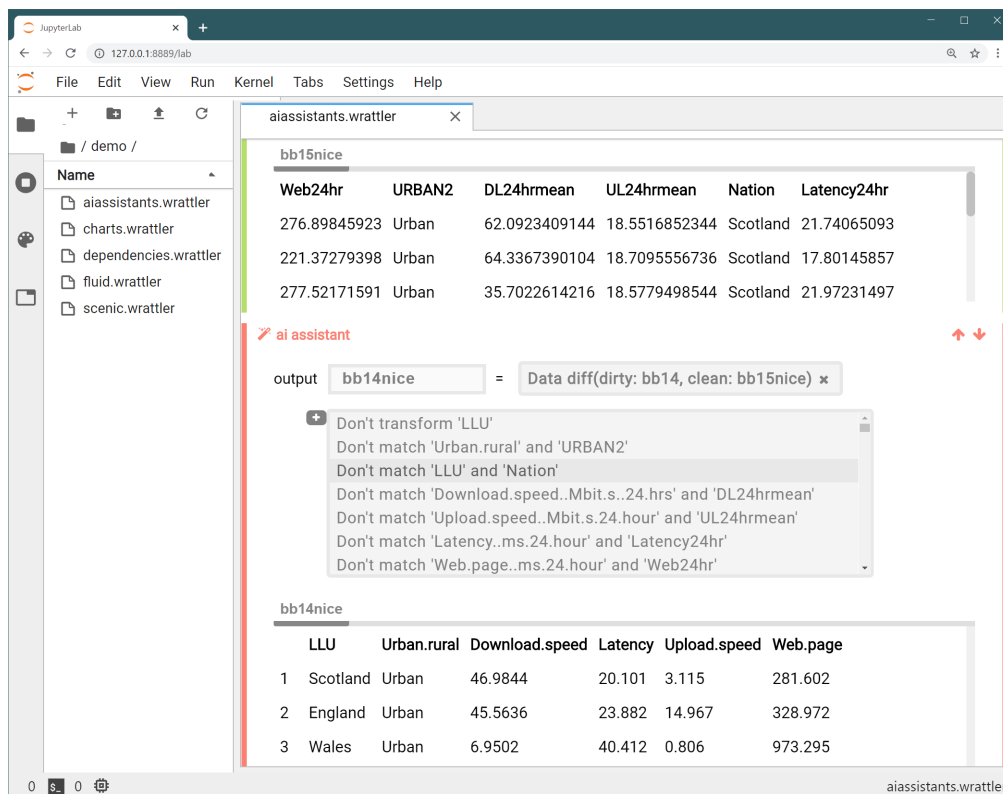


Figure 4.3: Using the datadiff AI assistant inside Wrattler to semi-automatically merge UK Broadband quality data from two files, parsed by an earlier R script. The user is in the process of adding a constraint to correct an error in the automatically inferred column matching.

As illustrated in Figure 4.3, AI assistants are available in the Wrattler notebook system discussed in Chapter 3. In addition to code cells that obtain, process, and visualize data, users can create AI assistant cells that invoke a semi-automatic data cleaning tool on some of the available datasets. After invoking the assistant, users are shown a preview of the generated clean dataset. If they see an error in the automatically inferred solution, they can choose one from the offered options to guide the AI tool and correct the error. Iterative prompting for AI assistants uses a graphical user interface, but the interaction mechanism of repeatedly choosing one from the offered options remains the same.

4.3.1 Merging data with Datadiff

To give an overview of how AI assistants work, consider the task of merging multiple incompatible datasets, using the UK broadband quality data, published by the UK communications regulator Ofcom.³ The regulator collects data annually, but the formats of the files are inconsistent over the years. The order of columns changes, some columns are renamed, and new columns are added. We take the 2014 dataset and select six interesting columns (latency, download and upload speed, time needed to load a sample page, country, and whether the observation is from an urban or a rural area). We then want to find corresponding columns in the 2015 dataset.

³ Available at: <https://www.ofcom.org.uk/research-and-data/data/pendata>

The 2015 dataset has 66 different columns so finding corresponding columns manually would be tedious. An alternative is to use the automatic datadiff tool (Sutton et al., 2018), which matches columns by analyzing the distributions of the data in each column. Datadiff generates a list of *patches* that reconcile the structure of the two datasets. A patch describes a single data transformation to, for example, reorder columns or recode a categorical column according to an inferred mapping. Datadiff is available as an R function that takes two datasets and several hyperparameters that affect the likelihood of the different types of patches.

When merging Broadband datasets, datadiff correctly matches five out of six columns, but it incorrectly attempts to match a column representing Local-loop unbundling (LLU) to a column representing UK countries. This happens because datadiff allows the recoding of categorical columns, and seeks to match them based on the relative frequencies in the two columns. Consequently, the inferred transformation includes a patch to recode the Cable, LLU, and Non-LLU values to Scotland, Wales, and England. To correct this, we could manually edit the resulting list of patches, or tweak the likelihood of the *recode* patch. Such parameter tuning is typical for real-world data wrangling, but finding the values that give the desired result can be hard.

The semi-automatic datadiff AI assistant presented in this chapter enables the analyst to guide the inference process by specifying human insights in the form of constraints. The AI assistant first suggests an initial set of patches with one incorrect mapping. After the analyst chooses one of the offered constraints, shown in Figure 4.3, datadiff runs again and presents a new solution that respects the specified constraints until, after two more simple interactions, it reaches the correct solution.

4.3.2 Formal model of AI assistants

The central contribution presented in Chapter 11 is a formal model of AI assistants that captures their structure. The chapter uses the standard methodology of theoretical programming language research, but applied to a problem from the data engineering research field. The definition of an AI assistant captures a common structure that semi-automatic data wrangling tools can follow in order to use iterative prompting as a mechanism for incorporating human insights into the data wrangling process.

The formal model defines AI assistants as a mathematical entity that consists of several operations, modeled as mathematical functions between different sets. Every AI assistant is defined by three operations that work with expressions e , past human interactions H , input data X , and output data Y . Expressions e can also be thought of as data-cleaning scripts. Input and output data are typically one or more data tables, often annotated with meta-data such as column types. While AI assistants share a common structure, the language of expressions e that an assistant produces, the notion of human interactions H , and the notion of X and Y can differ between assistants.

Definition 1 (AI assistant). Given expressions e , input data X , output data Y , and human interactions H , an AI assistant $(H_0, f, best, choices)$ is a tuple where H_0 is a set denoting an empty human interaction and f , $best$ and $choices$ are operations such that:

- $f(e, X) = Y$
- $best_X(H) = e$
- $choices_X(H) = (H_1, H_2, H_3, \dots, H_n)$.

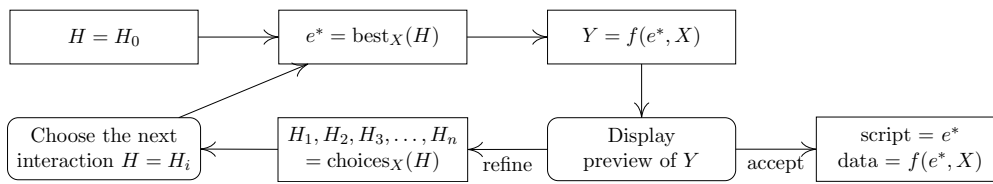


Figure 4.4: Flowchart illustrating the interaction between an analyst and an AI assistant. Steps drawn as rounded rectangles correspond to user interactions with the system.

The operation f transforms an input dataset X into an output dataset Y according to the expression (data cleaning script) e . The operation $best_X$ recommends the best expression for a given input dataset X , respecting past human interactions H . Finally, the operation $choices_X$ generates a sequence of options $H_1, H_2, H_3, \dots, H_n$ that the analyst can choose from (e.g. through the user interface illustrated in Figure 4.3). When interacting with an assistant, the selected human interaction H is passed back to $best_X$ in order to refine the recommended expression. Note that the sequence of human interactions given by $choices_X$ may be sorted, starting with the one deemed the most likely. To initialize this process, the AI assistant defines an empty human interaction H_0 .

The interesting AI logic can be implemented in either the $best_X$ operation, the $choices_X$ operation, or both. The f operation is typically straightforward. It merely executes the inferred cleaning script. Both $best_X$ and $choices_X$ are parameterized by input data X , which could be the actual input or a smaller representative subset to make working with the assistant more efficient.

The working of AI assistants is illustrated in Figure 4.4. When using the assistant, we start with the empty interaction H_0 . We then iterate until the human analyst accepts the proposed data transformation. In each iteration, we first invoke $best_X(H)$ to get the best expression e^* respecting the current human insights captured by H . We then invoke $f(e^*, X)$ to transform the input data X according to e^* and obtain a transformed output dataset Y . After seeing a preview of Y , the analyst can either accept or reject the recommended expression e^* . In the latter case, we generate a list of possible human interactions $H_1, H_2, H_3, \dots, H_n$ using $choices_X(H)$ and ask the analyst to pick an option H_i . We use this choice as a new human interaction H and call the AI assistant again.

The Definition 1 serves both as a model that can be studied formally, but also as the basis for an implementation interface of AI assistants. The shared structure makes it possible to separate the development of individual AI assistants from the development of tools that use them, such as the AI assistant cell type implemented in Wrattler.

4.3.3 Practical AI assistants

To show that AI assistants provide a common structure for a wide range of semi-automatic data wrangling tools, the work included as Chapter 11 takes four existing AI-based data wrangling tools that follow the *onetime interaction* pattern and turn them into interactive tools that follow the *iterative* pattern. The original tools cover the entire spectrum of data wrangling ranging from parsing of CSV files (van den Burg et al., 2019) and merging data files (Sutton et al., 2018) to type and semantic information inference (see Chapter 11).

The approach we use for turning a non-interactive AI tool into an interactive AI assistant is similar in all four cases. The non-interactive tools generally define an objective function $Q(e, X)$ that scores data cleaning scripts (expressions e) based on how well they clean the specified input data X . The automatic AI tool performs an optimization, looking

for the best data cleaning from the set of all possible expressions E for the given data. Formally, the optimization task solved by the existing tools can be written as:

$$\arg \max_{e \in E} Q(X, e)$$

The AI assistants that we implement and formally describe in Chapter 11 adapt this optimization to take account of the human interactions H that have been collected through the iterative prompting process illustrated in Figure 4.4. For a given human interaction H (starting with H_0), we define a set of expressions E_H that is filtered to only include expressions satisfying the condition specified by the user through H . We also define a parameterized objective function Q_H that is based on the original Q but increases or decreases the score for certain expressions based on H . Given these two definitions, it is possible to define the $best_X(H)$ operation as solving an optimization problem:

$$best_X(H) = \arg \max_{e \in E_H} Q_H(X, e)$$

The four concrete AI assistants that we developed use this definition, but they do not always use human interactions to tweak both E_H and Q_H . It is often sufficient to restrict the set of expressions used by the search and reuse the original unmodified optimization algorithm. The implementation of the AI assistants (available in Wrattler) generally required a modification of the underlying non-interactive tool. The modification is tool-specific as each of the AI assistants is based on a different kind of search algorithm. The four practical AI assistants presented in Chapter 11 work as follows:

- The *datadiff AI assistant* infers a list of patches that transform the input dataset into a format matching that of the given reference dataset. The assistant optimizes score based on the similarity of the data distributions of the matched columns. The semi-interactive AI assistant allows the user to specify that certain patches (e.g., matching two particular columns) should or should not be included in the resulting set.
- The *CleverCSV AI assistant* infers formatting parameters of a CSV file to optimize a metric based on how regular the resulting parsed result is. The semi-interactive AI assistant allows the user to specify that a given character should be or should not be used as a delimiter, a quote, or an escape character.
- The *ptype AI assistant* infers types of columns in a dataset, detecting outliers and values representing missing data. The optimization function looks for a type with maximal likelihood based on a probabilistic model. The semi-interactive AI assistant allows the user to reject any aspect of the inferred type (type itself, outlier, missing value), effectively forcing the search to look for the next most likely type.
- The *ColNet AI assistant* annotates data with semantic information from a knowledge graph such as DBpedia (Lehmann et al., 2015). It uses a Convolutional Neural Network model to calculate the score that sampled data is of a given semantic type and then finds the type with the greatest score. The semi-interactive AI assistant adapts the scoring, allowing the user to specify that a given sample is (or is not) of a specified semantic type.

In Chapter 11, we evaluate the effectiveness of the four AI assistants both qualitatively and quantitatively. Our qualitative evaluation uses three scenarios in which the different earlier data wrangling tools are unable to solve a real-world data wrangling challenge using the *onetime* interaction. We document how the user can use iterative prompting to obtain

the desired result, by repeatedly choosing one option from the offered list. To evaluate AI assistants quantitatively, we developed a benchmark that counts how many human interactions are needed to complete a given data wrangling task for multiple datasets (either reusing an existing benchmark or synthetically generated). The evaluation shows that 1-2 human interactions are usually sufficient to complete the task.

4.4 Contributions



Key contributions. The publications included in Part IV include three main contributions. They introduce the novel *iterative prompting* interaction principle. They use it as the basis of *AI assistants*, novel semi-automated data wrangling tools, as well as multiple *type providers* for accessing data in graph databases, data cubes, and relational databases.

This chapter brings together two contributions that aim to reduce the gap between programming and spreadsheets by making two tasks that typically require some kind of programming easier. In the first contribution, I focused on data exploration, whereas the second contribution tackles the task of data wrangling. My work shows that, in both cases, it is possible to solve a large class of problems using the *iterative prompting* interaction principle where the user repeatedly chooses one from the offered options. The interaction principle is simple in that it reduces the cognitive load by using the *recognition over recall* design heuristic. When using iterative prompting, the users do not need to recall the kind of operation they could use to solve the problem. Instead, they can review the list of offered options and recognize the most suitable one.

The work included in Chapter 10 introduces the *iterative prompting* interaction principle and uses it to view the type provider for data querying outlined in Chapter 2 from a novel perspective using the human-computer interaction research methodology. Rather than treating auto-completion as a programmer assistance tool, it is now used as a mechanism that allows non-programmers to construct entire programs. The key characteristics of the type provider that make this possible are that it is complete and correct, i.e. it makes it possible to construct all programs and any program constructed by repeatedly choosing one of the offered options is correct (even though some may result in empty data). The user study that I briefly discussed in this chapter shows that iterative prompting can be used by non-programmers to complete a range of data exploration tasks in a code-oriented environment. This suggests that it is possible to combine the reproducibility and transparency of using code with ease of use approaching that of spreadsheets.

The work included in Chapter 11 uses the iterative prompting interaction principle (albeit without using the term) to provide human insights to semi-automatic data wrangling tools that I refer to as AI assistants. The challenge addressed by AI assistants is how to guide data wrangling tools based on AI techniques. Although such tools can solve many problems automatically, the complexity of real-world data sets often means that some human guidance is needed. Iterative prompting provides an easy method through which humans can provide such guidance. The chapter introduces a formal model of AI assistants and uses it as the basis for the implementation of four practical tools.

The contributions presented in this chapter link together many of the themes and contributions discussed in earlier chapters. In particular, the notion of type providers was introduced as a programming tool from the programming language theory perspective in Chapter 2. This chapter provides an alternative human-centric perspective. The techniques discussed in Chapter 3 make type providers even more usable by providing live previews during their usage. Finally, the Wrattler notebook system serves as a platform for integrating many of the experiments discussed in this thesis. For example, it makes it possible to combine interactive AI assistants with conventional programmatic data exploration using the widely used Python and R languages.

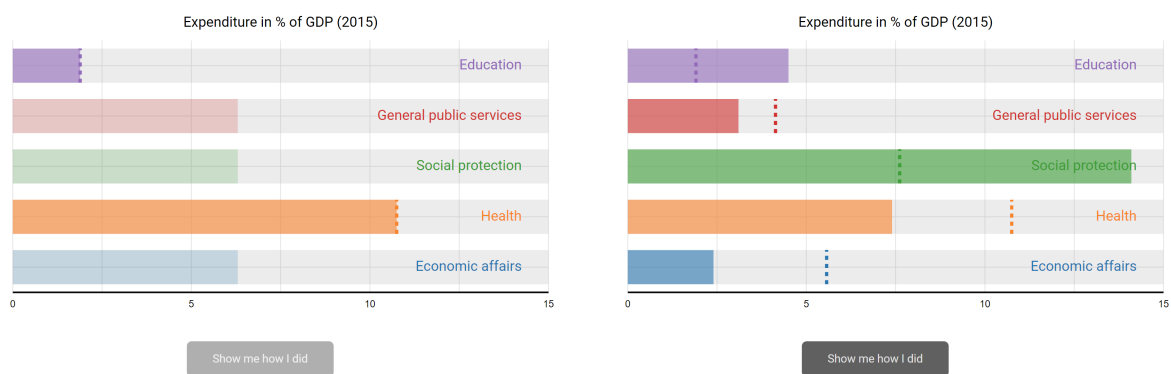
Chapter 5

Data visualization

Data visualization plays a dual role in the data science lifecycle. Quick data visualizations are needed during data exploration to help data analysts make sense of data, find errors, and understand how their processing scripts work. However, data visualizations can also be one of the outcomes of data science projects. In particular, data journalists often analyze data in order to find interesting insights and share those with their readers. A sophisticated and illuminating data visualization can be a powerful tool for such storytelling.

Producing a quick data visualization during data exploration is usually easy. In programmatic environments, it is typically a matter of calling a function with a few parameters to specify the type of chart one wants to see. However, developing a data visualization that helps the reader gain insight into a complex problem and critically think about it is typically a challenging programming task.

As an example, consider the interactive data visualization shown in Figure 5.1, created using the Compost library discussed below. The visualization is inspired by the New York Times “You draw it” article series (Aisch et al., 2015). It encourages critical thinking by first asking the reader to make a guess about the actual data. Only after the reader drags the bars according to their presuppositions, the chart reveals the actual values.



(a) The user first has to guess what the values are (here, guess how much the UK government spends per category).

(b) After clicking a button, actual data is shown (together with a marker showing the guess).

Figure 5.1: An interactive data visualization to encourage critical thinking about data created using the composable Compost data visualization library.

The chart is based on a standard bar chart, but there are multiple additional aspects that make creating such a chart a challenging programming problem:

- The chart combines multiple different visual elements. In addition to the bars themselves, it also needs to include the markers (dashed lines) that show the guess.
- The chart uses a custom color scheme, and background to indicate possible areas of the bar and it greys out the bars for which the user has not yet made a guess.
- The chart is interactive, allowing the user to drag the end of the bar to any location in the specified range (until the button is clicked).
- Once the button is clicked, a brief animation runs, and the bars move from the guessed value to the actual value (the marker stays at the original position).

Although numerous charting libraries support some of the above features, creating a custom data visualization such as the above typically requires using a low-level visualization library such as D3 (Bostock et al., 2011), which requires advanced programming skills.

More advanced programming skills are needed if one wants to implement data visualizations that support features such as brushing and linking (Buja et al., 1991). The former allows the user to focus on a particular region of the chart, while the latter connects two charts and adapts the visualization in the second chart based on the data selection in the first chart. Implementing linking is particularly challenging because it requires understanding what inputs contributed to the selected data points and then recomputing the data displayed in the other visualization.

In the following two sections, I provide an overview of two systems that are presented in Part V. The systems make it easier to create rich interactive visualizations. First, the paper included as Chapter 12 presents Compost, a novel functional data visualization library that makes it possible to compose rich charts from a small number of primitive building blocks and combinators. Second, the paper included as Chapter 13 presents a program analysis technique that can be used to automatically create linked data visualizations based on the code of scripts that construct charts from shared data. The two systems are aimed at programmers, but they are simple in that they make it possible to create sophisticated interactive visualizations using a small amount of straightforward code.

Both of the papers introduced in this chapter use programming language research methods. Chapter 13 presents the analysis technique formally, using a small model programming language, and discusses its properties. Chapter 12 gradually introduces the concepts of the Compost library in the form of a tutorial. Published as a functional pearl (Gibbons, 2010), it relies on the tacit assessment of the functional programming community.

5.1 Visualisations to encourage critical thinking

Data visualizations that aim to present data-driven insights to a broader audience often require significant programming effort. The “You draw it” series by New York Times (Aisch et al., 2015) lets the user draw on the chart, while the award-winning visualization of population density in Asian cities by Bremer and Ranzijn (2015) tells a story through multiple animated and interlinked charts. Visualizations like these are often built using D3 (Bostock et al., 2011), by constructing the chart piece by piece. D3 is easier than drawing pixels or primitive shapes, but it still requires tediously transforming values to coordinates, specifying positions in pixels, and modifying shape properties in response to events.

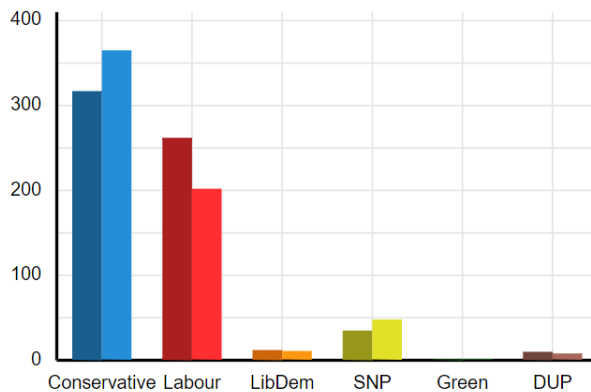


Figure 5.2: A bar chart that compares the UK general election results for years 2017 (left) and 2019 (right), created using the Compost library.

Higher-level compositional approaches to chart construction are typically based on the grammar of graphics (Wilkinson, 1999). In the grammar of graphics, a chart is a mapping from data to chart elements and their visual attributes. Libraries based on this idea include ggplot2 (Satyanarayan et al., 2016; Wickham, 2016) and Vega (Wickham, 2010). The mapping is limited to a number of built-in operations, which works well for common types of scientific charts, but has a limited generality. For example, in Altair (VanderPlas et al., 2018), it is possible to link multiple charts by specifying a data transformation that relates them, but this has to be specified using a limited set of combinators provided (and understood) by the library.

In contrast to systems based on the grammar of graphics, the two systems presented in this chapter rely on the host programming language to specify the mapping from data to chart elements. A chart is merely a resulting data type describing the visual elements using domain-specific primitives. In the two chapters, summarised in the next two sections, we first define a small, orthogonal set of expressive primitives and then introduce a program analysis technique that can automatically infer the mapping between source data and elements of the chart.

5.2 Composable data visualisations

The Compost library, presented in Chapter 12 can be seen as a functional domain-specific language for describing charts. As is often the case with domain-specific languages, finding the right primitives is more of an art than science. The Compost library is designed in a way that gives it a number of desirable properties:

- Concepts such as bar charts, line charts, or charts with aligned axes are all expressed in terms of more primitive building blocks using a small number of combinators.
- The primitives are specified in domain terms. When drawing a line, the value of an y coordinate is an exchange rate of 1.36 USD/GBP, not 67 pixels from the bottom.
- Common chart types such as bar charts or line charts can be easily captured as high-level abstractions, but many interesting custom charts can be created as well.
- The approach can easily be integrated with the Elm architecture (Czaplicki, 2016) to create web-based charts that involve animations or interaction with the user.

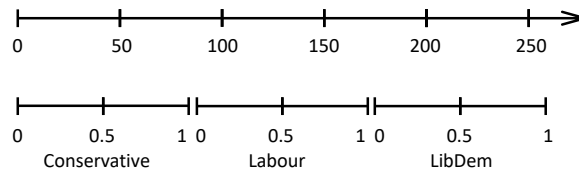


Figure 5.3: On a continuous scale (above), a position is determined by a number. On a categorical scale (below), a position is determined by the category and a number between 0 and 1.

5.2.1 Declarative chart descriptions

To illustrate the first two points, consider the chart in Figure 5.2, which compares UK election results for the years 2017 and 2019. In the chart, the x axis shows categorical values representing the political parties such as “Conservative” or “Labour”. The y axis shows numerical values representing the number of seats won such as 365 MPs. When creating a chart, most high-level libraries such as Google Charts expect values in domain terms, but more flexible libraries like D3 expect the user to first explicitly translate such domain values to pixels. In Compost, the user composes primitive graphical elements such as filled rectangles, but their position is specified in terms of domain values.

Our design focuses on two-dimensional charts with x and y axes. Values mapped to those axes can be either categorical (e.g. political parties, countries) or continuous (e.g. number of votes, exchange rates). The mapping from categorical and continuous values to positions on the chart, as well as the range of values associated with a scale, are calculated automatically. Figure 5.3 illustrates the two kinds of values. A continuous value, written as `cont n` contains any number n . A categorical value `cat c, r` consists of a categorical value c and a number r between 0 and 1. The second parameter determines an exact position in the range allocated for the categorical value such as “Green”.

Assuming we have a list `elections` which contains 5-element tuples with the party name, colours for 2017 and 2019 and the number of MPs for 2017 and 2019, we can construct the chart in Figure 5.2 as follow (using F# or similar language with list comprehensions):

```

1 axis_l (axis_b (overlay [
2   for party, c17, c19, mp17, mp19 in elections →
3     padding 0, 10, 0, 10, overlay [
4       fill clr17, [
5         (cat party, 0), (cont 0); (cat party, 0), (cont mp17);
6         (cat party, 0.5), (cont mp17); (cat party, 0.5), (cont 0) ],
7       fill clr19, [
8         (cat party, 0.5), (cont 0); (cat party, 0.5), (cont mp19);
9         (cat party, 1), (cont mp19); (cat party, 1), (cont 0) ]
10    ]
11  ]))

```

The central part of the code (lines 4-6 and 7-9) constructs two filled rectangles (bars) representing the number of MPs for 2017 and 2019, respectively. Each rectangle is specified by four corners (separated using “;”). The y axis is continuous and the rectangle occupies space from 0 to the specified number. The x axis is categorical. The first bar takes the first half of the space available for the party (0 to 0.5) while the second occupies the second

half (0.5 to 1). We then compose the two rectangles using `overlay`, which ensures they are rendered on a shared scale. The padding primitive adds a space around a given shape (specified in pixels). We generate a pair of rectangles for each party using a list comprehension and then overlay all the rectangles before adding axes on the left and bottom.

In addition to the primitives illustrated by the above example, Compost has a number of other basic shapes (lines, text, bubbles). Perhaps more interestingly, there are also a couple of combinators that make it possible to combine charts or create charts that share axes. The `nest` combinator, explained in Chapter 12, takes a shape (with its own scales) and nests it inside an explicitly specified range. We can, for example, take a space defined by a categorical value (from `cat c, 0` to `cat c, 1`) and nest another shape, or even a line chart, inside this space. In practice, this is useful for combining multiple charts. The combinator can also apply to one axis only, making it possible to create two charts that are side-by-side on one axis but share the other axis.

5.2.2 Rendering a Compost chart

The rendering logic of Compost takes a declarative chart description such as the one generated by the simple functional program above and transforms it to an SVG in three steps:

- *Inferring the scales of a shape.* The implementation first recursively walks over the composed shape and infers the ranges of the x and y scales of the shape. For shapes constructed using `overlay`, this unions the ranges of the scales of the contained shapes. In the case of continuous scale, we take the overall minimum and maximum. For a categorical scale, the sets of categories obtained for each shape are unioned.
- *Projecting coordinates.* Once the chart is annotated with the inferred scales, we turn all positions from values defined in domain terms to values specified in pixels. This is done using a projection function that takes a scale, a space it should be mapped onto (in pixels) and a value on the scale. The results are x and y coordinates in pixels.
- *Rendering chart shapes.* Finally, the recursively defined shape is turned into a flat list of shapes in the SVG format. This involves collecting and concatenating all the primitive shapes, lines, and text elements.

The implementation of the core logic consists of only 800 lines of code. Although the process is conceptually simple, there are a number of subtle details. In particular, operations that specify some parameters in pixels (such as padding) have to transform the projection operation so that the resulting shapes only occupy a space without the specified padding. Nesting also requires keeping track of the outer range and the inner scale. It is also worth noting that some operations, such as `axis` that add an axis with labels, can be eliminated at some point in the process. In particular, `axis` is replaced by overlaid lines and text elements in the first step.

5.2.3 Functional abstraction and interactivity

As noted earlier, Compost differs from libraries based on the grammar of graphics such as `ggplot2` (Wickham, 2016) that treat a chart as a mapping from data to chart elements and their visual attributes. In Compost, a chart is a concrete description of chart elements, generated from data by code written in an ordinary programming language. I illustrated

this above with the code that generated a bar chart using list comprehensions. This means that Compost can leverage other capabilities of the host language and its ecosystem.

First, it is possible to easily introduce new higher-level chart abstractions. For example, the chart shown in Figure 5.2 is sometimes referred to as Dual X-axis Bar Chart. Some high-level libraries such as Google Charts support this directly. We saw that the chart can be constructed using Compost, but in a somewhat tedious way. However, in a host language that lets us define new functions like F#, we can introduce a new abstraction for this kind of chart. The following merely extracts the rectangle construction from the previous example into a function `dualXBar`:

```
1 let dualXBar xcat clr1 clr2 yval1 yval2 = overlay [
2   fill clr1, [ (cat xcat, 0), (cont 0); (cat xcat, 0), (cont yval1);
3     (cat xcat, 0.5), (cont yval1); (cat xcat, 0.5), (cont 0) ],
4   fill clr2, [ (cat xcat, 0.5), (cont 0); (cat xcat, 0.5), (cont yval2);
5     (cat xcat, 1), (cont yval2); (cat xcat, 1), (cont 0) ]
6 ]
```

We can now use the function inside a list comprehension to construct the original chart in just three lines of code:

```
1 axisl (axisb (overlay [
2   for party, c17, c19, mp17, mp19 in elections →
3     dualXBar party c17 c19 mp17 mp19 ]))
```

One last interesting aspect of the Compost library that is discussed in Chapter 12 is the support for interactivity. The library can be used in conjunction with the Elm (model-view-update) architecture (Czaplicki, 2016) where the programmer defines the program *state* and *events*. They then provide a function that renders a chart based on the current state. They also specify a function that updates the state when an event occurs. For example, in the earlier interactive chart in Figure 5.1, one event is clicking on a bar, which then updates the guessed value. Compost makes programming such charts easier by reporting events in terms of domain values (using a backward projection). When the user clicks on a bar, the event handler receives a pair of values such as `cat "Health", 0.3` and `cont 12.7`. It then updates the guessed value for the category Health to the new value 12.7% GDP.

5.3 Automatic linking for data visualizations

The Compost library makes it possible to compose an appealing data visualization from individual graphical elements. This is necessary if we want to create rich interactive charts. However a single chart that offers a single perspective is not enough if we want to explain complex data. To support better understanding, *linked visualisations* (Buja et al., 1991) consist of multiple charts that display different aspects of the same data. When the user selects an element in one of the charts, the elements that are based on related data as the selected one are highlighted in the other charts. This makes it possible to relate different perspectives on the same data.

For example, consider the visualization in Figure 5.4 that displays data on energy production over time for different countries and different types of energy. A single chart with three variables would be difficult to read, so the visualization instead filters and aggregates the data in two ways. It shows data per country for the year 2015 and the ratio of energy

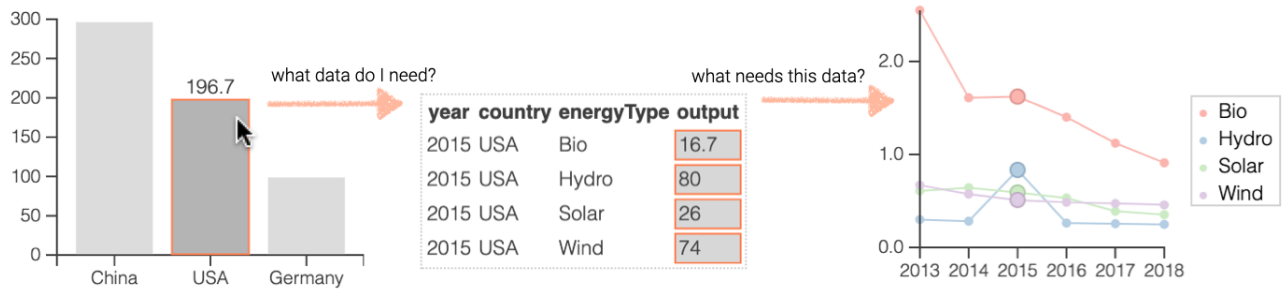


Figure 5.4: Linked data visualization of energy production, showing aggregated data per country for 2015 (left) and timeline with the ratio of production in the USA and China per energy type (right).

produced in the USA and China over time for each type of energy. The user can select a particular data point (bar on the left, point on the right). If they select the bar representing the USA, the system infers which input data contributed to the value (data for all types of energy for the USA in 2015) and computes what chart elements depend on this data in the second chart (data points for 2015). Selecting Germany would not highlight any points as the right chart depends only on data points for China and the USA.

Constructing linked visualization like the one in Figure 5.4 is difficult because the data visualization needs to understand how to map selection between the charts. This can be done automatically for simple data transformations in libraries based on the grammar of graphics such as Altair (VanderPlas et al., 2018). In those, simple data transformations can be expressed using primitives provided by the library. However, the approach does not work if the data transformation is more complex or if the user prefers to express it in an ordinary programming language (as ordinary list processing) as opposed to using a special-purpose domain-specific language (grammar of graphics primitives).

In Chapter 13, we use dynamic dependency analysis techniques to simplify the creation of linked data visualizations. Those techniques have traditionally been used in areas such as information-flow security, optimization, and debugging. In our work, we extend the techniques so that they can provide fine-grained dependency information for programs constructing structured outputs, such as the data structures used to describe charts in the Compost data visualization library. We introduce a simple functional programming language Fluid that the users can use to write arbitrary data transformations and construct charts. By combining two dynamic dependency analyses, we can then automatically infer the relationships between multiple charts constructed from the same data.

5.3.1 Creating linked visualizations using Fluid

Consider again the visualization shown in Figure 5.4. The two charts are based on the same input data, which is a collection of records storing the country, energy type, year, and the amount produced. To produce the chart on the left, we filter the data by year and then sum the produced energy for each of the countries. In Fluid, this is written as a simple functional program using list comprehensions:

```

1 let data2015 =
2   [ row | row ← data, row.year == 2015 ]
3 let totalFor c =
4   sum [ row.output | row ← data2015, row.country == c ]
5 let countryData =

```

```

6   [ { x: c, y: totalFor c } | c ← ["China", "USA", "Germany" ]
7   BarChart { data: countryData }

```

The code is similar to what one would write in widely-used functional languages such as F# or Haskell. It defines filtered `data2015`, a helper function `totalFor` and then computes data per country using the helper before constructing the resulting chart. The last step uses a built-in `BarChart` function. As discussed earlier, this could be implemented as an abstraction over more basic Compost primitives.

The program analysis automatically infers what values from the source data contribute to the resulting `y` value, computed using the `totalFor` function. The relevant rows are only those for the year 2015 (due to the filtering) and only those for the given country (due to the `totalFor` implementation). It is worth noting that unlike approaches based on the grammar of graphics, we are not restricted to a set of pre-defined primitives. The data transformation can be arbitrary and we can write it using custom functions such as `totalFor`.

The code to construct the second chart is slightly more complicated because it constructs a line chart with multiple lines. It defines `series` helper to obtain a time series for a given country and energy type, `plot` helper that calculates the ratio between the USA and China for a given energy type and then composes the chart:

```

1  let series type country =
2    [ { x: year, y: row.output } | year ← [2013..2018], row ← data,
3      row.year == year, row.energyType == type, row.country == country ]
4  let plot type =
5    zipWith (fun p1 p2 → { x: p1.x, y: p1.y / p2.y })
6      (series type "USA") (series type "China")
7  LineChart { plots: [
8    LinePlot { name: type, data: plot type }
9    | type ← ["Bio", "Hydro", "Solar", "Wind" ] ] }

```

The example illustrates two requirements that we have for dynamic dependency analyses that let us automatically generate linked data visualizations. First, we are not interested in the resources (code and data) needed to produce the entire result, but only in the resources needed to produce a part of the result. Specifically, if the user clicks on a data point of the line chart, we want to know what resources contributed to the `data` field of a specific record in the list passed to one particular `LinePlot`. In the analyses, we address this by introducing the concept of a selection for structured values, which lets us mark a particular part of the value. (We use this mechanism for analyzing charts, but it could equally be used to analyze code that produces e.g., representations of documents.)

Second, we need multiple different kinds of dependency analyses to automatically link the charts. If the user selects a part of one chart, we need a *backward* analysis to trace it back to the original data. This analysis needs to determine parts of the input that were needed for the output, even if they were also used elsewhere (an alternative analysis would look for inputs needed only for the particular output). Then we need a *forward* analysis to determine what outputs it affects in the linked chart. This, again, needs to determine parts of the output that needed the specific parts of the input, even if they also needed other unmarked inputs (an alternative analysis would look for outputs that only needed the particular inputs). As we will see, this analysis leads to four different program analyses. We use two of them to automatically generate linked data visualizations.

5.3.2 Language-based foundation for explainable charts

The program analyses introduced in Chapter 13 can be, more broadly, seen as tools that help us understand how programs work. They could be used to support a range of use cases outside of the narrow domain of data visualization. However, to keep the presentation simpler, I will focus on this particular use case. Given the scope of the Fluid language and the considerable number of analyses, I do not discuss many details in this overview and focus on sketching the overall structure of the analyses and their key properties.

The analyses are built around the central notion of a program trace T . The traces are generated during program evaluation and they collect information about how the evaluation proceeded. More formally, a trace is a compact representation of the derivation trees in the operational semantics. Given a program e and a variable environment ρ , the following judgment states that the program reduces to a value v , producing a trace T :

$$T :: \rho, e \Rightarrow v$$

The program analyses operate on values where some parts are marked as selected. For example, if a program e produces a value v that represents a chart using the primitives of the Compost library, we can mark some parts of the resulting chart, for example, a bar created using the `fill` primitive.

Formally, we annotate values and terms with selection states α from a bounded lattice of selections \mathcal{A} . It is possible to annotate primitive values such as numbers n_α or empty lists $[]_\alpha$ but also values that contain further values such as a non-empty list (a cons cell) such as $v_1 :_\alpha v_2$. This would indicate that we are interested in data that determined that the list will be non-empty (but not the specific values it contains). In practice, the most useful kind of annotation is a two-point lattice (selected, non-selected), but the generality allows for other uses, such as a vector of selections (to be computed in one step and displayed using different colors).

5.3.3 Bidirectional dependency analyses

The program analyses that are introduced in Chapter 13 are defined for a fixed computation $T :: \rho, e \Rightarrow v$ where T is the trace. In practice, the system evaluates the expression e and uses the resulting trace T repeatedly for different analyses and with multiple possible selections. The first two analyses that we define are the backward and forward data dependency analyses written as \Downarrow_T and \Uparrow_T , respectively. The two analyses are defined over the trace T and take the availability-annotated variable context and an expression ρ, e as inputs. They produce an annotated value v , which is the same as the value computed initially, but with availability annotations. Their intuitive meaning is as follows:

- *Forward data dependency* $\rho, e, \alpha \Downarrow_T v$ or “*what only needs me*”. Given a context and expression where annotations indicate what resources (data and parts of the program) are available, the analysis produces a value with annotations indicating what part of the value can be computed using only the available resources.
- *Backward data dependency* $\rho, e, \alpha \Uparrow_T v$ or “*what do I need*”. Given an annotated value v , the analysis follows the evaluation backward (using the trace T) and annotates resources (data and parts of the program) that have to be available in order to produce the annotated parts of the value v .

The two analyses form a Galois connection, i.e., the two analyses are not exactly inverses, because they approximate in some ways, but they are close. For example, if we have a value with a selection v , ask what is needed for the selection using \nearrow_T and then ask what can be computed using the same data using \searrow_T , we may find that other parts of the value can also be computed, but the original selection will certainly also be included.

Interestingly, the forward and backward data dependency analyses are not sufficient to support linked data visualizations. The backward analysis can correctly determine what parts of the input are needed in order to produce the selected output. However, to highlight the related elements of the other chart, we do not need to know what the input selection is *sufficient* for, but what it is *necessary* for. In other words, we do not need to know what can be computed using only the selected parts of the input, but what part of the result will they certainly contribute to. This can be formulated as a dual of the original analyses. What would we not be able to compute if the selected data were unavailable?

In the paper included as Chapter 13, we define this duality formally and exploit it to define De Morgan dual ($\searrow_T^\circ, \nearrow_T^\circ$) of the Galois connection (\nearrow_T, \searrow_T), which itself also forms a Galois connection. The intuitive meaning of the new analyses is as follows:

- *Dual forward data dependency* $\rho, e, \alpha \searrow_T^\circ v$ or “*what needs me*”. For an annotated context and expression (resources), the analysis highlights what parts of the resulting value depend on the specified resources, i.e., what would we not be able to compute if the selected parts of the value were unavailable.
- *Dual backward data dependency* $\rho, e, \alpha \nearrow_T^\circ v$ “*what do only I need*”. Given an annotated value, the analysis highlights parts of the program and data that are only needed for producing the selected output, i.e., resources that would not be needed if the selected part of the output was not needed.

In order to construct a linked data visualization, we need to combine two of the data analyses. When the user selects a part of one chart, we annotate the relevant part of the structured value that represents the chart with a selection $\alpha \in \mathcal{A}$ that marks the part. We then use the backward data dependency analysis \nearrow_T to compute what resources are needed for producing the output. To mark corresponding parts of another chart, we then use the dual forward dependency analysis \searrow_T° . This marks parts of the other chart that depend on the resources needed for the first chart.

5.4 Contributions



Key contributions. The publications included in Part V include two main contributions. They introduce a *composable data visualization* library based on novel functional design and a dynamic program analysis technique that enables easy construction of *linked data visualizations*.

In this chapter, I provided an overview of the contributions to data visualization included in Part V. The focus of this work has been on simplifying the construction of rich interactive charts that assist the viewer gain deeper insight into the presented data. While producing simple charts is often easy, building a visualization that combines custom visual elements, interactivity and allows linking multiple charts requires advanced programming skills. Unlike with the work outlined in Chapter 4, I do not hope that the work presented here will

allow non-programmers to create rich interactive data visualizations. However, Compost and Fluid show that programming data visualisations that encourage critical thinking about data can be significantly easier than using systems that are widely used today.

The two contributions presented in this chapter also serve as a good justification for using programming language research methods in the context of tools for data science. In particular, the work included as Chapter 12 relies on the minimalist design of highly composable functional (domain-specific) languages. It presents the Compost charting library, which lets programmers compose charts by combining a small number of primitives (such as a filled shape or a text) using a small number of combinators (overlying, nesting of scales). The library is simple to use mainly due to the fact that all positions are specified in terms of domain units on a continuous numerical scale or a categorical scale (with 0 to 1 range for each category).

The work included as Chapter 13 uses dynamic program analysis techniques to automatically create linked data visualization where the user can explore relationships between data in multiple charts. The system works by analyzing the program that is used to construct the chart from source data (by filtering and aggregating it in various ways). When the user clicks on an element of a chart, the system uses two different dynamic dependency analysis techniques to highlight elements of other charts that depend on the same data as the element selected by the user. The Chapter 13 uses methods of programming language theory to describe the mechanism. We formalize the programming language, define two analyses, derive their duals, and describe their formal properties.

Although the prototype implementation of the two systems is not currently integrated, the two contributions presented in this chapter are closely related. In particular, the program analysis techniques developed for Fluid would combine well with a composable data visualization library like Compost.

In this chapter, I also completed one loop of the data science lifecycle illustrated in Figure 1.4. The loop started with data acquisition using type providers, continued with data cleaning and exploring using novel notebook systems and iterative prompting and now concludes with data visualization. This is the last step in an exploratory phase of the data science lifecycle where the aim is to understand and explain interesting aspects of data. An illuminating data visualization that conveys interesting insights found in data is often the end goal of the process. The production phase that aims to turn the result of the data analysis into a running component of a larger system is perhaps equally interesting but is out of scope for this thesis.

Part II

Publications: Type providers

Chapter 6

Types from data: Making structured data first-class citizens in F#

Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. Types from data: making structured data first-class citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 477–490. <https://doi.org/10.1145/2908080.2908115>

Types from data: Making structured data first-class citizens in F#

Tomas Petricek

University of Cambridge
tomas@tomasp.net

Gustavo Guerra

Microsoft Corporation, London
gustavo@codebeside.org

Don Syme

Microsoft Research, Cambridge
dsyme@microsoft.com

Abstract

Most modern applications interact with external services and access data in structured formats such as XML, JSON and CSV. Static type systems do not understand such formats, often making data access more cumbersome. Should we give up and leave the messy world of external data to dynamic typing and runtime checks? Of course, not!

We present F# Data, a library that integrates external structured data into F#. As most real-world data does not come with an explicit schema, we develop a shape inference algorithm that infers a shape from representative sample documents. We then integrate the inferred shape into the F# type system using type providers. We formalize the process and prove a relative type soundness theorem.

Our library significantly reduces the amount of data access code and it provides additional safety guarantees when contrasted with the widely used weakly typed techniques.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

Keywords F#, Type Providers, Inference, JSON, XML

1. Introduction

Applications for social networks, finding tomorrow's weather or searching train schedules all communicate with external services. Increasingly, these services provide end-points that return data as CSV, XML or JSON. Most such services do not come with an explicit schema. At best, the documentation provides sample responses for typical requests.

For example, <http://openweathermap.org/current> contains one example to document an end-point to get the current weather. Using standard libraries, we might call it as¹:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright 2016 held by Owner/Author. Publication Rights Licensed to ACM.

PLDI '16 June 13–17, 2016, Santa Barbara, CA, United States
Copyright © 2016 ACM 978-1-nnnn-nnnn-n/yy/mm...\$15.00
DOI: [http://dx.doi.org/10.1145/\(to come\)](http://dx.doi.org/10.1145/(to come))

```
let doc = Http.Request("http://api.owm.org/?q=NYC")
match JsonValue.Parse(doc) with
| Record(root) →
  match Map.find "main" root with
  | Record(main) →
    match Map.find "temp" main with
    | Number(num) → printfn "Lovely %f!" num
    | _ → failwith "Incorrect format"
  | _ → failwith "Incorrect format"
  | _ → failwith "Incorrect format"
```

The code assumes that the response has a particular shape described in the documentation. The root node must be a record with a `main` field, which has to be another record containing a numerical `temp` field representing the current temperature. When the shape is different, the code fails. While not immediately unsound, the code is prone to errors if strings are misspelled or incorrect shape assumed.

Using the JSON type provider from F# Data, we can write code with exactly the same functionality in two lines:

```
type W = JsonProvider<"http://api.owm.org/?q=NYC">
printfn "Lovely %f!" (W.GetSample().Main.Temp)
```

`JsonProvider<"...">` invokes a type provider [23] at compile-time with the URL as a sample. The type provider infers the structure of the response and provides a type with a `GetSample` method that returns a parsed JSON with nested properties `Main.Temp`, returning the temperature as a number.

In short, the *types* come from the sample *data*. In our experience, this technique is both practical and surprisingly effective in achieving more sound information interchange in heterogeneous systems. Our contributions are as follows:

- We present F# Data type providers for XML, CSV and JSON (§2) and practical aspects of their implementation that contributed to their industrial adoption (§6).
- We describe a predictable shape inference algorithm for structured data formats, based on a *preferred shape* relation, that underlies the type providers (§3).
- We give a formal model (§4) and use it to prove *relative type safety* for the type providers (§5).

¹ We abbreviate the full URL and omit application key (available after registration). The returned JSON is shown in Appendix A and can be used to run the code against a local file.

2. Type providers for structured data

We start with an informal overview that shows how F# Data type providers simplify working with JSON and XML. We introduce the necessary aspects of F# type providers along the way. The examples in this section also illustrate the key design principles of the shape inference algorithm:

- The mechanism is predictable (§6.5). The user directly works with the provided types and should understand why a specific type was produced from a given sample.
- The type providers prefer F# object types with properties. This allows extensible (open-world) data formats (§2.2) and it interacts well with developer tooling (§2.1).
- The above makes our techniques applicable to any language with nominal object types (e.g. variations of Java or C# with a type provider mechanism added).
- Finally, we handle practical concerns including support for different numerical types, `null` and missing data.

The supplementary screencast provides further illustration of the practical developer experience using F# Data.²

2.1 Working with JSON documents

The JSON format is a popular data exchange format based on JavaScript data structures. The following is the definition of `JsonValue` used earlier (§1) to represent JSON data:

```
type JsonValue =
    | Number of float
    | Boolean of bool
    | String of string
    | Record of Map<string, JsonValue>
    | Array of JsonValue[]
    | Null
```

The earlier example used only a nested record containing a number. To demonstrate other aspects of the JSON type provider, we look at an example that also involves an array:

```
[ { "name": "Jan", "age": 25 },
  { "name": "Tomas" },
  { "name": "Alexander", "age": 3.5 } ]
```

The standard way to print the names and ages would be to pattern match on the parsed `JsonValue`, check that the top-level node is a `Array` and iterate over the elements checking that each element is a `Record` with certain properties. We would throw an exception for values of an incorrect shape. As before, the code would specify field names as strings, which is error prone and can not be statically checked.

Assuming `people.json` is the above example and `data` is a string containing JSON of the same shape, we can write:

```
type People = JsonProvider<"people.json">
for item in People.Parse(data) do
    printf "%s " item.Name
    Option.iter (printf "(%f)") item.Age
```

We now use a local file as a sample for the type inference, but then processes data from another source. The code achieves a similar simplicity as when using dynamically typed languages, but it is statically type-checked.

Type providers. The notation `JsonProvider<"people.json">` passes a *static parameter* to the type provider. Static parameters are resolved at compile-time and have to be constant. The provider analyzes the sample and provides a type `People`. F# editors also execute the type provider at development-time and use the provided types for auto-completion on “.” and for background type-checking.

The `JsonProvider` uses a shape inference algorithm and provides the following F# types for the sample:

```
type Entity =
    member Name : string
    member Age : option<float>
type People =
    member GetSample : unit → Entity[]
    member Parse : string → Entity[]
```

The type `Entity` represents the person. The field `Name` is available for all sample values and is inferred as `string`. The field `Age` is marked as optional, because the value is missing in one sample. In F#, we use `Option.iter` to call the specified function (printing) only when an optional value is available. The two age values are an integer 25 and a float 3.5 and so the common inferred type is `float`. The names of the properties are normalized to follow standard F# naming conventions as discussed later (§6.3).

The type `People` has two methods for reading data. `GetSample` parses the sample used for the inference and `Parse` parses a JSON string. This lets us read data at runtime, provided that it has the same shape as the static sample.

Error handling. In addition to the structure of the types, the type provider also specifies the code of operations such as `item.Name`. The runtime behaviour is the same as in the earlier hand-written sample (§1) – a member access throws an exception if data does not have the expected shape.

Informally, the safety property (§5) states that if the inputs are compatible with one of the static samples (i.e. the samples are representative), then no exceptions will occur. In other words, we cannot avoid all failures, but we can prevent some. Moreover, if <http://openweathermap.org> changes the shape of the response, the code in §1 will not re-compile and the developer knows that the code needs to be corrected.

Objects with properties. The sample code is easy to write thanks to the fact that most F# editors provide auto-completion when “.” is typed (see the supplementary screencast). The developer does not need to examine the sample JSON file to see what fields are available. To support this scenario, our type providers map the inferred shapes to F# objects with (possibly optional) properties.

² Available at <http://tomasp.net/academic/papers/fsharp-data>.

This is demonstrated by the fact that Age becomes an optional member. An alternative is to provide two different record types (one with Name and one with Name and Age), but this would complicate the processing code. It is worth noting that languages with stronger tooling around pattern matching such as Idris [12] might have different preferences.

2.2 Processing XML documents

XML documents are formed by nested elements with attributes. We can view elements as records with a field for each attribute and an additional special field for the nested contents (which is a collection of elements).

Consider a simple extensible document format where a root element `<doc>` can contain a number of document elements, one of which is `<heading>` representing headings:

```
<doc>
  <heading>Working with JSON</heading>
  <p>Type providers make this easy.</p>
  <heading>Working with XML</heading>
  <p>Processing XML is as easy as JSON.</p>
  <image source="xml.png" />
</doc>
```

The F# Data library has been designed primarily to simplify reading of data. For example, say we want to print all headings in the document. The sample shows a part of the document structure (in particular the `<heading>` element), but it does not show all possible elements (say, `<table>`). Assuming the above document is `sample.xml`, we can write:

```
type Document = XmlProvider("sample.xml")
let root = Document.Load("p1di/another.xml")
for elem in root.Doc do
    Option.iter (printf "%s" elem.Heading
```

The example iterates over a collection of elements returned by `root.Doc`. The type of `elem` provides typed access to elements known statically from the sample and so we can write `elem.Heading`, which returns an optional string value.

Open world. By its nature, XML is extensible and the sample cannot include all possible nodes.³ This is the fundamental *open world assumption* about external data. Actual input might be an element about which nothing is known.

For this reason, we do not infer a closed choice between heading, paragraph and image. In the subsequent formalization, we introduce a *top shape* (§3.1) and extend it with labels capturing the statically known possibilities (§3.5). The *labelled top shape* is mapped to the following type:

```
type Element =
    member Heading : option<string>
    member Paragraph : option<string>
    member Image : option<Image>
```

Element is an abstract type with properties. It can represent the statically known elements, but it is not limited to them. For a table element, all three properties would return None.

Using a type with optional properties provides access to the elements known statically from the sample. However the user needs to explicitly handle the case when a value is not a statically known element. In object-oriented languages, the same could be done by providing a class hierarchy, but this loses the easy discoverability when “.” is typed.

The provided type is also consistent with our design principles, which prefers optional properties. The gain is that the provided types support both open-world data and developer tooling. It is also worth noting that our shape inference uses labelled top shapes only as the last resort (Lemma 1, §6.4).

2.3 Real-world JSON services

Throughout the introduction, we used data sets that demonstrate the typical problems frequent in the real-world (missing data, inconsistent encoding of primitive values and heterogeneous shapes). The government debt information returned by the World Bank⁴ includes all three:

```
[ { "pages": 5 },
  [ { "indicator": "GC.DOD.TOTL.GD.ZS",
      "date": "2012", "value": null },
    { "indicator": "GC.DOD.TOTL.GD.ZS",
      "date": "2010", "value": "35.14229" } ] ]
```

First, the field value is `null` for some records. Second, numbers in JSON can be represented as numeric literals (without quotes), but here, they are returned as string literals instead.⁵ Finally, the top-level element is a collection containing two values of different shape. The record contains meta-data with the total number of pages and the array contains the data. F# Data supports a concept of heterogeneous collection (outlined in §6.4) and provides the following type:

```
type Record =
    member Pages : int

type Item =
    member Date : int
    member Indicator : string
    member Value : option<float>

type WorldBank =
    member Record : Record
    member Array : Item[]
```

The inference for heterogeneous collections infers the multiplicities and shapes of nested elements. As there is exactly one record and one array, the provided type `WorldBank` exposes them as properties `Record` and `Array`.

In addition to type providers for JSON and XML, F# Data also implements a type provider for CSV (§6.2). We treat CSV files as lists of records (with field for each column) and so CSV is handled directly by our inference algorithm.

³ Even when the document structure is defined using XML Schema, documents may contain elements prefixed with other namespaces.

⁴ Available at <http://data.worldbank.org>

⁵ This is often used to avoid non-standard numerical types of JavaScript.

3. Shape inference for structured data

The shape inference algorithm for structured data is based on a shape preference relation. When inferring the shape, it infers the most specific shapes of individual values (CSV rows, JSON or XML nodes) and recursively finds a common shape of all child nodes or all sample documents.

We first define the shape of structured data σ . We use the term *shape* to distinguish shapes of data from programming language *types* τ (type providers generate the latter from the former). Next, we define the preference relation on shapes σ and describe the algorithm for finding a common shape.

The shape algebra and inference presented here is influenced by the design principles we outlined earlier and by the type definitions available in the F# language. The same principles apply to other languages, but details may differ, for example with respect to numerical types and missing data.

3.1 Inferred shapes

We distinguish between *non-nullable shapes* that always have a valid value (written as $\hat{\sigma}$) and *nullable shapes* that encompass missing and **null** values (written as σ). We write ν for record names and record field names.

$$\hat{\sigma} = \nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n, \rho_i \} \\ | \text{float} | \text{int} | \text{bool} | \text{string}$$

$$\sigma = \hat{\sigma} | \text{nullable}\langle \hat{\sigma} \rangle | [\sigma] | \text{any} | \text{null} | \perp$$

Non-nullable shapes include records (consisting of a name and fields with their shapes) and primitives. The row variables ρ_i are discussed below. Names of records arising from XML are the names of the XML elements. For JSON records we always use a single name \bullet . We assume that record fields can be freely reordered.

We include two numerical primitives, **int** for integers and **float** for floating-point numbers. The two are related by the preference relation and we prefer **int**.

Any non-nullable shape $\hat{\sigma}$ can be wrapped as $\text{nullable}\langle \hat{\sigma} \rangle$ to explicitly permit the **null** value. Type providers map **nullable** shapes to the F# option type. A collection $[\sigma]$ is also nullable and **null** values are treated as empty collections. This is motivated by the fact that a **null** collection is usually handled as an empty collection by client code. However there is a range of design alternatives (make collections non-nullable or treat **null** string as an empty string).

The shape **null** is inhabited by the **null** value (using an overloaded notation) and \perp is the bottom shape. The **any** shape is the top shape, but we later add labels for statically known alternative shapes (§3.5) as discussed earlier (§2.2).

During inference we use row-variables ρ_i [1] in record shapes to represent the flexibility arising from records in samples. For example, when a record $\text{Point} \{x \mapsto 3\}$ occurs in a sample, it may be combined with $\text{Point} \{x \mapsto 3, y \mapsto 4\}$ that contains more fields. The overall shape inferred must account for the fact that any extra fields are optional, giving an inferred shape $\text{Point} \{x : \text{int}, y : \text{nullable}\langle \text{int} \rangle\}$.

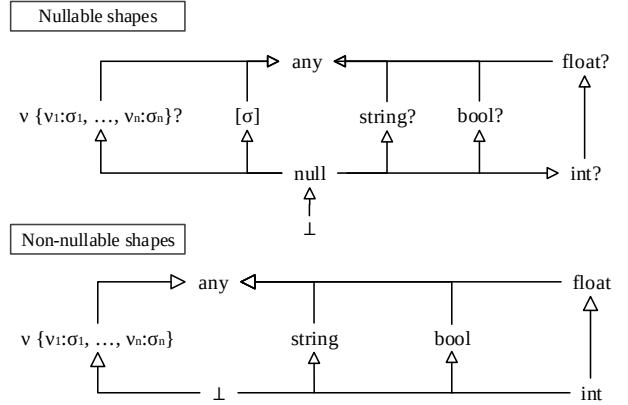


Figure 1. Important aspects of the preferred shape relation

3.2 Preferred shape relation

Figure 1 provides an intuition about the preference between shapes. The lower part shows non-nullable shapes (with records and primitives) and the upper part shows nullable shapes with **null**, collections and nullable shapes. In the diagram, we abbreviate $\text{nullable}\langle \sigma \rangle$ as $\sigma?$ and we omit links between the two parts; a shape $\hat{\sigma}$ is preferred over $\text{nullable}\langle \hat{\sigma} \rangle$.

Definition 1. For ground σ_1, σ_2 (i.e. without ρ_i variables), we write $\sigma_1 \sqsubseteq \sigma_2$ to denote that σ_1 is preferred over σ_2 . The shape preference relation is defined as a transitive reflexive closure of the following rules:

$$\text{int} \sqsubseteq \text{float} \quad (1)$$

$$\text{null} \sqsubseteq \sigma \quad (\text{for } \sigma \neq \hat{\sigma}) \quad (2)$$

$$\hat{\sigma} \sqsubseteq \text{nullable}\langle \hat{\sigma} \rangle \quad (\text{for all } \hat{\sigma}) \quad (3)$$

$$\text{nullable}\langle \hat{\sigma}_1 \rangle \sqsubseteq \text{nullable}\langle \hat{\sigma}_2 \rangle \quad (\text{if } \hat{\sigma}_1 \sqsubseteq \hat{\sigma}_2) \quad (4)$$

$$[\sigma_1] \sqsubseteq [\sigma_2] \quad (\text{if } \sigma_1 \sqsubseteq \sigma_2) \quad (5)$$

$$\perp \sqsubseteq \sigma \quad (\text{for all } \sigma) \quad (6)$$

$$\sigma \sqsubseteq \text{any} \quad (7)$$

$$\nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \} \sqsubseteq \nu \{ \nu_1 : \sigma'_1, \dots, \nu_n : \sigma'_n \} \quad (\text{if } \sigma_i \sqsubseteq \sigma'_i) \quad (8)$$

$$\nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \} \sqsubseteq \nu \{ \nu_1 : \sigma_1, \dots, \nu_m : \sigma_m \} \quad (\text{when } m \leq n) \quad (9)$$

Here is a summary of the key aspects of the definition:

- Numeric shape with smaller range is preferred (1) and we choose 32-bit **int** over **float** when possible.
- The **null** shape is preferred over all nullable shapes (2), i.e. all shapes excluding non-nullable shapes $\hat{\sigma}$. Any non-nullable shape is preferred over its nullable version (3)
- Nullable shapes and collections are covariant (4, 5).
- There is a bottom shape (6) and **any** behaves as the top shape, because any shape σ is preferred over **any** (7).
- The record shapes are covariant (8) and preferred record can have additional fields (9).

$$\begin{aligned}
\text{csh}(\sigma, \sigma) &= \sigma && (eq) \\
\text{csh}([\sigma_1], [\sigma_2]) &= [\text{csh}(\sigma_1, \sigma_2)] && (list) \\
\text{csh}(\perp, \sigma) = \text{csh}(\sigma, \perp) &= \sigma && (bot) \\
\text{csh}(\text{null}, \sigma) = \text{csh}(\sigma, \text{null}) &= \lceil \sigma \rceil && (null) \\
\text{csh}(\text{any}, \sigma) = \text{csh}(\sigma, \text{any}) &= \text{any} && (top) \\
\text{csh}(\text{float}, \text{int}) = \text{csh}(\text{int}, \text{float}) &= \text{float} && (num) \\
\text{csh}(\sigma_2, \text{nullable}\langle \hat{\sigma}_1 \rangle) = \text{csh}(\text{nullable}\langle \hat{\sigma}_1 \rangle, \sigma_2) &= \lceil \text{csh}(\hat{\sigma}_1, \sigma_2) \rceil && (opt) \\
\text{csh}(\nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \}, \nu \{ \nu_1 : \sigma'_1, \dots, \nu_n : \sigma'_n \}) &= \nu \{ \nu_1 : \text{csh}(\sigma_1, \sigma'_1), \dots, \nu_n : \text{csh}(\sigma_n, \sigma'_n) \} && (recd) \\
\text{csh}(\sigma_1, \sigma_2) &= \text{any} \quad (\text{when } \sigma_1 \neq \nu \{ \dots \} \text{ or } \sigma_2 \neq \nu \{ \dots \}) && (any)
\end{aligned}$$

$$\begin{aligned}
\lceil \hat{\sigma} \rceil &= \text{nullable}\langle \hat{\sigma} \rangle \quad (\text{non-nullable shapes}) && \lceil \text{nullable}\langle \hat{\sigma} \rangle \rceil &= \hat{\sigma} \quad (\text{nullable shape}) \\
\lceil \sigma \rceil &= \sigma \quad (\text{otherwise}) && \lceil \sigma \rceil &= \sigma \quad (\text{otherwise})
\end{aligned}$$

Figure 2. The rules that define the common preferred shape function

3.3 Common preferred shape relation

Given two ground shapes, the *common preferred shape* is the least upper bound of the shape with respect to the preferred shape relation. The least upper bound prefers records, which is important for usability as discussed earlier (§2.2).

Definition 2. A common preferred shape of two ground shapes σ_1 and σ_2 is a shape $\text{csh}(\sigma_1, \sigma_2)$ obtained according to Figure 2. The rules are matched from top to bottom.

The fact that the rules of csh are matched from top to bottom resolves the ambiguity between certain rules. Most importantly (*any*) is used only as the last resort.

When finding a common shape of two records (*recd*) we find common preferred shapes of their respective fields. We can find a common shape of two different numbers (*num*); for two collections, we combine their elements (*list*). When one shape is nullable (*opt*), we find the common non-nullable shape and ensure the result is nullable using $\lceil - \rceil$, which is also applied when one of the shapes is **null** (*null*).

When defined, csh finds the unique least upper bound of the partially ordered set of ground shapes (Lemma 1).

Lemma 1 (Least upper bound). For ground σ_1 and σ_2 , if $\text{csh}(\sigma_1, \sigma_2) \vdash \sigma$ then σ is a least upper bound by \sqsubseteq .

Proof. By induction over the structure of the shapes σ_1, σ_2 . Note that csh only infers the top shape **any** when one of the shapes is the top shape (*top*) or when there is no other option (*any*); a nullable shape is introduced in $\lceil - \rceil$ only when no non-nullable shape can be used (*null*), (*opt*). \square

3.4 Inferring shapes from samples

We now specify how we obtain the shape from data. As clarified later (§6.2), we represent JSON, XML and CSV documents using the same first-order *data* value:

$$d = i \mid f \mid s \mid \text{true} \mid \text{false} \mid \text{null} \mid [d_1; \dots; d_n] \mid \nu \{ \nu_1 \mapsto d_1, \dots, \nu_n \mapsto d_n \}$$

The definition includes primitive values (*i* for integers, *f* for floats and *s* for strings) and **null**. A collection is written as a list of values in square brackets. A record starts with a name ν , followed by a sequence of field assignments $\nu_i \mapsto d_i$.

Figure 3 defines a mapping $S(d_1, \dots, d_n)$ which turns a collection of sample data d_1, \dots, d_n into a shape σ . Before applying S , we assume each record in each d_i is marked with a fresh row inference variable ρ_i . We then choose a ground, minimal substitution θ for row variables. Because ρ_i variables represent potentially missing fields, the $\lceil - \rceil$ operator from Figure 2 is applied to all types in the vector.

This is sufficient to equate the record field labels and satisfy the pre-conditions in rule (*recd*) when multiple record shapes are combined. The csh function is not defined for two records with mis-matching fields, however, the fields can always be made to match, through a substitution for row variables. In practice, θ is found via row variable unification [17]. We omit the details here. No ρ_i variables remain after inference as the substitution chosen is ground.

Primitive values are mapped to their corresponding shapes. When inferring a shape from multiple samples, we use the common preferred shape relation to find a common shape for all values (starting with \perp). This operation is used when calling a type provider with multiple samples and also when inferring the shape of collection values.

$$\begin{aligned}
S(i) &= \text{int} & S(\text{null}) &= \text{null} & S(\text{true}) &= \text{bool} \\
S(f) &= \text{float} & S(s) &= \text{string} & S(\text{false}) &= \text{bool}
\end{aligned}$$

$$S([d_1; \dots; d_n]) = [S(d_1, \dots, d_n)]$$

$$S(\nu \{ \nu_1 \mapsto d_1, \dots, \nu_n \mapsto d_n \}_{\rho_i}) = \nu \{ \nu_1 : S(d_1), \dots, \nu_n : S(d_n), \lceil \theta(\rho_i) \rceil \}$$

$$S(d_1, \dots, d_n) = \sigma_n \quad \text{where} \\ \sigma_0 = \perp, \forall i \in \{1..n\}. \sigma_{i-1} \nabla S(d_i) \vdash \sigma_i$$

Choose minimal θ by ordering \sqsubseteq lifted over substitutions

Figure 3. Shape inference from sample data

tag = collection number nullable string ν any bool	tagof(string) = string tagof(bool) = bool tagof(int) = number tagof(float) = number	tagof(any $\langle\sigma_1, \dots, \sigma_n\rangle$) = any tagof($\nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \}$) = ν tagof(nullable $\langle\hat{\sigma}\rangle$) = nullable tagof($[\sigma]$) = collection
$\text{csh}(\text{any}\langle\sigma_1, \dots, \sigma_k, \dots, \sigma_n\rangle, \text{any}\langle\sigma'_1, \dots, \sigma'_k, \dots, \sigma'_m\rangle) =$ $\text{any}\langle\text{csh}(\sigma_1, \sigma'_1), \dots, \text{csh}(\sigma_k, \sigma'_k), \sigma_{k+1}, \dots, \sigma_n, \sigma'_{k+1}, \dots, \sigma'_m\rangle \quad (\text{top-merge})$ <p style="text-align: center;">For i, j such that $(\text{tagof}(\sigma_i) = \text{tagof}(\sigma'_j)) \Leftrightarrow (i = j) \wedge (i \leq k)$</p>		
$\text{csh}(\sigma, \text{any}\langle\sigma_1, \dots, \sigma_n\rangle) = \text{csh}(\text{any}\langle\sigma_1, \dots, \sigma_n\rangle, \sigma) =$ $\text{any}\langle\sigma_1, \dots, [\text{csh}(\sigma, \sigma_i)], \dots, \sigma_n\rangle \quad (\text{top-incl})$ <p style="text-align: center;">For i such that $\text{tagof}(\sigma_i) = \text{tagof}([\sigma])$</p>		
$\text{csh}(\sigma, \text{any}\langle\sigma_1, \dots, \sigma_n\rangle) = \text{any}\langle\sigma_1, \dots, \sigma_n, [\sigma]\rangle \quad (\text{top-add})$		
$\text{csh}(\sigma_1, \sigma_2) = \text{any}\langle[\sigma_1], [\sigma_2]\rangle \quad (\text{top-any})$		

Figure 4. Extending the common preferred shape relation for labelled top shapes

3.5 Adding labelled top shapes

When analyzing the structure of shapes, it suffices to consider a single top shape `any`. The type providers need more information to provide typed access to the possible alternative shapes of data, such as XML nodes.

We extend the core model (sufficient for the discussion of relative safety) with *labelled top shapes* defined as:

$$\sigma = \dots \mid \text{any}\langle\sigma_1, \dots, \sigma_n\rangle$$

The shapes $\sigma_1, \dots, \sigma_n$ represent statically known shapes that appear in the sample and that we expose in the provided type. As discussed earlier (§2.2) this is important when reading external *open world* data. The labels do not affect the preferred shape relation and `any` $\langle\sigma_1, \dots, \sigma_n\rangle$ should still be seen as the top shape, regardless of the labels⁶.

The common preferred shape function is extended to find a labelled top shape that best represents the sample. The new rules for `any` appear in Figure 4. We define shape *tags* to identify shapes that have a common preferred shape which is not the top shape. We use it to limit the number of labels and avoid nesting by grouping shapes by the shape tag. Rather than inferring `any` $\langle\text{int}, \text{any}\langle\text{bool}, \text{float}\rangle\rangle$, our algorithm joins `int` and `float` and produces `any` $\langle\text{float}, \text{bool}\rangle$.

When combining two top shapes (*top-merge*), we group the annotations by their tags. When combining a top with another shape, the labels may or may not already contain a case with the tag of the other shape. If they do, the two shapes are combined (*top-incl*), otherwise a new case is added (*top-add*). Finally, (*top-all*) replaces earlier (*any*) and combines two distinct non-top shapes. As top shapes implicitly permit `null` values, we make the labels non-nullable using `[-]`.

The revised algorithm still finds a shape which is the least upper bound. This means that labelled top shape is only inferred when there is no other alternative.

Stating properties of the labels requires refinements to the *preferred shape* relation. We leave the details to future work, but we note that the algorithm infers the best labels in the sense that there are labels that enable typed access to every possible value in the sample, but not more. The same is the case for nullable fields of records.

4. Formalizing type providers

This section presents the formal model of F# Data integration. To represent the programming language that hosts the type provider, we introduce the Foo calculus, a subset of F# with objects and properties, extended with operations for working with weakly typed structured data along the lines of the F# Data runtime. Finally, we describe how type providers turn inferred shapes into Foo classes (§4.2).

$$\tau = \text{int} \mid \text{float} \mid \text{bool} \mid \text{string} \mid C \mid \text{Data} \\ \mid \tau_1 \rightarrow \tau_2 \mid \text{list}\langle\tau\rangle \mid \text{option}\langle\tau\rangle$$

$$L = \text{type } C(\bar{x} : \bar{\tau}) = \bar{M}$$

$$M = \text{member } N : \tau = e$$

$$v = d \mid \text{None} \mid \text{Some}(v) \mid \text{new } C(\bar{v}) \mid v_1 :: v_2 \\ e = d \mid \text{op} \mid e_1 e_2 \mid \lambda x. e \mid e.N \mid \text{new } C(\bar{e}) \\ \mid \text{None} \mid \text{match } e \text{ with } \text{Some}(x) \rightarrow e_1 \mid \text{None} \rightarrow e_2 \\ \mid \text{Some}(e) \mid e_1 = e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{nil} \\ \mid e_1 :: e_2 \mid \text{match } e \text{ with } x_1 :: x_2 \rightarrow e_1 \mid \text{nil} \rightarrow e_2$$

$$\text{op} = \text{convFloat}(\sigma, e) \mid \text{convPrim}(\sigma, e) \\ \mid \text{convField}(\nu_1, \nu_2, e, e) \mid \text{convNull}(e_1, e_2) \\ \mid \text{convElements}(e_1, e_2) \mid \text{hasShape}(\sigma, e)$$

Figure 5. The syntax of the Foo calculus

⁶ An alternative would be to add unions of shapes, but doing so in a way that is compatible with the open-world assumption breaks the existence of unique lower bound of the preferred shape relation.

Part I. Reduction rules for conversion functions

$\text{hasShape}(\nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \}, \nu' \{ \nu'_1 \mapsto d_1, \dots, \nu'_m \mapsto d_m \}) \rightsquigarrow (\nu = \nu') \wedge$ $((\nu_1 = \nu'_1) \wedge \text{hasShape}(\sigma_1, d_1)) \vee \dots \vee ((\nu_1 = \nu'_m) \wedge \text{hasShape}(\sigma_1, d_m)) \vee \dots \vee$ $((\nu_n = \nu'_1) \wedge \text{hasShape}(\sigma_n, d_1)) \vee \dots \vee ((\nu_n = \nu'_m) \wedge \text{hasShape}(\sigma_n, d_m))$	$\text{convFloat}(\text{float}, i) \rightsquigarrow f \ (f = i)$ $\text{convFloat}(\text{float}, f) \rightsquigarrow f$
$\text{hasShape}([\sigma], [d_1; \dots; d_n]) \rightsquigarrow \text{hasShape}(\sigma, d_1) \wedge \dots \wedge \text{hasShape}(\sigma, d_n)$	$\text{convNull}(\text{null}, e) \rightsquigarrow \text{None}$ $\text{convNull}(d, e) \rightsquigarrow \text{Some}(e \ d)$
$\text{hasShape}([\sigma], \text{null}) \rightsquigarrow \text{true}$	
$\text{hasShape}(\text{string}, s) \rightsquigarrow \text{true}$	$\text{convPrim}(\sigma, d) \rightsquigarrow d \quad (\sigma, d \in \{(\text{int}, i), (\text{string}, s), (\text{bool}, b)\})$
$\text{hasShape}(\text{int}, i) \rightsquigarrow \text{true}$	$\text{convField}(\nu, \nu_i, \nu \{ \dots, \nu_i = d_i, \dots \}, e) \rightsquigarrow e \ d_i$
$\text{hasShape}(\text{bool}, d) \rightsquigarrow \text{true} \quad (\text{when } d \in \{\text{true}, \text{false}\})$	$\text{convField}(\nu, \nu', \nu \{ \dots, \nu_i = d_i, \dots \}, e) \rightsquigarrow e \ \text{null} \quad (\nexists i. \nu_i = \nu')$
$\text{hasShape}(\text{float}, d) \rightsquigarrow \text{true} \quad (\text{when } d = i \text{ or } d = f)$	$\text{convElements}([d_1; \dots; d_n], e) \rightsquigarrow e \ d_1 :: \dots :: e \ d_n :: \text{nil}$
$\text{hasShape}(_, _) \rightsquigarrow \text{false}$	$\text{convElements}(\text{null}) \rightsquigarrow \text{nil}$

Part II. Reduction rules for the rest of the Foo calculus

$\text{(member)} \quad \frac{\text{type } C(\bar{x} : \bar{\tau}) = \text{member } N_i : \tau_i = e_i \dots \in L}{L, (\text{new } C(\bar{v})).N_i \rightsquigarrow e_i[\bar{x} \leftarrow \bar{v}]}$	$\text{(match1)} \quad \text{match None with}$ $\text{Some}(x) \rightarrow e_1 \mid \text{None} \rightarrow e_2 \rightsquigarrow e_2$
$\text{(cond1)} \quad \text{if true then } e_1 \text{ else } e_2 \rightsquigarrow e_1$	$\text{(match2)} \quad \text{match Some}(v) \text{ with}$ $\text{Some}(x) \rightarrow e_1 \mid \text{None} \rightarrow e_2 \rightsquigarrow e_1[x \leftarrow v]$
$\text{(cond2)} \quad \text{if false then } e_1 \text{ else } e_2 \rightsquigarrow e_2$	$\text{(match3)} \quad \text{match nil with}$ $x_1 :: x_2 \rightarrow e_1 \mid \text{nil} \rightarrow e_2 \rightsquigarrow e_2$
$\text{(eq1)} \quad v = v' \rightsquigarrow \text{true} \quad (\text{when } v = v')$	$\text{(match4)} \quad \text{match } v_1 :: v_2 \text{ with}$ $x_1 :: x_2 \rightarrow e_1 \mid \text{nil} \rightarrow e_2 \rightsquigarrow e_1[\bar{x} \leftarrow \bar{v}]$
$\text{(eq2)} \quad v = v' \rightsquigarrow \text{false} \quad (\text{when } v \neq v')$	$\text{(ctx)} \quad E[e] \rightsquigarrow E[e'] \quad (\text{when } e \rightsquigarrow e')$
$\text{(fun)} \quad (\lambda x. e) v \rightsquigarrow e[x \leftarrow v]$	

Figure 6. Foo – Reduction rules for the Foo calculus and dynamic data operations

Type providers for structured data map the “dirty” world of weakly typed structured data into a “nice” world of strong types. To model this, the Foo calculus does not have `null` values and data values d are never directly exposed. Furthermore Foo is simply typed: despite using class types and object notation for notational convenience, it has no subtyping.

4.1 The Foo calculus

The syntax of the calculus is shown in Figure 5. The type `Data` is the type of structural data d . A class definition L consists of a single constructor and zero or more parameterless members. The declaration implicitly closes over the constructor parameters. Values v include previously defined data d ; expressions e include class construction, member access, usual functional constructs (functions, lists, options) and conditionals. The `op` constructs are discussed next.

Dynamic data operations. The Foo programs can only work with `Data` values using certain primitive operations. Those are modelled by the `op` primitives. In `F# Data`, those are internal and users never access them directly.

The behaviour of the dynamic data operations is defined by the reduction rules in Figure 6 (Part I). The typing is shown in Figure 7 and is discussed later. The `hasShape`

function represents a runtime shape test. It checks whether a `Data` value d (Section 3.4) passed as the second argument has a shape specified by the first argument. For records, we have to check that for each field ν_1, \dots, ν_n in the record, the actual record value has a field of the same name with a matching shape. The last line defines a “catch all” pattern, which returns `false` for all remaining cases. We treat $e_1 \vee e_2$ and $e_1 \wedge e_2$ as a syntactic sugar for `if..then..else` so the result of the reduction is just a Foo expression.

The remaining operations convert data values into values of less preferred shape. The `convPrim` and `convFloat` operations take the required shape and a data value. When the data does not match the required type, they do not reduce. For example, `convPrim(bool, 42)` represents a stuck state, but `convFloat(float, 42)` turns an integer `42` into a floating-point numerical value `42.0`.

The `convNull`, `convElements` and `convField` operations take an additional parameter e which represents a function to be used in order to convert a contained value (non-null optional value, list elements or field value); `convNull` turns `null` data value into `None` and `convElements` turns a data collection $[d_1, \dots, d_n]$ into a Foo list $v_1 :: \dots :: v_n :: \text{nil}$ and a `null` value into an empty list.

$L; \Gamma \vdash d : \text{Data}$	$L; \Gamma \vdash i : \text{int}$	$L; \Gamma \vdash f : \text{float}$	$\frac{L; \Gamma, x : \tau_1 \vdash e : \tau_2}{L; \Gamma \vdash \lambda x. e : \tau_2}$	$\frac{L; \Gamma \vdash e_2 : \tau_1 \quad L; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2}{L; \Gamma \vdash e_1 e_2 : \tau_2}$
$\frac{L; \Gamma \vdash e : \text{Data}}{L; \Gamma \vdash \text{hasShape}(\sigma, e) : \text{bool}}$	$\frac{L; \Gamma \vdash e : \text{Data} \quad \tau \in \{\text{int}, \text{float}\}}{L; \Gamma \vdash \text{convFloat}(\sigma, e) : \text{float}}$	$\frac{L; \Gamma \vdash e_1 : \text{Data} \quad L; \Gamma \vdash e_2 : \text{Data} \rightarrow \tau}{L; \Gamma \vdash \text{convNull}(e_1, e_2) : \text{option}\langle \tau \rangle}$		
$\frac{L; \Gamma \vdash e : \text{Data} \quad \text{prim} \in \{\text{int}, \text{string}, \text{bool}\}}{L; \Gamma \vdash \text{convPrim}(\text{prim}, e) : \text{prim}}$	$\frac{L; \Gamma \vdash e_1 : \text{Data} \quad L; \Gamma \vdash e_2 : \text{Data} \rightarrow \tau}{L; \Gamma \vdash \text{convElements}(e_1, e_2) : \text{list}\langle \tau \rangle}$	$\frac{L; \Gamma \vdash e_1 : \text{Data} \quad L; \Gamma \vdash e_2 : \text{Data} \rightarrow \tau}{L; \Gamma \vdash \text{convField}(v, v', e_1, e_2) : \tau}$		
$\frac{L; \Gamma \vdash e : C \quad \text{type } C(x : \bar{\tau}) = .. \text{member } N_i : \tau_i = e_i .. \in L}{L; \Gamma \vdash e.N_i : \tau_i}$		$\frac{L; \Gamma \vdash e_i : \tau_i \quad \text{type } C(x_1 : \tau_1, \dots, x_n : \tau_n) = \dots \in L}{L; \Gamma \vdash \text{new } C(e_1, \dots, e_n) : C}$		

Figure 7. Foo – Fragment of type checking

Reduction. The reduction relation is of the form $L, e \rightsquigarrow e'$. We omit class declarations L where implied by the context and write $e \rightsquigarrow^* e'$ for the reflexive, transitive closure of \rightsquigarrow .

Figure 6 (Part II) shows the reduction rules. The (*member*) rule reduces a member access using a class definition in the assumption. The (*ctx*) rule models the eager evaluation of F# and performs a reduction inside a sub-expression specified by an evaluation context E :

$$\begin{aligned}
E = & v :: E \mid v E \mid E.N \mid \text{new } C(\bar{v}, E, \bar{e}) \\
& \mid \text{if } E \text{ then } e_1 \text{ else } e_2 \mid E = e \mid v = E \\
& \mid \text{Some}(E) \mid \text{op}(\bar{v}, E, \bar{e}) \\
& \mid \text{match } E \text{ with } \text{Some}(x) \rightarrow e_1 \mid \text{None} \rightarrow e_2 \\
& \mid \text{match } E \text{ with } x_1 :: x_2 \rightarrow e_1 \mid \text{nil} \rightarrow e_2
\end{aligned}$$

The evaluation proceeds from left to right as denoted by \bar{v}, E, \bar{e} in constructor and dynamic data operation arguments or $v :: E$ in list initialization. We write $e[\bar{x} \leftarrow \bar{v}]$ for the result of replacing variables \bar{x} by values \bar{v} in an expression. The remaining six rules give standard reductions.

Type checking. Well-typed Foo programs reduce to a value in a finite number of steps or get stuck due to an error condition. The stuck states can only be due to the dynamic data operations (e.g. an attempt to convert `null` value to a number `convFloat(float, null)`). The relative safety (Theorem 3) characterizes the additional conditions on input data under which Foo programs do not get stuck.

Typing rules in Figure 7 are written using a judgement $L; \Gamma \vdash e : \tau$ where the context also contains a set of class declarations L . The fragment demonstrates the differences and similarities with Featherweight Java [10] and typing rules for the dynamic data operations *op*:

- All data values d have the type `Data`, but primitive data values (Booleans, strings, integers and floats) can be implicitly converted to Foo values and so they also have a primitive type as illustrated by the rule for i and f .
- For non-primitive data values (including `null`, data collections and records), `Data` is the only type.

- Operations *op* accept `Data` as one of the arguments and produce a non-`Data` Foo type. Some of them require a function specifying the conversion for nested values.
- Rules for checking class construction and member access are similar to corresponding rules of Featherweight Java.

An important part of Featherweight Java that is omitted here is the checking of type declarations (ensuring the members are well-typed). We consider only classes generated by our type providers and those are well-typed by construction.

4.2 Type providers

So far, we defined the type inference algorithm which produces a shape σ from one or more sample documents (§3) and we defined a simplified model of evaluation of F# (§4.1) and F# `Data` runtime (§4.2). In this section, we define how the type providers work, linking the two parts.

All F# `Data` type providers take (one or more) sample documents, infer a common preferred shape σ and then use it to generate F# types that are exposed to the programmer.⁷

Type provider mapping. A type provider produces an F# type τ together with a Foo expression and a collection of class definitions. We express it using the following mapping:

$$\llbracket \sigma \rrbracket = (\tau, e, L) \quad (\text{where } L, \emptyset \vdash e : \text{Data} \rightarrow \tau)$$

The mapping $\llbracket \sigma \rrbracket$ takes an inferred shape σ . It returns an F# type τ and a function that turns the input data (value of type `Data`) into a Foo value of type τ . The type provider also generates class definitions that may be used by e .

Figure 8 defines $\llbracket - \rrbracket$. Primitive types are handled by a single rule that inserts an appropriate conversion function; `convPrim` just checks that the shape matches and `convFloat` converts numbers to a floating-point.

⁷ The actual implementation provides *erased types* as described in [23]. Here, we treat the code as actually generated. This is an acceptable simplification, because F# `Data` type providers do not rely on laziness or erasure of type provision.

$$\begin{aligned}
\llbracket \sigma_p \rrbracket &= \tau_p, \lambda x.op(\sigma_p, x), \emptyset \quad \text{where} \\
&\sigma_p, \tau_p, op \in \{ (\text{bool}, \text{bool}, \text{convPrim}), \\
&\quad (\text{int}, \text{int}, \text{convPrim}), (\text{float}, \text{float}, \text{convFloat}), \\
&\quad (\text{string}, \text{string}, \text{convPrim}) \} \\
\llbracket \nu \{ \nu_1 : \sigma_1, \dots, \nu_n : \sigma_n \} \rrbracket &= \\
&C, \lambda x.new C(x), L_1 \cup \dots \cup L_n \cup \{L\} \quad \text{where} \\
&C \text{ is a fresh class name} \\
&L = \text{type } C(x_1 : \text{Data}) = M_1 \dots M_n \\
&M_i = \text{member } \nu_i : \tau_i = \text{convField}(\nu, \nu_i, x_1, e_i), \\
&\quad \tau_i, e_i, L_i = \llbracket \sigma_i \rrbracket \\
\llbracket [\sigma] \rrbracket &= \text{list} \langle \tau \rangle, \lambda x.convElements(x, e'), L \quad \text{where} \\
&\tau, e', L = \llbracket \hat{\sigma} \rrbracket \\
\llbracket \text{any} \langle \sigma_1, \dots, \sigma_n \rangle \rrbracket &= \\
&C, \lambda x.new C(x), L_1 \cup \dots \cup L_n \cup \{L\} \quad \text{where} \\
&C \text{ is a fresh class name} \\
&L = \text{type } C(x : \text{Data}) = M_1 \dots M_n \\
&M_i = \text{member } \nu_i : \text{option} \langle \tau_i \rangle = \\
&\quad \text{if hasShape}(\sigma_i, x) \text{ then } \text{Some}(e_i x) \text{ else } \text{None} \\
&\tau_i, e_i, L_i = \llbracket \sigma_i \rrbracket, \nu_i = \text{tagof}(\sigma_i) \\
\llbracket \text{nullable} \langle \hat{\sigma} \rangle \rrbracket &= \\
&\text{option} \langle \tau \rangle, \lambda x.convNull(x, e), L \\
&\text{where } \tau, e, L = \llbracket \hat{\sigma} \rrbracket \\
\llbracket \perp \rrbracket &= \llbracket \text{null} \rrbracket = C, \lambda x.new C(x), \{L\} \quad \text{where} \\
&C \text{ is a fresh class name} \\
&L = \text{type } C(v : \text{Data})
\end{aligned}$$

Figure 8. Type provider – generation of Foo types from inferred structural types

For records, we generate a class C that takes a data value as a constructor parameter. For each field, we generate a member with the same name as the field. The body of the member calls `convField` with a function obtained from $\llbracket \sigma_i \rrbracket$. This function turns the field value (data of shape σ_i) into a Foo value of type τ_i . The returned expression creates a new instance of C and the mapping returns the class C together with all recursively generated classes. Note that the class name C is not directly accessed by the user and so we can use an arbitrary name, although the actual implementation in F# Data attempts to infer a reasonable name.⁸

A collection shape becomes a Foo `list` $\langle \tau \rangle$. The returned expression calls `convElements` (which returns the empty list for data value `null`). The last parameter is the recursively obtained conversion function for the shape of elements σ . The handling of the nullable shape is similar, but uses `convNull`.

As discussed earlier, labelled top shapes are also generated as Foo classes with properties. Given $\text{any} \langle \sigma_1, \dots, \sigma_n \rangle$, we get corresponding F# types τ_i and generate n members of type `option` $\langle \tau_i \rangle$. When the member is accessed, we need to perform a runtime shape test using `hasShape` to ensure that the value has the right shape (similarly to runtime type conversions from the top type in languages like Java). If the shape matches, a `Some` value is returned. The shape inference algorithm also guarantees that there is only one case for each shape tag (§3.3) and so we can use the tag for the name of the generated member.

Example 1. To illustrate how the mechanism works, we consider two examples. First, assume that the inferred shape is a record `Person { Age : option <int>, Name : string }`. The rules from Figure 8 produce the `Person` class shown below with two members.

The body of the `Age` member uses `convField` as specified by the case for optional record fields. The field shape is nul-

lable and so `convNull` is used in the continuation to convert the value to `None` if `convField` produces a `null` data value and `hasShape` is used to ensure that the field has the correct shape. The `Name` value should be always available and should have the right shape so `convPrim` appears directly in the continuation. This is where the evaluation can get stuck if the field value was missing:

```

type Person(x1 : Data) =
  member Age : option<int> =
    convField(Person, Age, x1, λx2 →
      convNull(x2, λx3 → convPrim(int, x3)))
  member Name : string =
    convField(Person, Name, x1, λx2 →
      convPrim(string, x2))

```

The function to create the Foo value `Person` from a data value is `λx.new Person(x)`.

Example 2. The second example illustrates the handling of collections and labelled top types. Reusing `Person` from the previous example, consider $\text{any} \langle \text{Person} \{ \dots \}, \text{string} \rangle$:

```

type PersonOrString(x : Data) =
  member Person : option<Person> =
    if hasShape(Person { .. }, x) then
      Some(new Person(x)) else None
  member String : option<string> =
    if hasShape(string, x) then
      Some(convPrim(string, x)) else None

```

The type provider maps the collection of labelled top shapes to a type `list` $\langle \text{PersonOrString} \rangle$ and returns a function that parses a data value as follows:

⁸ For example, in `{ "person" : { "name" : "Tomas" } }`, the nested record will be named `Person` based on the name of the parent record field.

$\lambda x_1 \rightarrow \text{convElements}(x_1 \lambda x_2 \rightarrow \text{new PersonOrString}(x_2))$

The `PersonOrString` class contains one member for each of the labels. In the body, they check that the input data value has the correct shape using `hasShape`. This also implicitly handles `null` by returning `false`. As discussed earlier, labelled top types provide easy access to the known cases (string or Person), but they require a runtime shape check.

5. Relative type safety

Informally, the safety property for structural type providers states that, given representative sample documents, any code that can be written using the provided types is guaranteed to work. We call this *relative safety*, because we cannot avoid all errors. In particular, one can always provide an input that has a different structure than any of the samples. In this case, it is expected that the code will throw an exception in the implementation (or get stuck in our model).

More formally, given a set of sample documents, code using the provided type is guaranteed to work if the inferred shape of the input is preferred with respect to the shape of any of the samples. Going back to §3.2, this means that:

- Input can contain smaller numerical values (e.g., if a sample contains float, the input can contain an integer).
- Records in the input can have additional fields.
- Records in the input can have fewer fields than some of the records in the sample document, provided that the sample also contains records that do not have the field.
- When a labelled top type is inferred from the sample, the actual input can also contain any other value, which implements the open world assumption.

The following lemma states that the provided code (generated in Figure 8) works correctly on an input d' that is a subshape of d . More formally, the provided expression (with input d') can be reduced to a value and, if it is a class, all its members can also be reduced to values.

Lemma 2 (Correctness of provided types). *Given sample data d and an input data value d' such that $S(d') \sqsubseteq S(d)$ and provided type, expression and classes $\tau, e, L = \llbracket S(d) \rrbracket$, then $L, e \ d' \rightsquigarrow^* v$ and if τ is a class ($\tau = C$) then for all members N_i of the class C , it holds that $L, (e \ d').N_i \rightsquigarrow^* v$.*

Proof. By induction over the structure of $\llbracket - \rrbracket$. For primitives, the conversion functions accept all subshapes. For other cases, analyze the provided code to see that it can work on all subshapes (for example `convElements` works on `null` values, `convFloat` accepts an integer). Finally, for labelled top types, the `hasShape` operation is used to guaranteed the correct shape at runtime. \square

This shows that provided types are correct with respect to the preferred shape relation. Our key theorem states that, for any input which is a subshape the inferred shape and any

expression e , a well-typed program that uses the provided types does not “go wrong”. Using standard syntactic type safety [26], we prove type preservation (reduction does not change type) and progress (an expression can be reduced).

Theorem 3 (Relative safety). *Assume d_1, \dots, d_n are samples, $\sigma = S(d_1, \dots, d_n)$ is an inferred shape and $\tau, e, L = \llbracket \sigma \rrbracket$ are a type, expression and class definitions generated by a type provider.*

For all inputs d' such that $S(d') \sqsubseteq \sigma$ and all expressions e' (representing the user code) such that e' does not contain any of the dynamic data operations op and any Data values as sub-expressions and $L; y : \tau \vdash e' : \tau'$, it is the case that $L, e[y \leftarrow e' \ d'] \rightsquigarrow^ v$ for some value v and also $\emptyset; \vdash v : \tau'$.*

Proof. We discuss the two parts of the proof separately as type preservation (Lemma 4) and progress (Lemma 5). \square

Lemma 4 (Preservation). *Given the τ, e, L generated by a type provider as specified in the assumptions of Theorem 3, then if $L, \Gamma \vdash e : \tau$ and $L, e \rightsquigarrow^* e'$ then $\Gamma \vdash e' : \tau$.*

Proof. By induction over \rightsquigarrow . The cases for the ML subset of Foo are standard. For (*member*), we check that code generated by type providers in Figure 8 is well-typed. \square

The progress lemma states that evaluation of a well-typed program does not reach an undefined state. This is not a problem for the Standard ML [15] subset and object-oriented subset [10] of the calculus. The problematic part are the dynamic data operations (Figure 6, Part I). Given a data value (of type Data), the reduction can get stuck if the value does not have a structure required by a specific operation.

The Lemma 2 guarantees that this does not happen inside the provided type. We carefully state that we only consider expressions e' which “[do] not contain primitive operations op as sub-expressions”. This ensure that only the code generated by a type provider works directly with data values.

Lemma 5 (Progress). *Given the assumptions and definitions from Theorem 3, there exists e'' such that $e'[y \leftarrow e \ d'] \rightsquigarrow e''$.*

Proof. Proceed by induction over the typing derivation of $L; \emptyset \vdash e[y \leftarrow e' \ d'] : \tau'$. The cases for the ML subset are standard. For member access, we rely on Lemma 2. \square

6. Practical experience

The F# Data library has been widely adopted by users and is one of the most downloaded F# libraries.⁹ A practical demonstration of development using the library can be seen in an attached screencast and additional documentation can be found at <http://fsharp.github.io/FSharp.Data>.

In this section, we discuss our experience with the safety guarantees provided by the F# Data type providers and other notable aspects of the implementation.

⁹ At the time of writing, the library has over 125,000 downloads on NuGet (package repository), 1,844 commits and 44 contributors on GitHub.

6.1 Relative safety in practice

The *relative safety* property does not guarantee safety in the same way as traditional closed-world type safety, but it reflects the reality of programming with external data that is becoming increasingly important [16]. Type providers increase the safety of this kind of programming.

Representative samples. When choosing a representative sample document, the user does not need to provide a sample that represents all possible inputs. They merely need to provide a sample that is representative with respect to data they intend to access. This makes the task of choosing a representative sample easier.

Schema change. Type providers are invoked at compile-time. If the schema changes (so that inputs are no longer related to the shape of the sample used at compile-time), the program can fail at runtime and developers have to handle the exception. The same problem happens when using weakly-typed code with explicit failure cases.

F# Data can help discover such errors earlier. Our first example (§1) points the JSON type provider at a sample using a live URL. This has the advantage that a re-compilation fails when the sample changes, which is an indication that the program needs to be updated to reflect the change.

Richer data sources. In general, XML, CSV and JSON data sources without an explicit schema will necessarily require techniques akin to those we have shown. However, some data sources provide an explicit schema with versioning support. For those, a type provider that adapts automatically could be written, but we leave this for future work.

6.2 Parsing structured data

In our formalization, we treat XML, JSON and CSV uniformly as *data values*. With the addition of names for records (for XML nodes), the definition of structural values is rich enough to capture all three formats.¹⁰ However, parsing real-world data poses a number of practical issues.

Reading CSV data. When reading CSV data, we read each row as an unnamed record and return a collection of rows. One difference between JSON and CSV is that in CSV, the literals have no data types and so we also need to infer the shape of primitive values. For example:

```
Ozone, Temp, Date,      Autofilled
41,    67,    2012-05-01,  0
36.3,  72,    2012-05-02,  1
12.1,  74,    3 kveten,   0
17.5,  #N/A,  2012-05-04,  0
```

The value #N/A is commonly used to represent missing values in CSV and is treated as `null`. The Date column uses mixed formats and is inferred as string (we support many date formats and “May 3” would be parsed as date). More interestingly, we also infer Autofilled as Boolean, because the sample contains only 0 and 1. This is handled by adding a bit shape which is preferred of both int and bool.

Reading XML documents. Mapping XML documents to structural values is more interesting. For each node, we create a record. Attributes become record fields and the body becomes a field with a special name. For example:

```
<root id="1">
  <item>Hello!</item>
</root>
```

This XML becomes a record root with fields `id` and `•` for the body. The nested element contains only the `•` field with the inner text. As with CSV, we infer shape of primitive values:

```
root {id ↦ 1, • ↦ [item {• ↦ "Hello!"}]}
```

The XML type provider also includes an option to use *global inference*. In that case, the inference from values (§3.4) unifies the shapes of *all* records with the same name. This is useful because, for example, in XHTML all `<table>` elements will be treated as values of the same type.

6.3 Providing idiomatic F# types

In order to provide types that are easy to use and follow the F# coding guidelines, we perform a number of transformations on the provided types that simplify their structure and use more idiomatic naming of fields. For example, the type provided for the XML document in §6.2 is:

```
type Root =
  member Id : int
  member Item : string
```

To obtain the type signature, we used the type provider as defined in Figure 8 and applied three additional transformations and simplifications:

- When a class *C* contains a member `•`, which is a class with further members, the nested members are lifted into the class *C*. For example, the above type `Root` directly contains `Item` rather than containing a member `•` returning a class with a member `Item`.
- Remaining members named `•` in the provided classes (typically of primitive types) are renamed to `Value`.
- Class members are renamed to follow PascalCase naming convention, when a collision occurs, a number is appended to the end as in PascalCase2. The provided implementation preforms the lookup using the original name.

Our current implementation also adds an additional member to each class that returns the underlying JSON node (called `JsonValue`) or XML element (called `XElement`). Those return the standard .NET or F# representation of the value and can be used to dynamically access data not exposed by the type providers, such as textual values inside mixed-content XML elements.

¹⁰ The same mechanism has later been used by the HTML type provider (<http://fsharp.github.io/FSharp.Data/HtmlProvider.html>), which provides similarly easy access to data in HTML tables and lists.

6.4 Heterogeneous collections

When introducing type providers (§2.3), we mentioned how F# Data handles heterogeneous collections. This allows us to avoid inferring labelled top shapes in many common scenarios. In the earlier example, a sample collection contains a record (with `pages` field) and a nested collection with values.

Rather than storing a single shape for the collection elements as in $[\sigma]$, heterogeneous collections store multiple possible element shapes together with their *inferred multiplicity* (exactly one, zero or one, zero or more):

$$\begin{aligned} \psi &= 1? \mid 1 \mid * \\ \sigma &= \dots \mid [\sigma_1, \psi_1] \dots \mid [\sigma_n, \psi_n] \end{aligned}$$

We omit the details, but finding a preferred common shape of two heterogeneous collections is analogous to the handling of labelled top types. We merge cases with the same tag (by finding their common shape) and calculate their new shared multiplicity (for example, by turning 1 and 1? into 1?).

6.5 Predictability and stability

As discussed in §2, our inference algorithm is designed to be predictable and stable. When a user writes a program using the provided type and then adds another sample (e.g. with more missing values), they should not need to restructure their program. For this reason, we keep the algorithm simple. For example, we do not use probabilistic methods to assess the similarity of record types, because a small change in the sample could cause a large change in the provided types.

We leave a general theory of stability and predictability of type providers to future work, but we formalize a brief observation in this section. Say we write a program using a provided type that is based on a collection of samples. When a new sample is added, the program can be modified to run as before with only small local changes.

For the purpose of this section, assume that the Foo calculus also contains an `exn` value representing a runtime exception that propagates in the usual way, i.e. $C[\text{exn}] \rightsquigarrow \text{exn}$, and also a conversion function `int` that turns floating-point number into an integer.

Remark 1 (Stability of inference). *Assume we have a set of samples d_1, \dots, d_n , a provided type based on the samples $\tau_1, e_1, L_1 = \llbracket S(d_1, \dots, d_n) \rrbracket$ and some user code e written using the provided type, such that $L_1; x : \tau_1 \vdash e : \tau$.*

Next, we add a new sample d_{n+1} and consider a new provided type $\tau_2, e_2, L_2 = \llbracket S(d_1, \dots, d_n, d_{n+1}) \rrbracket$.

Now there exists e' such that $L_2; x : \tau_2 \vdash e' : \tau$ and if for some d it is the case that $e[x \leftarrow e_1 d] \rightsquigarrow v$ then also $e'[x \leftarrow e_2 d] \rightsquigarrow v$.

Such e' is obtained by transforming sub-expressions of e using one of the following translation rules:

- (i) $C[e]$ to $C[\text{match } e \text{ with Some}(v) \rightarrow v \mid \text{None} \rightarrow \text{exn}]$
- (ii) $C[e]$ to $C[e.M]$ where $M = \text{tagof}(\sigma)$ for some σ
- (iii) $C[e]$ to $C[\text{int}(e)]$

Proof. For each case in the type provision (Figure 8) an original shape σ may be replaced by a less preferred shape σ' . The user code can always be transformed to use the newly provided shape:

- Primitive shapes can become nullable (i), `int` can become `float` (iii) or become a part of a labelled top type (ii).
- Record shape fields can change shape (recursively) and record may become a part of a labelled top type (ii).
- For list and nullable shapes, the shape of the value may change (we apply the transformations recursively).
- For the `any` shape, the original code will continue to work (none of the labels is ever removed). \square

Intuitively, the first transformation is needed when the new sample makes a type optional. This happens when it contains a `null` value or a record that does not contain a field that all previous samples have. The second transformation is needed when a shape σ becomes `any` $\langle \sigma, \dots \rangle$ and the third one is needed when `int` becomes `float`.

This property also underlines a common way of handling errors when using F# Data type providers. When a program fails on some input, the input can be added as another sample. This makes some fields optional and the code can be updated accordingly, using a variation of (i) that uses an appropriate default value rather than throwing an exception.

7. Related and future work

The F# Data library connects two lines of research that have been previously disconnected. The first is extending the type systems of programming languages to accommodate external data sources and the second is inferring types for real-world data sources.

The type provider mechanism has been introduced in F# [23, 24], added to Idris [3] and used in areas such as semantic web [18]. The F# Data library has been developed as part of the early F# type provider research, but previous publications focused on the general mechanisms. This paper is novel in that it shows the programming language theory behind a concrete type providers.

Extending the type systems. Several systems integrate external data into a programming language. Those include XML [9, 21] and databases [5]. In both of these, the system requires the user to explicitly define the schema (using the host language) or it has an ad-hoc extension that reads the schema (e.g. from a database). LINQ [14] is more general, but relies on code generation when importing the schema.

The work that is the most similar to F# Data is the data integration in `C ω` [13]. It extends C# language with types similar to our structural types (including nullable types, choices with subtyping and heterogeneous collections with multiplicities). However, `C ω` does not infer the types from samples and extends the type system of the host language (rather than using a general purpose embedding mechanism).

In contrast, F# Data type providers do not require any F# language extensions. The simplicity of the Foo calculus shows we have avoided placing strong requirements on the host language. We provide nominal types based on the shapes, rather than adding an advanced system of structural types into the host language.

Advanced type systems and meta-programming. A number of other advanced type system features could be used to tackle the problem discussed in this paper. The Ur [2] language has a rich system for working with records; meta-programming [6, 19] and multi-stage programming [25] could be used to generate code for the provided types; and gradual typing [20, 22] can add typing to existing dynamic languages. As far as we are aware, none of these systems have been used to provide the same level of integration with XML, CSV and JSON.

Typing real-world data. Recent work [4] infers a succinct type of large JSON datasets using MapReduce. It fuses similar types based on similarity. This is more sophisticated than our technique, but it makes formal specification of safety (Theorem 3) difficult. Extending our *relative safety* to *probabilistic safety* is an interesting future direction.

The PADS project [7, 11] tackles a more general problem of handling *any* data format. The schema definitions in PADS are similar to our shapes. The structure inference for LearnPADS [8] infers the data format from a flat input stream. A PADS type provider could follow many of the patterns we explore in this paper, but formally specifying the safety property would be more challenging.

8. Conclusions

We explored the F# Data type providers for XML, CSV and JSON. As most real-world data does not come with an explicit schema, the library uses *shape inference* that deduces a shape from a set of samples. Our inference algorithm is based on a preferred shape relation. It prefers records to encompass the open world assumption and support developer tooling. The inference algorithm is predictable, which is important as developers need to understand how changing the samples affects the resulting types.

We explored the theory behind type providers. F# Data is a prime example of type providers, but our work demonstrates a more general point. The types generated by type providers can depend on external input and so we can only guarantee *relative safety*, which says that a program is safe only if the actual inputs satisfy additional conditions.

Type providers have been described before, but this paper is novel in that it explores the properties of type providers that represent the “types from data” approach. Our experience suggests that this significantly broadens the applicability of statically typed languages to real-world problems that are often solved by error-prone weakly-typed techniques.

Acknowledgments

We thank to the F# Data contributors on GitHub and other colleagues working on type providers, including Jomo Fisher, Keith Battocchi and Kenji Takeda. We are grateful to anonymous reviewers of the paper for their valuable feedback and to David Walker for shepherding of the paper.

References

- [1] L. Cardelli and J. C. Mitchell. Operations on Records. In *Mathematical Foundations of Programming Semantics*, pages 22–52. Springer, 1990.
- [2] A. Chlipala. Ur: Statically-typed Metaprogramming with Type-level Record Computation. In *ACM SIGPLAN Notices*, volume 45, pages 122–133. ACM, 2010.
- [3] D. R. Christiansen. Dependent Type Providers. In *Proceedings of Workshop on Generic Programming, WGP '13*, pages 25–34, 2013. ISBN 978-1-4503-2389-5.
- [4] D. Colazzo, G. Ghelli, and C. Sartiani. Typing Massive JSON Datasets. In *International Workshop on Cross-model Language Design and Implementation, XLDI '12*, 2012.
- [5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming without Tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.
- [6] J. Donham and N. Pouillard. Camlp4 and Template Haskell. In *Commercial Users of Functional Programming*, 2010.
- [7] K. Fisher and R. Gruber. PADS: A Domain-specific Language for Processing Ad Hoc Data. *ACM SIGPLAN Notices*, 40(6): 295–304, 2005.
- [8] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *Proceedings of ACM Symposium on Principles of Programming Languages, POPL '08*, pages 421–434, 2008. ISBN 978-1-59593-689-9.
- [9] H. Hosoya and B. C. Pierce. XDuce: A Statically Typed XML Processing Language. *Transactions on Internet Technology*, 3(2):117–148, 2003.
- [10] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM SIGPLAN Notices*, volume 34, pages 132–146. ACM, 1999.
- [11] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A Functional Data Description Language. In *ACM SIGPLAN Notices*, volume 42, pages 77–83. ACM, 2007.
- [12] H. Mehnert and D. Christiansen. Tool Demonstration: An IDE for Programming and Proving in Idris. In *Proceedings of Vienna Summer of Logic, VSL'14*, 2014.
- [13] E. Meijer, W. Schulte, and G. Bierman. Unifying Tables, Objects, and Documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, pages 145–166, 2003.
- [14] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the International Conference on Management of Data, SIGMOD '06*, pages 706–706, 2006.

- [15] R. Milner. *The Definition of Standard ML: Revised*. MIT press, 1997.
- [16] T. Petricek and D. Syme. In the Age of Web: Typed Functional-first Programming Revisited. *Post-Proceedings of ML Workshop*, 2015.
- [17] D. Rémy. *Type Inference for Records in a Natural Extension of ML*. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press, 1993.
- [18] S. Scheglmann, R. Lämmel, M. Leinberger, S. Staab, M. Thimm, and E. Viegas. IDE Integrated RDF Exploration, Access and RDF-Based Code Typing with LITEQ. In *The Semantic Web: ESWC 2014 Satellite Events*, pages 505–510. Springer, 2014.
- [19] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the ACM Workshop on Haskell*, pages 1–16. ACM, 2002.
- [20] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.
- [21] M. Sulzmann and K. Z. M. Lu. A Type-safe Embedding of XDuce into ML. *Electr. Notes in Theoretical Comp. Sci.*, 148 (2):239–264, 2006.
- [22] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual Typing Embedded Securely in JavaScript. In *ACM SIGPLAN Notices*, volume 49, pages 425–437. ACM, 2014.
- [23] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. Strongly-typed Language Support for Internet-scale Information Sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.
- [24] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in Information-rich Functional Programming for Internet-scale Data Sources. In *Proceedings of the Workshop on Data Driven Functional Programming, DDFP'13*, pages 1–4, 2013.
- [25] W. Taha and T. Sheard. Multi-stage Programming with Explicit Annotations. *ACM SIGPLAN Notices*, 32(12):203–217, 1997. ISSN 0362-1340.
- [26] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1):38–94, 1994.

A. OpenWeatherMap service response

The introduction uses the `JsonProvider` to access weather information using the OpenWeatherMap service. After registering, you can access the service using a URL <http://api.openweathermap.org/data/2.5/weather> with query string parameters `q` and `APPID` representing the city name and application key. A sample response looks as follows:

```
{
  "coord": {
    "lon": 14.42,
    "lat": 50.09
  },
  "weather": [
    {
      "id": 802,
      "main": "Clouds",
      "description": "scattered clouds",
      "icon": "03d"
    }
  ],
  "base": "cmc stations",
  "main": {
    "temp": 5,
    "pressure": 1010,
    "humidity": 100,
    "temp_min": 5,
    "temp_max": 5
  },
  "wind": { "speed": 1.5, "deg": 150 },
  "clouds": { "all": 32 },
  "dt": 1460700000,
  "sys": {
    "type": 1,
    "id": 5889,
    "message": 0.0033,
    "country": "CZ",
    "sunrise": 1460693287,
    "sunset": 1460743037
  },
  "id": 3067696,
  "name": "Prague",
  "cod": 200
}
```


Chapter 7

Data exploration through dot-driven development

Tomas Petricek. 2017. Data Exploration through Dot-driven Development. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:27. <https://doi.org/10.4230/LIPICS.ECOOP.2017.21>

Data exploration through dot-driven development*

Tomas Petricek¹

1 The Alan Turing Institute, London, UK
and Microsoft Research, Cambridge, 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.ECOOP.2017.55

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 [11] suggests that there has never been a greater need for increasing data literacy and tools that let anyone 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 [17, 25]. 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 The Alan Turing Institute under the EPSRC grant EP/N510129/1 and by the Google Digital News Initiative.



© Tomas Petricek;

licensed under Creative Commons License CC-BY

31st European Conference on Object-Oriented Programming (ECOOP 2017).

Editor: Peter Müller; Article No. 55; pp. 55:1–55:27



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```

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

```

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

The language presented in this paper reduces the number of language features by making member access the primary programming mechanism. Finding top 8 athletes by the number of gold medals from Rio 2016 can be written as:

```

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 data exploration through single programming mechanism (Section 3.2).
- Our mechanism illustrates how to embed “fancy types” [37] 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 (the-gamma.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 [34]. 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 [27] 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 [26] 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 [26], 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` [21] 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.

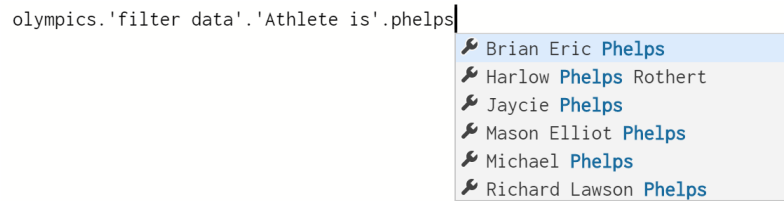
- The filtering operation is written using indexing `[...]` while all other operations are written using member invocation with (optionally named) parameters. In the first case, we write an expression `olympics["Games"] == "Rio (2016)"` returning a vector of Booleans while in the other, we specify a 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 [22] 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 example¹. Statically-typed languages provide better tooling, but at the cost of higher complexity².
- In the Python example (as well as in most other data manipulation libraries), column names are specified as strings³. 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 [9] tracks column names statically, arguably at the cost of higher code complexity when compared with Python.



■ **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»	«group data»
→ «drop Athlete»	→ «by Athlete»
→ «drop Discipline»	→ «average Year»
→ «drop Year»	→ «sum Year»
«sort data»	→ «by Year»
→ «by Athlete»	→ «distinct Athlete»
→ «by Athlete descending»	→ «concat Athlete»
→ «by Discipline»	→ «distinct Discipline»
→ «by Discipline descending»	→ «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 language⁴. 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 [34] 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 [26] 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.

Figure 2 lists a subset of the members from the example in Section 1. After choosing «sort data», the user is offered the possible sorting keys. 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 development⁵.

Our language supports method calls and so some of the operations that are currently exposed as member access could equally be provided as methods. For example, filtering could be written as «Games is»("Rio (2016)"). However, the fact that we can offer possible values when filtering 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 [26], 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 distinct team names ❶. Then we sort the grouped data using two 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 [34] 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#, renaming of columns could be encoded using methods with static parameters [33] 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 [35]. Row types make it possible to statically track operations on records that add or remove fields and so they can be used for the typing of operations such as «drop Year». In addition, we need to annotate type with a form of tpestate [32] 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.

$$\begin{array}{c}
(\textit{drop-start}) \frac{\Gamma \vdash e : [f_1 : \tau_1, \dots, f_n : \tau_n]}{\Gamma \vdash e. \langle \textit{drop columns} \rangle : [f_1 : \tau_1, \dots, f_n : \tau_n]_{\textit{drop}}} \\
\\
(\textit{drop-col}) \frac{\Gamma \vdash e : [f_1 : \tau_1, \dots, f_n : \tau_n]_{\textit{drop}}}{\Gamma \vdash e. \langle \textit{drop } f_i \rangle : [f_1 : \tau_1, \dots, f_{i-1} : \tau_{i-1}, f_{i+1} : \tau_{i+1}, \dots, f_n : \tau_n]_{\textit{drop}}} \\
\\
(\textit{drop-then}) \frac{\Gamma \vdash e : [f_1 : \tau_1, \dots, f_n : \tau_n]_{\textit{drop}}}{\Gamma \vdash e. \langle \textit{then} \rangle : [f_1 : \tau_1, \dots, f_n : \tau_n]}
\end{array}$$

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

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 :

$$\begin{array}{l}
e = v \mid e.N \\
\tau = \alpha \mid \tau_{\textit{lbl}} \mid [f_1 : \tau_1, \dots, f_n : \tau_n]
\end{array}$$

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:

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

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:

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

The rule, adapted from [26], 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.

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

■ **Figure 4** Relational algebra with values, sorting and aggregation

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:

► **Remark 1** (Encoding of fancy types). If $\Gamma \vdash e : [f_1 : \tau_1, \dots, f_n : \tau_n]$ 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 $\text{fields}(C) = \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\}$.

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 [8, 24]. 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.

$$\begin{aligned}
& \Pi_{f_{p(1)}, \dots, f_{p(m)}} \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,r} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,r} \rangle\} \rightsquigarrow \\
& \quad \{f_{p(1)} \mapsto \langle v_{p(1),1}, \dots, v_{p(1),r} \rangle, \dots, f_{p(m)} \mapsto \langle v_{p(m),1}, \dots, v_{p(m),r} \rangle\} \\
& \sigma_\varphi \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,r} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,r} \rangle\} \rightsquigarrow \\
& \quad \{f_1 \mapsto \langle \dots, v_{1,j}, \dots \rangle, \dots, f_n \mapsto \langle \dots, v_{n,j}, \dots \rangle\} \quad (\forall j. \varphi \{f_1 \mapsto v_{1,j}, \dots, f_n \mapsto v_{n,j}\}) \\
& \tau_{f_{p(1)} \mapsto \omega_1, \dots, f_{p(m)} \mapsto \omega_m} \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,r} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,r} \rangle\} \rightsquigarrow \\
& \quad \{f_1 \mapsto \langle v_{1,q(1)}, \dots, v_{1,q(r)} \rangle, \dots, f_n \mapsto \langle v_{n,q(1)}, \dots, v_{n,q(r)} \rangle\} \quad \text{where } q \text{ permutation} \\
& \quad \text{such that } \forall i, j. i \leq j \implies (u_{1,i}, \dots, v_{m,i}) \leq (v_{1,j}, \dots, v_{m,j}) \text{ where} \\
& \quad \quad u_{k,l} = v_{p(k),q(l)} \quad (\text{when } \omega_k = \text{asc}) \\
& \quad \quad u_{k,l} = -v_{p(k),q(l)} \quad (\text{when } \omega_k = \text{desc}) \\
& \Phi_{f_i, \rho_1 / f'_1, \dots, \rho_m / f'_m} \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,r} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,r} \rangle\} \rightsquigarrow \\
& \quad \{f'_1 \mapsto a_1, \dots, f'_m \mapsto a_m, f_i \mapsto b\} \quad \text{where} \\
& \quad \{g_1, \dots, g_s\} = \{\{l \mid k \in 1 \dots r, v_{i,l} = v_{i,k}\}, l \in 1 \dots r\} \\
& \quad \quad b = \langle v_{i,k_1}, \dots, v_{i,k_s} \rangle \quad \text{where } k_j \in g_j \\
& \quad \quad a_i = \langle |g_1|, \dots, |g_s| \rangle \quad \text{when } \rho_i = \text{count} \\
& \quad \quad a_i = \langle \sum_{k \in g_1} v_{j,k}, \dots, \sum_{k \in g_s} v_{j,k} \rangle \quad \text{when } \rho_i = \text{sum } f_j \\
& \quad \quad a_i = \langle \prod_{k \in g_1} v_{j,k}, \dots, \prod_{k \in g_s} v_{j,k} \rangle \quad \text{when } \rho_i = \text{conc } f_j \\
& \quad \quad a_i = \langle |\{v_{j,k} \mid k \in g_1\}|, \dots, |\{v_{j,k} \mid k \in g_s\}| \rangle \quad \text{when } \rho_i = \text{dist } f_j
\end{aligned}$$

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

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 $f_{p(1)}, \dots, f_{p(m)}$ to refer to a selection of fields from f_1, \dots, f_n . Assuming $m \leq n$, p can be seen as a mapping from $\{1 \dots m\}$ to a subset of $\{1 \dots n\}$. 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 ρ_1, \dots, ρ_m . To do this, we first obtain a set of groups g_1, \dots, 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 [26]. 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

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

$$\begin{aligned}
(\text{member}) \quad & \frac{L(C) = (\text{type } C(x : \tau) = \dots \text{member } N_i : \tau_i = e_i \dots), L'}{(C(v)).N_i \rightsquigarrow_L e_i[x \leftarrow v]} \\
(\text{context}) \quad & \frac{e \rightsquigarrow_L e'}{E[e] \rightsquigarrow_L E[e']}
\end{aligned}$$

■ **Figure 6** Syntax and remaining reduction rules of the Foo calculus

using the relational algebra. The type $\text{series}\langle\tau_1, \tau_2\rangle$ 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.

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 *(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 [34]. 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. This aspect of type providers have been omitted in previous work [26, 19], but it is necessary for the pivot type provider. The typing rules given are written as:

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

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 .

$$\begin{array}{c}
\text{(num)} \frac{}{L; \Gamma \vdash n : \mathbf{num}; L} \quad \text{(string)} \frac{}{L; \Gamma \vdash s : \mathbf{string}; L} \quad \text{(var)} \frac{}{L; \Gamma, x : \tau \vdash x : \tau; L} \\
\\
\text{(data)} \frac{L; \Gamma \vdash v_{i,j} : \tau; L \quad \tau \in \{\mathbf{num}, \mathbf{string}\}}{L; \Gamma \vdash \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,r} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,r} \rangle\} : \mathbf{Query}; L} \\
\\
\text{(proj)} \frac{L_1; \Gamma \vdash e : \mathbf{Query}; L_2}{L_1; \Gamma \vdash \Pi_{f_1, \dots, f_n}(e) : \mathbf{Query}; L_2} \quad \text{(sort)} \frac{L_1; \Gamma \vdash e : \mathbf{Query}; L_2}{L_1; \Gamma \vdash \tau_{f_1, \dots, f_n}(e) : \mathbf{Query}; L_2} \\
\\
\text{(sel)} \frac{L_1; \Gamma \vdash e : \mathbf{Query}; L_2}{L_1; \Gamma \vdash \sigma_\varphi(e) : \mathbf{Query}; L_2} \quad \text{(group)} \frac{L_1; \Gamma \vdash e : \mathbf{Query}; L_2}{L_1; \Gamma \vdash \Phi_{f, \rho_1/f_1, \dots, \rho_n/f_n}(e) : \mathbf{Query}; L_2} \\
\\
\text{(series)} \frac{L_1; \Gamma \vdash e : \mathbf{Query}; L_2}{L_1; \Gamma \vdash \mathbf{series}\langle \tau_1, \tau_2 \rangle(e) : \mathbf{series}\langle \tau_1, \tau_2 \rangle; L_2} \\
\\
\text{(new)} \frac{L_1; \Gamma \vdash e : \tau, L_2 \quad L_2(C) = (\mathbf{type} \ C(x : \tau) = \dots), L}{L_1; \Gamma \vdash C(e) : C; L_2 \cup L} \\
\\
\text{(member)} \frac{L_1; \Gamma \vdash e : C; L_2 \quad L_2 \cup L; \Gamma, x : \tau \vdash e_i : \tau_i; L_3 \quad L_2(C) = (\mathbf{type} \ C(x : \tau) = \dots \mathbf{member} \ N_i : \tau_i = e_i \dots), L}{L_1; \Gamma \vdash e.N_i : \tau_i; L_3}
\end{array}$$

■ **Figure 7** Type-checking of Foo expressions with lazy context

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:

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

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 [31]. When provided by the pivot type provider, the operation produces series with values of correct types.

$$\begin{aligned}
\text{pivot}(F) &= C, \{C \mapsto (l, L_1 \cup \dots \cup L_4)\} & \textcircled{1} \\
l &= \text{type } C(x : \text{Query}) = \\
&\quad \text{member } \langle \text{drop columns} \rangle : C_1 = C_1(x) && \text{where } C_1, L_1 = \text{drop}(F) \\
&\quad \text{member } \langle \text{sort data} \rangle : C_2 = C_2(x) && \text{where } C_2, L_2 = \text{sort}(F) \\
&\quad \text{member } \langle \text{group data} \rangle : C_3 = C_3(x) && \text{where } C_3, L_3 = \text{group}(F) \\
&\quad \text{member } \langle \text{get series} \rangle : C_4 = C_4(x) && \text{where } C_4, L_4 = \text{get-key}(F) \\
\\
\text{get-key}(F) &= C, \{C \mapsto (l, \bigcup L_f)\} & \textcircled{2} \\
l &= \text{type } C(x : \text{Query}) = && \forall f \in \text{dom}(F) \text{ where} \\
&\quad \text{member } \langle \text{with key } f \rangle : C_f = C_f(x) && C_f, L_f = \text{get-val}(F, f) \\
\\
\text{get-val}(F, f_k) &= C, \{C \mapsto (l, \{\})\} & \textcircled{3} \\
l &= \text{type } C(x : \text{Query}) = && \forall f \in \text{dom}(F) \setminus \{f_k\} \text{ where} \\
&\quad \text{member } \langle \text{and value } f \rangle : \text{series}\langle \tau_k, \tau_v \rangle = && \tau_k = F(f_k), \tau_v = F(f) \\
&\quad \text{series}\langle \tau_k, \tau_v \rangle(\prod_{f_k, f}(x)) && \textcircled{4}
\end{aligned}$$

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

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. [26], 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.

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:

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

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 $\textcircled{1}$ 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

$$\begin{aligned}
\text{drop}(F) &= C, \{C \mapsto (l, L' \cup \bigcup L_f)\} && \textcircled{1} \\
l = \text{type } C(x : \text{Query}) &= && \forall f \in \text{dom}(F) \text{ where } C_f, L_f = \text{drop}(F') \\
\text{member } \llbracket \text{drop } f \rrbracket : C_f &= C_f(\Pi_{\text{dom}(F')}(x)) && \text{and } F' = \{f' \mapsto \tau' \in F, f' \neq f\} \\
\text{member then} : C' &= C'(x) && \textcircled{2} \quad \text{where } C', L' = \text{pivot}(F) \\
\\
\text{sort}(F) &= C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f)\} && \textcircled{3} \\
l = \text{type } C(x : \text{Query}) &= && \forall f \in \{f \mid F(f) = \text{num}\}, \text{ where} \\
\text{member } \llbracket \text{by } f \text{ desc} \rrbracket : C_f &= C_f(x) && C_f, L_f = \text{sort-and}(F, \langle f \mapsto \text{desc} \rangle) \\
\text{member } \llbracket \text{by } f \text{ asc} \rrbracket : C'_f &= C'_f(x) && C'_f, L'_f = \text{sort-and}(F, \langle f \mapsto \text{asc} \rangle) \\
\\
\text{sort-and}(F, \langle s_1, \dots, s_n \rangle) &= C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f \cup L')\} && \textcircled{4} \\
l = \text{type } C(x : \text{Query}) &= && \forall f \in \{f \mid F(f) = \text{num}, \nexists i. s_i = f' \mapsto \omega \wedge f' = f\} \\
\text{member } \llbracket f \text{ desc} \rrbracket : C_f &= C_f(x) && C_f, L_f = \text{sort-and}(F, \langle s_1, \dots, s_n, f \mapsto \text{desc} \rangle) \\
\text{member } \llbracket f \text{ asc} \rrbracket : C'_f &= C'_f(x) && C'_f, L'_f = \text{sort-and}(F, \langle s_1, \dots, s_n, f \mapsto \text{asc} \rangle) \\
\text{member then} : C' &= C'(\tau_{s_1, \dots, s_n}(x)) && \text{where } C', L' = \text{pivot}(F) \quad \textcircled{5}
\end{aligned}$$

■ **Figure 9** Pivot type provider – dropping columns and sorting data

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 $\textcircled{2}$ and get-val $\textcircled{3}$ 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 $\text{series}\langle\tau_1, \tau_2\rangle$. For example, the following expression has a type $\text{series}\langle\text{string}, \text{num}\rangle$:

`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 $\Pi_{f_k, f}$ $\textcircled{4}$. 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 $\textcircled{1}$ 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 $\textcircled{2}$, 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 $\Pi_{\text{dom}(F')}$.

Types for defining the sorting transformation are split between two functions; sort $\textcircled{3}$ generates type for choosing the first sorting key and sort-and $\textcircled{4}$ lets the user add more keys. For space reasons, we abbreviate ascending and descending as asc and desc in the generated member names and we omit and in name of further keys such as «and Gold descending».

$$\begin{aligned}
\text{group}(F) &= C, \{C \mapsto (l, \bigcup L_f)\} \quad \textcircled{1} \\
l &= \text{type } C(x : \text{Query}) = & \forall f \in \text{dom}(F) \text{ where} \\
&\quad \text{member } \llbracket \text{by } f \rrbracket : C_f = C_f(x) & C_f, L_f = \text{agg}(F, f, \{f \mapsto F(f)\}, \emptyset) \quad \textcircled{2} \\
\\
\text{agg}(F, f, G, S) &= C, \{C \mapsto (l, \bigcup L_f \cup \bigcup L'_f \cup \bigcup L''_f \cup L' \cup L'')\} \quad \textcircled{3} \\
l &= \text{type } C(x : \text{Query}) = & \forall f \in \text{dom}(F) \setminus \text{dom}(S) \\
&\quad \text{member } \llbracket \text{sum } f \rrbracket : C'_f = C'_f(x) & \text{when } F(f) = \text{num} \quad \textcircled{4} \\
&\quad \text{member } \llbracket \text{concat } f \rrbracket : C''_f = C''_f(x) & \text{when } F(f) = \text{string} \quad \textcircled{5} \\
&\quad \text{member } \llbracket \text{count all} \rrbracket : C' = C'(x) & \text{when } \text{Count} \notin G \quad \textcircled{6} \\
&\quad \text{member } \llbracket \text{distinct } f \rrbracket : C_f = C_f(x) \\
&\quad \text{member } \text{then} : C'' = C''(\Phi_{f, \rho_1/f_1, \dots, \rho_n/f_n}(x)) \text{ where } \{\rho_1/f_1, \dots, \rho_n/f_n\} = S \quad \textcircled{7}
\end{aligned}$$

where

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

■ **Figure 10** Pivot type provider – grouping and aggregation

The members are restricted to numerical columns (by checking $F(f) = \text{num}$). 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 $\textcircled{5}$ 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 $\textcircled{1}$ 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 $\textcircled{3}$ 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 $\textcircled{2}$, 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 $\textcircled{7}$, 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, $\llbracket \text{sum } f \rrbracket$ $\textcircled{4}$ is only provided for columns of type **num** and $\llbracket \text{concat } f \rrbracket$ $\textcircled{5}$ is only provided for strings. Finally, the $\llbracket \text{count all} \rrbracket$ aggregation $\textcircled{6}$ 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: $E[\Pi_{f_1, \dots, f_n}(D)]$, $E[\sigma_\varphi(D)]$, $\tau_{f_1, \dots, f_n}(D)$ or $E[\Phi_{f, \rho_1/f_1, \dots, \rho_m/f_m}(D)]$ 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: $E[\Pi_{f_1, \dots, f_n}(D)]$, $E[\sigma_\varphi(D)]$, $\tau_{f_1, \dots, f_n}(D)$ or $E[\Phi_{f, \rho_1/f_1, \dots, \rho_m/f_m}(D)]$ 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 [7]), 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 $\downarrow_n L = \{C_0 \mapsto (l_0, L_0), \dots, C_n \mapsto (l_n, L_n)\}$ and $\downarrow_0 L = L$.

► **Theorem 5** (Correctness of lazy contexts). *Given $C, L = \text{pivot}(F)$ 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 $C_0, L_0 = \text{pivot}(F)$ and assume that $C \in \text{dom}(\downarrow_n L)$ where $\downarrow_n L(C) = (\text{type } C(x : \tau) = .. \text{member } N_i : \tau_i = e_i ..), L'$. 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 $F = \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\}$, let $C, L = \text{pivot}(F)$ then for any expression e that does not contain relational algebra operations or Query-typed values as sub-expression, if $L; x : C \vdash e : \text{series}\langle\tau_1, \tau_2\rangle; L'$ then for all $D = \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,m} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,m} \rangle\}$ such that $\vdash v_{i,j} : \tau_i$ it holds that $e[x \leftarrow C(D)] \rightsquigarrow_{L'}^* \text{series}\langle\tau_k, \tau_v\rangle(\{f_k \mapsto k_1, \dots, k_r, f_v \mapsto v_1, \dots, v_r\})$ such that for all $j \vdash k_j : \tau_k$ and $\vdash v_j : \tau_v$.*

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

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

By induction over $\rightsquigarrow_{L'}$, show that when $C(v).N_i$ is reduced using (*member*) then v is a value $\{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,m} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,m} \rangle\}$ s.t. $\text{fields}(C) = \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\}$ 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)**».

$$\begin{aligned}
\text{pivot}(F, D) &= C, \{C \mapsto (l, L_1 \cup L_2 \cup \dots)\} \quad \textcircled{1} \\
l &= \text{type } C(x : \text{Query}) = \\
&\quad \text{member } \langle \text{drop columns} \rangle : C_1 = C_1(x) \quad \text{where } C_1, L_1 = \text{drop}(F, D) \\
&\quad \text{member } \langle \text{filter data} \rangle : C_2 = C_2(x) \quad \text{where } C_2, L_2 = \text{filter}(F, D) \\
&\quad (\dots) \\
\text{drop}(F, D) &= C, \{C \mapsto (l, L' \cup \bigcup L_f)\} \quad \textcircled{2} \\
l &= \text{type } C(x : \text{Query}) = \\
&\quad \text{member } \langle \text{drop } f \rangle : C_f = \\
&\quad \quad C_f(\Pi_{\text{dom}(F')}(x)) \quad \forall f \in \text{dom}(F) \\
&\quad \quad \text{where } F' = \{f' \mapsto \tau' \in F, f' \neq f\} \\
&\quad \quad \text{and } C_f, L_f = \text{drop}(F', \Pi_{\text{dom}(F')}(D)) \quad \textcircled{3} \\
&\quad \text{member } \text{then} : C' = C'(x) \quad \text{where } C', L' = \text{pivot}(F, D) \\
\text{filter}(F, D) &= C, \{C \mapsto (l, L' \cup \bigcup L_f)\} \\
l &= \text{type } C(x : \text{Query}) = \\
&\quad \text{member } \langle f \text{ is} \rangle : C_f = C_f(x) \quad \forall f \in \text{dom}(F) \\
&\quad \quad \text{where } C_f, L_f = \text{filter-val}(F, f, D) \quad \textcircled{4} \\
&\quad \text{member } \text{then} : C' = C'(x) \quad \text{where } C', L' = \text{pivot}(F, D) \\
\text{filter-val}(F, f, D) &= C, \{C \mapsto (l, \bigcup L_v)\} \quad \text{where } D = \{f \mapsto \langle v_1, \dots, v_n \rangle, \dots\} \quad \textcircled{5} \\
l &= \text{type } C(x : \text{Query}) = \\
&\quad \text{member } \langle v \rangle : C_v = \\
&\quad \quad C_v(\sigma_{\varphi_v}(x)) \quad \forall v \in \{v_1, \dots, v_n\} \\
&\quad \quad \text{where } C_v, L_v = \text{filter}(F, \sigma_{\varphi_v}(D)) \\
&\quad \quad \text{and } \varphi_v(r) = r(f) = v \quad \textcircled{6}
\end{aligned}$$

■ **Figure 11** Pivot type provider – grouping and aggregation

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:

$$\begin{aligned}
\text{pivot}(F, D) &= C, L \quad \text{where} \\
F &= \{f_1 \mapsto \tau_1, \dots, f_n \mapsto \tau_n\} \\
D &= \{f_1 \mapsto \langle v_{1,1}, \dots, v_{1,r} \rangle, \dots, f_n \mapsto \langle v_{n,1}, \dots, v_{n,r} \rangle\}
\end{aligned}$$

In prior work [26], 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 $\textcircled{1}$ together with one of the operations discussed before and the newly added `filter` function. As members performing data transformations are generated, the provider applies the same transformation on the sample data. For example, the revised `drop` function $\textcircled{2}$ takes the sample data set D ; when calling `drop` recursively to generate nested class after dropping a column $\textcircled{3}$, it removes 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 $\textcircled{4}$ 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 $\textcircled{5}$. The predicate that filters data based on the value $\textcircled{6}$ 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`.

```
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

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 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.

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

■ **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 calculate 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

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.

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 calculated over a group and select columns used for sorting. It is generated automatically by looking for a 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».

55:22 Data exploration through dot-driven development

```
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

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 [28] 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. This is not usually the case when writing code in text editor. However, 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 editor⁶. 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 and so the secondary ordering is arbitrary. As the user moves through the code, or writes the code, the live preview is updated accordingly.

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 to specify additional parameters within the “then” pattern. In case of sorting, this allows adding further sorting keys.

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.

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

8 Related and further work

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

8.1 Further work

The pivot type provider shows the feasibility of using dot-driven development as a mechanism behind simple programming tools for data exploration. Extending the mechanism to handle large and dirty datasets poses a number of interesting challenges.

Scalability. A benefit of our approach based on relational algebra, is that the query constructed by the pivot type provider can be translated to SQL and executed by a database engine. This means that evaluating the query over large data sets does not pose a problem. However, the completion lists generated from data when filtering may require further consideration.

We plan to explore a number of possibilities such as grouping the values by a prefix (e.g. «starting with LO».London and «starting with CA».Cambridge) or grouping the values by their frequency (for example, «occurring less than 100 times».Grantchester and «occurring more than 10000 times».London). Such encoding makes it possible to scale to an arbitrary data size, provided that the backing data storage is equipped with an appropriate index.

Expressivity. The case studies presented in the paper show that the pivot type provider is practically useful in its current form, but we acknowledge that its expressivity is limited to simple queries. Making the tool more expressive to allow tasks such as denormalisation, handling of missing values and dirty data is an important problem. Unlike data querying (which is captured by the relational algebra), there is no generally accepted “algebra of data cleaning” and so more foundational work is needed, possibly building on from tools such as Wrangler [16] and PADS [12]. We believe that the “dot-driven development” methodology can support richer languages and we intend to explore this direction in the future.

8.2 Related work

Our work builds on type providers, which have been pioneered in F# [34]. The technical contributions are related to several works on type systems. This section also gives an overview of related work on human-computer interaction and commercial tools for data visualization.

Type providers. Type providers first appeared in F# [34] and can also be seen as a form of dependent typing [7]; 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 [26, 19, 27], which focuses on mapping of external data into types. To our best knowledge, the Azure type provider [3] is the first type provider that provides members for specifying a restricted form of queries.

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 [9, 37]. The approach sketched in Section 4.1 used row types and tpestate or phantom types [35, 32, 18]. The idea of using type providers to encode fancy types has also been explored for session types [13, 2] and it would be interesting to see whether our approach can be applied in other areas such as web development [6].

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 [28, 29, 30]. Unlike spreadsheets, we construct a transformation rather than actually transforming data, which makes it more related to systems for query construction [20, 5]. 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 [8, 24], although our focus was on aggregation, which has been added to the core algebra in a number of different ways [23, 14, 4, 10]. 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.

Commercial tools. There is a wide range of commercial tools for building dashboards and data visualizations such as Microsoft Power BI [36], Tableau [38] and Qlik [15]. Those allow users to build data visualizations through a user interface and embedded scripting capabilities. The main difference from the pivot type provider is that none of these tools treats source code as primary and so they do not provide the same level of reproducibility as scripts written using the pivot type provider.

9 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. On 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 tpestate 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!

Acknowledgements. The author is grateful to Don Syme for numerous discussions about type providers, James Geddes and Kenji Takeda for suggestions and useful references and to Mariana Marasoiu and Alan Blackwell for ideas on human-computer interaction aspects of the work. Finally, thanks to the anonymous reviewers for useful suggestions and corrections.

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 Isaac Abraham. Azure storage type provider. Available online., 2016. URL: <http://fsprojects.github.io/AzureStorageTypeProvider/>.
- 4 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.
- 5 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.
- 6 Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. *SIGPLAN Not.*, 45(6):122–133, June 2010. doi:10.1145/1809028.1806612.
- 7 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.
- 8 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.
- 9 Anthony Cowley. Frames: Data frames for tabular data. Available on GitHub, 2017. URL: <https://github.com/acowley/Frames>.
- 10 Richard Cyganiak. A relational algebra for sparql. *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170*, page 35, 2005.
- 11 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>.
- 12 Kathleen Fisher and Robert Gruber. Pads: a domain-specific language for processing ad hoc data. In *ACM Sigplan Notices*, volume 40, pages 295–304. ACM, 2005.
- 13 Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *European Symposium on Programming*, pages 74–90. Springer, 1999.
- 14 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.
- 15 Christopher Ilacqua, Henric Cronstrom, and James Richardson. *Learning Qlik Sense®: The Official Guide*. Packt Publishing Ltd, 2015.
- 16 Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011. URL: <http://vis.stanford.edu/papers/wrangler>.
- 17 Paul Krugman. The Excel depression. *New York Times*, 18, 2013.
- 18 Daan Leijen and Erik Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, December 1999. doi:10.1145/331963.331977.
- 19 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.
- 20 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.
- 21 Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

- 22 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.
- 23 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.
- 24 M. Tamer Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, 3rd edition, 2007.
- 25 Raymond R Panko. What we know about spreadsheet errors. *Journal of Organizational and End User Computing (JOEUC)*, 10(2):15–21, 1998.
- 26 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.
- 27 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.
- 28 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.
- 29 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.
- 30 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.
- 31 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
- 32 Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Eng.*, (1):157–171, 1986.
- 33 Don Syme. F# 4.0 specllet - 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>.
- 34 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.
- 35 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.
- 36 Christopher Webb et al. *Power Query for Power BI and Excel*. Apress, 2014.
- 37 Stephanie Weirich. Depending on types. *SIGPLAN Not.*, 49(9):241–241, August 2014. doi:10.1145/2692915.2631168.
- 38 Richard Wesley, Matthew Eldridge, and Pawel T Terlecki. An analytic data engine for visualization in tableau. In *Proceedings of International Conference on Management of Data*, pages 1185–1194. ACM, 2011.

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.

Part III

Publications: Data infrastructure

Chapter 8

Foundations of a live data exploration environment

Tomas Petricek. 2020. Foundations of a live data exploration environment. *Art Sci. Eng. Program.* 4, 3 (2020), 8. <https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2020/4/8>

Foundations of a live data exploration environment

Tomas Petricek^{a,b}

a University of Kent, UK

b and The Alan Turing Institute, UK

Abstract A growing amount of code is written to explore and analyze data, often by data analysts who do not have a traditional background in programming, for example by journalists. The way such data analysts write code is different from the way software engineers do so. They use few abstractions, work interactively and rely heavily on external libraries. We aim to capture this way of working and build a programming environment that makes data exploration easier by providing instant live feedback.

We combine theoretical and applied approach. We present the *data exploration calculus*, a formal language that captures the structure of code written by data analysts. We then implement a data exploration environment that evaluates code instantly during editing and shows previews of the results. We formally describe an algorithm for providing instant previews for the data exploration calculus that allows the user to modify code in an unrestricted way in a text editor. Supporting interactive editing is tricky as any edit can change the structure of code and fully recomputing the output would be too expensive. We prove that our algorithm is correct and that it reuses previous results when updating previews after a number of common code edit operations. We also illustrate the practicality of our approach with an empirical evaluation and a case study.

As data analysis becomes an ever more important use of programming, research on programming languages and tools needs to consider new kinds of programming workflows appropriate for those domains and conceive new kinds of tools that can support them. The present paper is one step in this important direction.

ACM CCS 2012

- **Human-centered computing** → *Interactive systems and tools*;
- **Information systems** → *Data mining*;
- **Software and its engineering** → *Compilers*;

Keywords Data exploration, live programming, data journalism, instant previews

The Art, Science, and Engineering of Programming

Submitted September 30, 2019

Published February 17, 2020

doi 10.22152/programming-journal.org/2020/4/8



© Tomas Petricek

This work is licensed under a “CC BY 4.0” license.

In *The Art, Science, and Engineering of Programming*, vol. 4, no. 3, 2020, article 8; 31 pages.

Foundations of a live data exploration environment

1 Introduction

One of the aspects that make spreadsheets easier to use than other programming tools is their liveness. When you change a cell in Excel, the whole spreadsheet updates instantly and you immediately see new results, without having to explicitly trigger re-computation and without having to wait for an extensive period of time.

An increasing number of programming environments aim to provide a live development experience for standard programming languages, but doing this is not easy. Fully recomputing the whole program after every keystroke is inefficient and calculating how a change in the source code changes the result is extremely hard when the text editor allows arbitrary changes. Consider the following snippet that gets the release years of the 10 most expensive movies from a data set `movies`:

```
let top = movies.sortBy( $\lambda x \rightarrow x.getBudget()$ )
                .take(10).map( $\lambda x \rightarrow x.getReleased().format("yyyy")$ )
```

A live programming environment computes and displays the list of years. Suppose that the programmer then makes `10` a variable and changes the date format:

```
let count = 10
let top = movies.sortBy( $\lambda x \rightarrow x.getBudget()$ )
                .take(count).map( $\lambda x \rightarrow x.getReleased().format("dd-mm-yyyy")$ )
```

Ideally, the live programming environment should understand the change, reuse a cached result of the first two transformations (sorting and taking the first 10 elements) and only evaluate the map operation to differently format the release dates of top 10 movies. Our environment does this for a simple data exploration language in an unrestricted text editor. We discuss related work in section 7, but we briefly review the most important directions here, in order to situate our contributions.

Contributions

We present the design and implementation of a live programming environment for a simple data exploration language that provides correct and efficient instant feedback, yet is integrated into an ordinary text editor. Our key contributions are:

- We introduce the *data exploration calculus* (section 3), a small formally tractable language for data exploration. The calculus is motivated by our review of how data analysts work (section 2) and makes it possible to write simple, transparent and reproducible data analyses.
- A live programming environment does not operate in batch mode and so we cannot follow classic compiler literature. We capture the essence of such new perspective (section 4) and use it to build our *instant preview* mechanism (section 5) that evaluates code instantly during editing.
- We prove that our instant preview mechanism is correct (section 6.1) and that it reuses previously evaluated values when possible (section 6.2). We illustrate the practicality of the mechanism using an empirical evaluation (section 6.3) and a case study (section 6.4).

UN Comtrade exports data

```
material = 'plastics' # 'plastics', 'paper'
```

Loading exports data

```
df_mat = pd.read_csv('{material}-2017.csv').fillna(0).sort_values(['country_name', 'period'])
df_mat.head()
```

	period	country_name	kg	country_code
0	2017-01-01	Algeria	43346.0	12
1	2017-03-01	Algeria	32800.0	12
2	2017-03-01	Antigua and Barbuda	17000.0	28

Join to country codes

```
# Set keep_default_na because the Namibia has ISO code NA
df_isos = pd.read_excel('iso.xlsx', keep_default_na=False).drop_duplicates('country_code')
df = df_mat.copy().merge(df_isos, 'left', 'country_code').rename({'iso2': 'country_code'}, axis=1)
df.head()
```

	period	country_name	kg	country_code
0	2017-01-01	Algeria	43346.0	DZ
1	2017-03-01	Algeria	32800.0	DZ
2	2017-03-01	Antigua and Barbuda	17000.0	AG

■ **Figure 1** Financial Times analysis that joins UN trade database with ISO country codes.

2 Understanding how data scientists work

Data scientists often use general-purpose programming languages such as Python, but the kind of code they write and the way they interact with the system is very different from how software engineers work [21]. This paper focuses on simple data wrangling and data exploration as done, for example, by journalists analysing government datasets. This new kind of non-expert programmers is worth our attention as they often work on informing the public. They need easy-to-use tools, but not necessarily a full programming language. In this section, we illustrate how such data analyses look and we provide a justification for the design of our data exploration calculus.

2.1 Simple data exploration in Jupyter

Data analysts increasingly use notebook systems such as Jupyter, which make it possible to combine text, equations and code with results of running the code, such as tables or visualizations. Notebooks blur the conventional distinction between

Foundations of a live data exploration environment

development and execution. Data analysts write small snippets of code, run them to see results immediately and then revise them.

Notebooks are used by users ranging from scientists who implement complex models of physical systems to journalists who perform simple data aggregations and create visualizations. Our focus is on the simplest use cases. Making programmatic data exploration more spreadsheet-like should encourage users to choose programming tools over spreadsheets, resulting in more reproducible and transparent data analyses.

Consider the Financial Times analysis of plastic waste [7, 25]. It joins datasets from Eurostat, UN Comtrade and more, aggregates the data and builds a visualization comparing waste flows in 2017 and 2018. Figure 1 shows an excerpt from one notebook of the data analysis. The code has a number of important properties:

- There is no abstraction. The analysis uses lambda functions as arguments to library calls, but it does not define any custom functions. Code is parameterized by having a global variable material set to "plastics" and keeping other possible values in a comment. This lets the analyst run and check results of intermediate steps.
- The code relies on external libraries. Our example uses Pandas [36], which provides operations for data wrangling such as merge to join datasets or drop_duplicates to delete rows with duplicate column values. Such standard libraries are external to the data analysis and are often implemented in another language like C++.
- The code is structured as a sequence of commands. Some commands define a variable, either by loading data, or by transforming data loaded previously. Even in Python, data is often treated as immutable. Other commands produce an output that is displayed in the notebook.
- There are many corner cases, such as the fact that keep_default_na needs to be set to handle Namibia correctly. These are discovered interactively by running the code and examining the output, so providing an instant feedback is essential.

Many Jupyter notebooks are more complex than the above example and might use helper functions or object-oriented code. However, simple data analyses such as the one discussed here are frequent enough and pose interesting problems for programming tools. This paper aims to bring such analyses to the attention of programming research community by capturing their essential properties as a formal calculus.

2.2 Dot-driven data exploration in The Gamma

Simple data exploration has been the motivation for a scripting language The Gamma [45]. Scripts in The Gamma are sequences of commands that either define a variable or produce an output. It does not support top-level functions and lambda functions can be used only as method arguments. Given the limited expressiveness of The Gamma, libraries are implemented in other languages, such as JavaScript. The Gamma uses type providers [54] for accessing external data sources. Type providers generate object types with members and The Gamma offers those in an auto-complete list when the user types dot (‘.’) to access a member. The combination of type providers and auto-complete makes it possible to solve a large number of data exploration tasks through the very simple interaction of selecting operations from a list.

```
olympics
.'filter data'. 'Games is'. 'Rio (2016)'. then
.'group data'. 'by Athlete'
.'count distinct Event'
.'su
```

- count distinct Gender
- count distinct Gold
- count distinct Medal
- count distinct Silver
- count distinct Sport
- count distinct Team
- count distinct Year
- sum Bronze
- sum Gold
- sum Silver
- sum Year
- then

(a) The analysis counts the number of distinct events per athlete. After typing '.' the editor offers further aggregation operations.

```
olympics
.'filter data'. 'Games is'. 'Rio (2016)'. then
.'group data'. 'by Athlete'
.'count distinct Event'. 'sum Gold'. then
.'sort data'. 'by Gold descending'. then
.'paging.take(5)
```

Athlete	Event	Gold
Michael Phelps	6	5
Katie Ledecky	5	4
Simone Biles	5	4
Katinka Hosszu	4	3
Usain Bolt	3	3

(b) Our live programming environment for The Gamma. The table is updated on-the-fly and shows the result at the current cursor position.

■ **Figure 2** Previous work on The Gamma with auto-complete based on type information (left) and our new editor with instant preview (right).

The example in figure 2a summarizes data on Olympic medals. Identifiers such as 'sum Bronze' are names of members generated by the type provider. The type provider used in this example generates an object with members for data transformations such as 'group data', which return further objects with members for specifying transformation parameters, such as selecting the grouping key using 'by Athlete'.

The Gamma language is richer, but the example in figure 2a shows that non-trivial data exploration can be done using a very simple language. The assumptions about structure of code that are explicit in The Gamma are implicitly present in Python and R data analyses produced by journalists, economists and other users with other than programming background. When we refer to The Gamma in this paper, readers not familiar with it can consider a small subset of Python.

2.3 Live programming for data exploration

The implementation that accompanies this paper builds a live programming environment for The Gamma. It is discussed in section 6.4 and it replaces the original text editor with just auto-complete with a live programming environment that provides *instant previews*.

An example of a instant preview is shown in figure 2b. As noted earlier, The Gamma programs consist of lists of commands which are either expressions or let bindings. Our editor displays a instant preview below the command that the user is currently editing. The preview shows the result of evaluating the expression or the value assigned to a bound variable. When the user changes the code, the preview is updated automatically, without any additional interaction with the user.

Foundations of a live data exploration environment

Programs, commands, terms, expressions and values

$$\begin{array}{lll}
 p ::= c_1; \dots; c_n & t ::= o \mid x & e ::= t \mid \lambda x \rightarrow e \\
 c ::= \text{let } x = t \mid t & \mid t.m(e, \dots, e) & v ::= o \mid \lambda x \rightarrow e
 \end{array}$$

Evaluation contexts of expressions

$$\begin{array}{l}
 C_e[-] = C_e[-].m(e_1, \dots, e_n) \mid o.m(v_1, \dots, v_m, C_e[-], e_1, \dots, e_n) \mid - \\
 C_c[-] = \text{let } x = C_e[-] \mid C_e[-] \\
 C_p[-] = o_1; \dots; o_k; C_c[-]; c_1; \dots; c_n
 \end{array}$$

Let elimination and member reduction

$$\begin{array}{l}
 o_1; \dots; o_k; \text{let } x = o; c_1; \dots; c_n \rightsquigarrow \\
 \quad o_1; \dots; o_k; o; c_1[x \leftarrow o]; \dots; c_n[x \leftarrow o] \quad (\text{let}) \\
 \frac{o.m(v_1, \dots, v_n) \rightsquigarrow_\epsilon o'}{C_p[o.m(v_1, \dots, v_n)] \rightsquigarrow C_p[o']} \quad (\text{external})
 \end{array}$$

■ **Figure 3** Syntax, contexts and reduction rules of the data exploration calculus

There are a number of guiding principles that inform our design. First, we allow the analyst to edit code in an unrestricted form in a text editor. Although structured editors provide an appealing alternative and make recomputation easier, we prefer the flexibility of plain text. Second, we focus on the scenario when code changes, but input does not. Rapid feedback allows the analyst to quickly adapt code to correctly handle corner cases that typical analysis involves. In contrast to work on incremental computation, we do not consider the case when data changes, although supporting interactive data exploration of streaming data is an interesting future direction.

3 Data exploration calculus

The *data exploration calculus* is a small formal language for data exploration. The calculus is not, in itself, Turing-complete and it can only be used together with external libraries that define what objects are available and what the behaviour of their members is. This is sufficient to capture the simple data analyses discussed in section 2. We define the calculus in this section and then use it to formalise our instant preview mechanism in section 4. The instant preview mechanism does not rely on types and so we postpone the discussion of static typing to appendix C. Interestingly, it reuses the mechanism used for live previews.

3.1 Language syntax

The calculus combines object-oriented features such as member access with functional features including lambda functions. The syntax is defined in figure 3. Object values o are defined by external libraries that are used in conjunction with the core calculus.

A program p in the data exploration calculus consists of a sequence of commands c . A command can be either a let binding or a term. Let bindings define variables x that can be used in subsequent commands. Lambda functions can only appear as arguments in method calls. A term t can be a value, variable or a member access, while an expression e , which can appear as an argument in member access, can be a lambda function or a term.

3.2 Operational semantics

The data exploration calculus is a call-by-value language. We model evaluation as a small-step reduction \rightsquigarrow . Fully evaluating a program results in an irreducible sequence of objects $o_1; \dots; o_n$ (one object for each command, including let bindings) which can be displayed as intermediate results of the data analysis. The operational semantics is parameterized by a relation \rightsquigarrow_e that models the functionality of the external libraries used with the calculus and defines the reduction behaviour for member accesses. The relation has the following form:

$$o_1.m(v_1, \dots, v_n) \rightsquigarrow_e o_2$$

Here, the operation m is invoked on an object and takes values (objects or function values) as arguments. The reduction always results in an object. Figure 3 defines the reduction rules in terms of \rightsquigarrow_e and evaluation contexts; C_e specifies left-to-right evaluation of arguments of a method call, C_c specifies evaluation of a command and C_p defines left-to-right evaluation of a program. The rule (external) calls a method provided by an external library in a call-by-value fashion; (let) substitutes a value of an evaluated variable in all subsequent commands and leaves the result in the list of commands. Note that our semantics does not define how λ applications are reduced. This is done by external libraries, which will typically supply functions with arguments using standard β -reduction. The behaviour is subject to constraints discussed next.

3.3 Example external library

The data exploration calculus is not limited to the data exploration domain. It can be used with external libraries for a wide range of other simple programming tasks, such as image manipulation, as done in section 6.3. However, we choose a name that reflects the domain that motivated this paper. To illustrate how a definition of an external library looks, consider the following simple data manipulation script:

```
let l = list.range(0, 10)
l.map( $\lambda x \rightarrow$  math.mul( $x$ , 10))
```

An external library provides the list and math values with members range, map and mul. The objects of the external library consist of numbers n , lists of objects $[o_1, \dots, o_k]$

Foundations of a live data exploration environment

and failed computations \perp [37]. Next, the external library needs to define the $\rightsquigarrow_\epsilon$ relation that defines the evaluation of member accesses. The following shows the rules for members of lists, assuming the only supported member is map:

$$\frac{e[x \leftarrow n_i] \rightsquigarrow o_i \quad (\text{for all } i \in 1 \dots k)}{[n_1, \dots, n_k].\text{map}(\lambda x \rightarrow e) \rightsquigarrow_\epsilon [o_1, \dots, o_k]} \quad \frac{(\text{otherwise})}{[n_1, \dots, n_k].m(v_1, \dots, v_n) \rightsquigarrow_\epsilon \perp}$$

When evaluating map, we apply the provided function to all elements of the list using standard β -reduction, defined recursively using \rightsquigarrow , and return a list with resulting objects. The $\rightsquigarrow_\epsilon$ relation is defined on all member accesses, but non-existent members reduce to the failed computation \perp .

3.4 Properties

The data exploration calculus has a number of desirable properties. Some of those require that the relation $\rightsquigarrow_\epsilon$, which defines evaluation for external libraries, satisfies a number of conditions. We discuss *normalization* and *let elimination* in this section. Those two are particularly important as they will allow us to prove correctness of our method of evaluating instant previews in section 6.1.

Definition 1 (Further reductions). We define two additional reduction relations:

- We write \rightsquigarrow^* for the reflexive, transitive closure of \rightsquigarrow
- We write $\rightsquigarrow_{\text{let}}$ for a call-by-name let binding elimination $c_1; \dots; c_{k-1};$
 $\text{let } x = t; c_{k+1}; \dots; c_n \rightsquigarrow_{\text{let}} c_1; \dots; c_{k-1}; t; c_{k+1}[x \leftarrow t]; \dots; c_n[x \leftarrow t]$

We say that two expressions e and e' are *observationally equivalent* if, for any context C , the expressions $C[e]$ and $C[e']$ reduce to the same value. Lambda functions $\lambda x \rightarrow 2$ and $\lambda x \rightarrow 1+1$ are not equal, but they are observationally equivalent. We require that external libraries satisfy two conditions. First, when a method is called with observationally equivalent values as arguments, it should return the same value. Second, the evaluation of $o.m(v_1, \dots, v_n)$ should be defined for all o, n and v_i . The definition in section 3.3 satisfies those by using standard β -reduction for lambda functions and by reducing all invalid calls to the \perp object.

Definition 2 (External library). An external library consists of a set of objects O and a reduction relation $\rightsquigarrow_\epsilon$ that satisfies the following two properties:

Totality For all o, m, i and all v_1, \dots, v_i , there exists o' such that $o.m(v_1, \dots, v_i) \rightsquigarrow_\epsilon o'$.

Compositionality For observationally equivalent arguments, the reduction should always return the same object, i.e. given e_0, e_1, \dots, e_n and e'_0, e'_1, \dots, e'_n and m such that $e_0.m(e_1, \dots, e_n) \rightsquigarrow^* o$ and $e'_0.m(e'_1, \dots, e'_n) \rightsquigarrow^* o'$ then if for any contexts C_0, C_1, \dots, C_n it holds that if $C_i[e_i] \rightsquigarrow^* o_i$ and $C_i[e'_i] \rightsquigarrow^* o_i$ for some o_i then $o = o'$.

Compositionality is essential for proving the correctness of our instant preview mechanism and implies determinism of external libraries. Totality allows us to prove normalization, i.e. all programs reduce to a value – although the resulting value may be an error value provided by the external library.

Theorem 1 (Normalization). *For all p , there exists n, o_1, \dots, o_n such that $p \rightsquigarrow^* o_1; \dots; o_n$.*

Proof. A program that is not a sequence of values can be reduced and reduction decreases the size of the program. See appendix A.1 for more detail. \square

Although the reduction rules (let) and (external) of the data exploration calculus define an evaluation in a call-by-value order, eliminating let bindings in a call-by-name way using the $\rightsquigarrow_{\text{let}}$ reduction does not affect the result. This simplifies our later proof of instant preview correctness in section 6.1.

Lemma 2 (Let elimination for a program). *Given any program p such that $p \rightsquigarrow^* o_1; \dots; o_n$ for some n and o_1, \dots, o_n then if $p \rightsquigarrow_{\text{let}} p'$ for some p' then also $p' \rightsquigarrow^* o_1; \dots; o_n$.*

Proof. By constructing $p' \rightsquigarrow^* o_1; \dots; o_n$ from $p \rightsquigarrow^* o_1; \dots; o_n$. See appendix A.2. \square

4 Formalising a live programming environment

A naive way of providing instant previews during code editing would be to re-evaluate the code after each change. This would be wasteful – when writing code to explore data, most changes are additive. To update a preview, we only need to evaluate newly added code. We describe an efficient mechanism in this section.

4.1 Maintaining the dependency graph

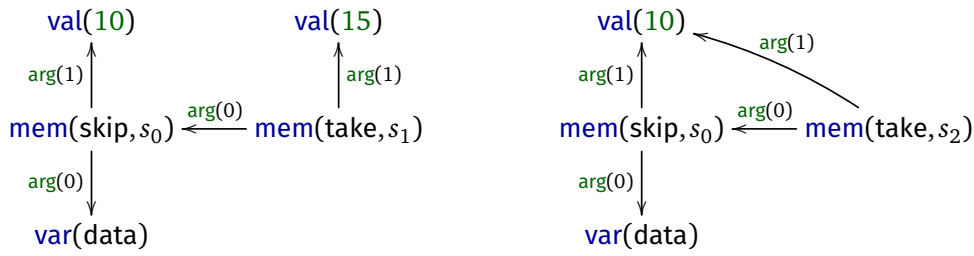
The key idea behind our method is to maintain a dependency graph [32] with nodes representing individual operations of the computation that can be evaluated to obtain a preview. Each time the program text is modified, we parse it afresh (using an error-recovering parser) and bind the abstract syntax tree to the dependency graph. When binding a new expression to the graph, we reuse previously created nodes as long as they have the same structure and the same dependencies. For expressions that have a new structure, we create new nodes.

The nodes of the graph serve as unique keys into a lookup table containing previously evaluated parts of the program. When a preview is requested for an expression, we use the graph node bound to the expression to find a preview. If a preview has not been evaluated, we force the evaluation of all dependencies in the graph and then evaluate the operation represented by the current node.

4.1.1 Elements of the graph

The nodes of the graph represent individual operations of the computation. In our design, the nodes are used as cache keys, so we attach a unique symbol s to some of the nodes. That way, we can create two unique nodes representing, for example, access to a member named `take` which differ in their dependencies.

Foundations of a live data exploration environment



(a) Graph constructed from initial expression:
`let x = 15 in data.skip(10).take(x)`

(b) Updated graph after changing x to 10:
`let x = 10 in data.skip(10).take(x)`

■ **Figure 4** Dependency graphs formed by two steps of the live programming process.

The graph edges are labelled with labels indicating the kind of dependency. For a method call, the labels are “first argument”, “second argument” and so on. Writing g for symbols and i for integers, nodes (vertices) v and edge labels l are defined as:

$$\begin{aligned}
 v &= \text{val}(o) \mid \text{var}(x) \mid \text{mem}(m, s) \mid \text{fun}(x, s) && \text{(Vertices)} \\
 l &= \text{body} \mid \text{arg}(i) && \text{(Edge labels)}
 \end{aligned}$$

The `val` node represents a primitive value and contains the object itself. Two occurrences of `10` in the source code will be represented by the same node. Member access `mem` contains the member name, together with a unique symbol s – two member access nodes with different dependencies will contain a different symbol. Dependencies of member access are labelled with `arg` indicating the index of the argument (0 for the instance and $1, \dots$ for the arguments). Finally, nodes `fun` and `var` represent function values and variables bound by λ abstraction.

4.1.2 Example graph

figure 4 illustrates how we build the dependency graph. Node representing `take(x)` depends on the argument – the number `15` – and the instance, which is a node representing `skip(10)`. This depends on the instance `data` and the number `10`. Note that variables bound via `let` binding such as x do not appear as `var` nodes. The node using it depends directly on the node representing the expression assigned to x .

After changing the value of x , we create a new graph. The dependencies of the node `mem(skip, s_0)` are unchanged and so the symbol s_0 attached to the node remains the same and previously computed previews can be reused. This part of the program is not recomputed. The `arg(1)` dependency of the `take` call changed and so we create a new node `mem(skip, s_2)` with a fresh symbol s_2 . The preview for this node is then computed as needed using the already known values of its dependencies.

4.1.3 Reusing graph nodes

The binding process takes an expression and constructs a dependency graph. It uses a lookup table to reuse previously created member access and function value nodes. The key in the lookup table is formed by a node kind together with a list of dependencies.

$$\begin{aligned}
\text{bind-expr}_{\Gamma,\Delta}(e_0.m(e_1,\dots,e_n)) &= \nu, (\{\nu\} \cup V_0 \cup \dots \cup V_n, E \cup E_0 \cup \dots \cup E_n) & (1) \\
&\text{ when } \nu_i, (V_i, E_i) = \text{bind-expr}_{\Gamma,\Delta}(e_i) \text{ and } \nu = \Delta(\text{mem}(m), [(\nu_0, \text{arg}(0)), \dots, (\nu_n, \text{arg}(n))]) \\
&\text{ let } E = \{(\nu, \nu_0, \text{arg}(0)), \dots, (\nu, \nu_n, \text{arg}(n))\} \\
\text{bind-expr}_{\Gamma,\Delta}(e_0.m(e_1,\dots,e_n)) &= \nu, (\{\nu\} \cup V_0 \cup \dots \cup V_n, E \cup E_0 \cup \dots \cup E_n) & (2) \\
&\text{ when } \nu_i, (V_i, E_i) = \text{bind-expr}_{\Gamma,\Delta}(e_i) \text{ and } \Delta(\text{mem}(m), [(\nu_0, \text{arg}(0)), \dots, (\nu_n, \text{arg}(n))]) \downarrow \\
&\text{ let } \nu = \text{mem}(m, s), s \text{ fresh and } E = \{(\nu, \nu_0, \text{arg}(0)), \dots, (\nu, \nu_n, \text{arg}(n))\} \\
\text{bind-expr}_{\Gamma,\Delta}(\lambda x \rightarrow e) &= \nu, (\{\nu\} \cup V, \{e\} \cup E) & (3) \\
&\text{ when } \Gamma_1 = \Gamma \cup \{x, \text{var}(x)\} \text{ and } \nu_0, (V, E) = \text{bind-expr}_{\Gamma_1,\Delta}(e) \text{ and } \nu = \Delta(\text{fun}(x), [(\nu_0, \text{body})]) \\
&\text{ let } e = (\nu, \nu_0, \text{body}) \\
\text{bind-expr}_{\Gamma,\Delta}(\lambda x \rightarrow e) &= \nu, (\{\nu\} \cup V, \{e\} \cup E) & (4) \\
&\text{ when } \Gamma_1 = \Gamma \cup \{x, \text{var}(x)\} \text{ and } \nu_0, (V, E) = \text{bind-expr}_{\Gamma_1,\Delta}(e) \text{ and } \Delta(\text{fun}(x), [(\nu_0, \text{body})]) \downarrow \\
&\text{ let } \nu = \text{fun}(x, s), s \text{ fresh and } e = (\nu, \nu_0, \text{body}) \\
\text{bind-expr}_{\Gamma,\Delta}(o) &= \text{val}(o), (\{\text{val}(o)\}, \emptyset) & (5) \\
\text{bind-expr}_{\Gamma,\Delta}(x) &= \nu, (\{\nu\}, \emptyset) \text{ when } \nu = \Gamma(x) & (6)
\end{aligned}$$

■ **Figure 5** Binding rules that define a construction of a dependency graph for an expression.

A node kind includes the member or variable name; a lookup table Δ then maps a node kind with a list of dependencies to a cached node:

$$\begin{aligned}
k &::= \text{fun}(x) \mid \text{mem}(m) \quad (\text{Node kinds}) \\
\Delta(k, [(\nu_1, l_1), \dots, (\nu_n, l_n)]) &\quad (\text{Lookup for a node})
\end{aligned}$$

The example on the second line looks for a node of a kind k that has dependencies ν_1, \dots, ν_n labelled with labels l_1, \dots, l_n . We write $\Delta(k, l) \downarrow$ when a value for a key is not defined. When creating the graph in figure 4b, we perform the following lookups:

$$\begin{aligned}
\Delta(\text{mem}(\text{skip}), [(\text{var}(\text{data}), \text{arg}(0)), (\text{val}(10), \text{arg}(1))]) & \quad (1) \\
\Delta(\text{mem}(\text{take}), [(\text{mem}(\text{skip}, s_0), \text{arg}(0)), (\text{val}(10), \text{arg}(1))]) & \quad (2)
\end{aligned}$$

First, we look for the skip member access. The result is the $\text{mem}(\text{skip}, s_0)$ known from the previous step. We then look for the take member access. In the earlier step, the argument of take was 15 rather than 10 and so this lookup fails. We then construct a new node $\text{mem}(\text{take}, s_2)$ and later add it to the cache.

4.2 Binding expressions to a graph

After parsing modified code, we update the dependency graph and link each node of the abstract syntax tree to a node of the dependency graph. This process is called binding and is defined by the `bind-expr` function (figure 5) and `bind-prog` function (figure 6). Both functions are annotated with a lookup table Δ and a variable context Γ .

Foundations of a live data exploration environment

$$\text{bind-prog}_{\Gamma,\Delta}(\text{let } x = e; c_2; \dots; c_n) = v_1; \dots; v_n, (\{v_1\} \cup V \cup V_1, E \cup E_1) \quad (7)$$

$$\begin{aligned} \text{let } v_1, (V_1, E_1) &= \text{bind-expr}_{\Gamma,\Delta}(e_1) \text{ and } \Gamma_1 = \Gamma \cup \{(x, v_1)\} \\ \text{and } v_2; \dots; v_n, (V, E) &= \text{bind-prog}_{\Gamma_1,\Delta}(c_2; \dots; c_n) \end{aligned}$$

$$\text{bind-prog}_{\Gamma,\Delta}(e; c_2; \dots; c_n) = v_1; \dots; v_n, (\{v_1\} \cup V \cup V_1, E \cup E_1) \quad (8)$$

$$\text{let } v_1, (V_1, E_1) = \text{bind-expr}_{\Gamma,\Delta}(e) \text{ and } v_2; \dots; v_n, (V, E) = \text{bind-prog}_{\Gamma_1,\Delta}(c_2; \dots; c_n)$$

$$\text{bind-prog}_{\Gamma,\Delta}([\] = [\], (\emptyset, \emptyset) \quad (9)$$

■ **Figure 6** Binding rules that define a construction of a dependency graph for a program.

$\text{update}_{V,E}(\Delta_{i-1}) = \Delta_i$ such that:

$$\begin{aligned} \Delta_i(\text{mem}(m), [(v_0, \text{arg}(0)), \dots, (v_n, \text{arg}(n))]) &= \text{mem}(m, s) \\ \text{when } \text{mem}(m, s) \in V \text{ and } (v_i, \text{arg}(i)) \in E \text{ for } i \in 0, \dots, n \\ \Delta_i(\text{fun}(x), [(v_1, \text{body})]) &= \text{fun}(x, s) \\ \text{when } \text{fun}(x, s) \in V \text{ and } (v_1, \text{body}) \in E \\ \Delta_i(v) &= \Delta_{i-1}(v) \quad (\text{otherwise}) \end{aligned}$$

■ **Figure 7** Updating the node cache after binding a new graph

The variable context is a map from variable names to dependency graph nodes and is used for variables bound using **let** binding.

When invoked, $\text{bind-expr}_{\Gamma,\Delta}(e)$ returns a node v that corresponds to the expression e , paired with a dependency graph (V, E) formed by nodes V and labelled edges E . That edges are written as (v_1, v_2, l) and include a label l . The $\text{bind-prog}_{\Gamma,\Delta}$ function works similarly, but turns a sequence of commands into a sequence of nodes.

When binding a member access, we use bind-expr recursively to get a node and a dependency graph for each sub-expression. The nodes representing sub-expressions are then used for lookup into Δ , together with their labels. If a node already exists in Δ it is reused (1). Alternatively, we create a new node containing a fresh symbol (2). The graph node bound to a function depends on a synthetic node $\text{var}(x)$ that represents a variable of unknown value. When binding a function, we create a variable node and add it to the variable context Γ_1 before binding the body. As with member access, the node representing a function may (3) or may not (4) already exist in the lookup table.

When binding a program, we bind the first command and recursively process remaining commands (9). For **let** binding (7), we bind the expression e assigned to the variable to obtain a graph node v_1 . We then store the node in the variable context Γ_1 and bind the remaining commands. The variable context is used when binding a variable in bind-expr (6) and so all variables declared using **let** will be bound to a graph node representing the value assigned to the variable. When the command is just an expression (8), we bind the expression using bind-expr .

4.3 Edit and rebind loop

During editing, the dependency graph is repeatedly updated according to the binding rules. We maintain a series of lookup table states $\Delta_0, \Delta_1, \Delta_2, \dots$. The initial lookup table is empty, i.e. $\Delta_0 = \emptyset$. At a step i , we parse a program p_i and obtain a new dependency graph using the previous Δ . The result is a sequence of nodes corresponding to commands of the program and a graph (V, E) :

$$v_1; \dots; v_n, (V, E) = \text{bind-prog}_{\emptyset, \Delta_{i-1}}(p_i)$$

The new state of the cache is computed using $\text{update}_{V,E}(\Delta_{i-1})$ defined in figure 7. The function adds newly created nodes from the graph (V, E) to the previous cache Δ_{i-1} and returns a new cache Δ_i .

5 Computing instant previews

The binding process constructs a dependency graph after code changes. The nodes in the dependency graph correspond to individual operations that will be performed when running the program. When evaluating a preview, we attach partial results to nodes of the graph. Since the binding process reuses nodes, previews for sub-expressions attached to graph nodes will also be reused.

In this section, we describe how previews are evaluated. The evaluation is done over the dependency graph, rather than over the structure of program expressions as in the operational semantics given in section 3.2. In section 6, we prove that resulting previews are the same as the result we would get by directly evaluating code and we also show that no recomputation occurs when code is edited in certain ways.

5.1 Previews and delayed previews

Programs in the data exploration calculus consist of sequence of commands. Those are evaluated to a value with a preview that can be displayed to the user. However, we also support previews for sub-expressions. This can be problematic if the current sub-expression is inside the body of a function. For example:

```
let top = movies.take(10).map( $\lambda x \rightarrow x.getReleased().format("dd-mm-yyyy")$ )
```

Here, we can directly evaluate sub-expressions `movies` and `movies.take(10)`, but not `x.getReleased()` because it contains a free variable `x`. Our preview evaluation algorithm addresses this by producing two kinds of previews. A *fully evaluated preview* is just a value, while a *delayed preview* is a partially evaluated expression with free variables:

$$\begin{aligned} p &= o \mid \lambda x \rightarrow e && \text{(Fully evaluated previews)} \\ d &= p \mid \llbracket e \rrbracket_{\Gamma} && \text{(Evaluated and delayed previews)} \end{aligned}$$

A fully evaluated preview p can be either a primitive object or a function value with no free variables. A possibly delayed preview d can be either an evaluated preview p

Foundations of a live data exploration environment

$$\begin{array}{c}
\text{(lift-expr)} \frac{v \Downarrow \llbracket e \rrbracket_{\Gamma}}{v \Downarrow_{\text{lift}} \llbracket e \rrbracket_{\Gamma}} \qquad \text{(fun-val)} \frac{(\text{fun}(x, s), v, \text{body}) \in E \quad v \Downarrow p}{\text{fun}(x, s) \Downarrow \lambda x \rightarrow p} \\
\text{(lift-prev)} \frac{v \Downarrow p}{v \Downarrow_{\text{lift}} \llbracket p \rrbracket_{\emptyset}} \qquad \text{(fun-bind)} \frac{(\text{fun}(x, s), v, \text{body}) \in E \quad v \Downarrow \llbracket e \rrbracket_x}{\text{fun}(x, s) \Downarrow \lambda x \rightarrow e} \\
\text{(val)} \frac{}{\text{val}(o) \Downarrow o} \qquad \text{(fun-expr)} \frac{(\text{fun}(x, s), v, \text{body}) \in E \quad v \Downarrow \llbracket e \rrbracket_{x, \Gamma}}{\text{fun}(x, s) \Downarrow \llbracket \lambda x \rightarrow e \rrbracket_{\Gamma}} \\
\text{(var)} \frac{}{\text{var}(x) \Downarrow \llbracket x \rrbracket_x} \\
\text{(mem-val)} \frac{\forall i \in \{0 \dots k\}. (\text{mem}(m, s), v_i, \text{arg}(i)) \in E \quad v_i \Downarrow p_i \quad p_0.m(p_1, \dots, p_k) \rightsquigarrow_{\epsilon} p}{\text{mem}(m, s) \Downarrow p} \\
\text{(mem-expr)} \frac{\forall i \in \{0 \dots k\}. (\text{mem}(m, s), v_i, \text{arg}(i)) \in E \quad \exists j \in \{0 \dots k\}. v_j \not\Downarrow p_j \quad v_i \Downarrow_{\text{lift}} \llbracket e_i \rrbracket_{\Gamma_i}}{\text{mem}(m, s) \Downarrow \llbracket e_0.m(e_1, \dots, e_k) \rrbracket_{\Gamma_0, \dots, \Gamma_k}}
\end{array}$$

■ **Figure 8** Rules that define evaluation of previews over a dependency graph for a program

or an expression e that requires variables Γ . We use an untyped language and so Γ is just a list of variables x_1, \dots, x_n . As discussed in appendix B, delayed previews have an interesting theoretical link with graded comonads. The body of a lambda function may have a fully evaluated preview if it uses only variables that are bound by earlier let bindings, but it will typically be delayed. We consider a speculative design for an abstraction mechanism that better supports instant previews in appendix D.

5.2 Evaluation of previews

The evaluation of previews is defined in figure 8. Given a dependency graph (V, E) , the relation $v \Downarrow d$ evaluates a sub-expression corresponding to the node v to a possibly delayed preview d . The nodes V and edges E of the graph are parameters of \Downarrow , but they do not change during the evaluation and so we do not explicitly write them.

The auxiliary relation $v \Downarrow_{\text{lift}} d$ always evaluates to a delayed preview. If the ordinary evaluation returns a delayed preview, so does the auxiliary relation (lift-expr). If the ordinary evaluation returns a value, the value is wrapped into a delayed preview requiring no variables (lift-prev). A node representing a value is evaluated to a value (val) and a node representing an unbound variable is reduced to a delayed preview that requires the variable and returns its value (var).

For member access, we distinguish two cases. If all arguments evaluate to values (member-val), then we use the evaluation relation defined by external libraries $\rightsquigarrow_{\epsilon}$ to immediately evaluate the member access and produce a value. If some of the arguments are delayed (member-expr), because the member access is inside the body

of a lambda function, we produce a delayed member access expression that requires the union of the variables required by the individual arguments.

The evaluation of function values is similar, but requires three cases. If the body can be reduced to a value with no unbound variables (fun-val), we return a lambda function that returns the value. If the body requires only the bound variable (fun-bind), we return a lambda function with the delayed preview as the body. If the body requires further variables, the result is a delayed preview.

5.3 Caching of evaluated previews

For simplicity, the relation \Downarrow in figure 8 does not specify how previews are cached. In practice, this is done by maintaining a lookup table from graph nodes v to previews p . Whenever \Downarrow is used to obtain a preview for a graph node, we first check the lookup table. If the preview has not been previously evaluated, we evaluate it and add it to the lookup. Cached previews can be reused in two ways. First, if the same sub-expression appears multiple times in the program, it will share a graph node and the preview will be reused. Second, when binding modified source code, the process reuses graph nodes and so previews are also reused during code editing.

6 Evaluating live programming environment

Computing previews using a dependency graph implements a correct and efficient optimization. In this section we show that this is the case, first theoretically in section 6.2, and then empirically in section 6.3. We also describe a case study where we developed an online service for data exploration based on the methods discussed in this paper (section 6.4).

6.1 Correctness of previews

To show that the previews are correct, we prove two properties. Correctness (theorem 6) guarantees that, the previews we calculate using a dependency graph are the same as the values we would obtain by evaluating the program directly. Determinacy (theorem 7) guarantees that previews assigned to a graph node based on an earlier graph are the same as previews that we would obtain afres using an updated graph.

To simplify the proofs, we consider programs without let bindings. Eliminating let bindings does not change the result of evaluation, as shown in lemma 2, and it also does not change the constructed dependency graph as shown below in lemma 3.

Lemma 3 (Let elimination for a dependency graph). *Given programs p_1, p_2 such that $p_1 \rightsquigarrow_{\text{let}} p_2$ and a lookup table Δ_0 then if $v_1; \dots; v_n, (V, E) = \text{bind-prog}_{\emptyset, \Delta_0}(p_1)$ and $v'_1; \dots; v'_n, (V', E') = \text{bind-prog}_{\emptyset, \Delta_1}(p_2)$ such that $\Delta_1 = \text{update}_{V, E}(\Delta_0)$ then for all i , $v_i = v'_i$ and also $(V, E) = (V', E')$.*

Proof. By analysis of the binding process. See appendix A.3. □

Foundations of a live data exploration environment

The lemma 3 provides a way of removing let bindings from a program, such that the resulting dependency graph remains the same. Here, we bind the original program first, which adds the node for e to Δ . In our implementation, this is not needed because Δ is updated while the graph is being constructed using `bind-expr`. To keep the formalisation simpler, we separate the process of building the dependency graph and updating Δ and thus lemma 3 requires an extra binding step.

Now, we can show that, given a let-free expression, the preview obtained using a correctly constructed dependency graph is the same as the one we would obtain by directly evaluating the expression. This requires a simple auxiliary lemma.

Lemma 4 (Lookup inversion). *Given Δ obtained using update in figure 7 then:*

- If $v = \Delta(\text{fun}(x), [(v_0, l_0)])$ then $v = \text{fun}(x, s)$ for some s .
- If $v = \Delta(\text{mem}(m), [(v_0, l_0), \dots, (v_n, l_n)])$ then $v = \text{mem}(m, s)$ for some s .

Proof. By construction of Δ in figure 7. □

Theorem 5 (Term preview correctness). *Given a term t that has no free variables, together with a lookup table Δ obtained from any sequence of programs using `bind-prog` (figure 6) and update (figure 7), then let $v, (V, E) = \text{bind-expr}_{\emptyset, \Delta}(t)$.*

If $v \Downarrow p$ over a graph (V, E) then $p = o$ for some value o and $t \rightsquigarrow^ o$.*

Proof. By induction over the binding process. See appendix A.4. □

Theorem 6 (Program preview correctness). *Consider a program $p = c_1; \dots; c_n$ that has no free variables, together with a lookup table Δ_0 obtained from any sequence of programs using `bind-prog` (figure 6) and update (figure 7). Assume a let-free program $p' = t_1; \dots; t_n$ such that $p \rightsquigarrow_{\text{let}}^* p'$.*

Let $v_1; \dots; v_n, (V, E) = \text{bind-prog}_{\emptyset, \Delta_0}(p)$ and define updated lookup table $\Delta_1 = \text{update}_{V, E}(\Delta_0)$ and let $v'_1; \dots; v'_n, (V', E') = \text{bind-prog}_{\emptyset, \Delta_1}(p)$.

If $v'_i \Downarrow p_i$ over a graph (V', E') then $p_i = o_i$ for some value o_i and $t_i \rightsquigarrow o_i$.

Proof. Direct consequence of lemma 3 and theorem 5. □

Our implementation updates Δ during the recursive binding process and so a stronger version of the property holds: previews calculated over a graph obtained directly for the original program p are the same as the values of the fully evaluated program. Our formalisation omits this for simplicity.

The second important property is determinacy, which makes it possible to cache the previews evaluated via \Downarrow using the corresponding graph node as a lookup key.

Theorem 7 (Preview determinacy). *For some Δ and for any programs p, p' , assume that the first program is bound, i. e. $v_1; \dots; v_n, (V, E) = \text{bind-prog}_{\emptyset, \Delta}(p)$, the graph node cache is updated $\Delta' = \text{update}_{V, E}(\Delta)$ and the second program is bound, i. e. $v'_1; \dots; v'_m, (V', E') = \text{bind-prog}_{\emptyset, \Delta'}(p')$. Now, for any v , if $v \Downarrow p$ over (V, E) then also $v \Downarrow p$ over (V', E') .*

Proof. By induction over \Downarrow , show that the same evaluation rules also apply over (V', E') . This is the case, because graph nodes added to Δ' by $\text{update}_{V, E}$ are added as new nodes in $\text{bind-prog}_{\emptyset, \Delta'}$ and nodes and edges of (V, E) are unaffected. □

Edit contexts of expressions

$$\begin{aligned}
K_e[-] &= K_e[-].m(e_1, \dots, e_n) \mid e.m(e_1, \dots, e_{l-1}, K_e[-], e_{l+1}, \dots, e_n) \mid - \\
K_c[-] &= \text{let } x = K_e[-] \mid K_e[-]
\end{aligned}$$

Code edit operations preserving preview for a sub-expression

- (let-intro-var) $\bar{c}_1; \langle e \rangle; \bar{c}_2$ changes to $\bar{c}_1; \text{let } x = e; \langle x \rangle; \bar{c}_2$ where x is fresh.
- (let-intro-ins) $\bar{c}_1; \bar{c}_2; \langle K_c[e] \rangle; \bar{c}_3$ is changed to $\bar{c}_1; \text{let } x = e; \bar{c}_2; \langle K_c[x] \rangle; \bar{c}_3$ via a semantically non-equivalent expression $\bar{c}_1; \bar{c}_2; K_c[x]; \bar{c}_3$ where x is free.
- (let-intro-del) $\bar{c}_1; \bar{c}_2; \langle K_c[e] \rangle; \bar{c}_3$ is changed to $\bar{c}_1; \text{let } x = e; \bar{c}_2; \langle K_c[x] \rangle; \bar{c}_3$ via an expression $\bar{c}_1; \text{let } x = e; \bar{c}_2; K_c[e]; \bar{c}_3$ with unused variable x .
- (let-elim-del) $\bar{c}_1; \text{let } x = e; \bar{c}_2; \langle K_c[x] \rangle; \bar{c}_3$ is changed to $\bar{c}_1; \bar{c}_2; \langle K_c[e] \rangle; \bar{c}_3$ via a semantically non-equivalent expression $\bar{c}_1; \bar{c}_2; K_c[x]; \bar{c}_3$ where x is free.
- (let-elim-ins) $\bar{c}_1; \text{let } x = e; \bar{c}_2; \langle K_c[x] \rangle; \bar{c}_3$ is changed to $\bar{c}_1; \bar{c}_2; \langle K_c[e] \rangle; \bar{c}_3$ via an expression $\bar{c}_1; \text{let } x = e; \bar{c}_2; K_c[e]; \bar{c}_3$ with unused variable x .
- (edit-mem) $\bar{c}_1; K_c[\langle e_0 \rangle.m(\bar{e})]; \bar{c}_2$ is changed to $\bar{c}_1; K_c[\langle e_0 \rangle.m'(\bar{e}')]; \bar{c}_2$
- (edit-let) $\bar{c}_1; \text{let } x = e_1; \bar{c}_2; K_c[\langle e_2 \rangle]; \bar{c}_3$ is changed to $\bar{c}_1; \text{let } x = e'_1; \bar{c}_2; K_c[\langle e_2 \rangle]; \bar{c}_3$ when $x \notin FV(e_2)$.

■ **Figure 9** Code edit operations that preserve previously evaluated preview

The cache of previews (section 5.3) associates a preview d with a node v as the key. Theorem 7 guarantees that this is valid. As we update dependency graph during code editing, previous nodes will continue representing the same sub-expressions.

6.2 Reuse of previews

In this section, we identify a number of code edit operations where the previously evaluated values for a sub-expression can be reused. This includes the motivating example from section 1 where the data analyst extracted a constant into a let binding and modified a parameter of the last method call in a call chain.

The list of preview-preserving edits is shown in figure 9. It includes several ways of introducing and eliminating let bindings and edits where the analyst modifies an unrelated part of the program. The list is not exhaustive. Rather, it illustrates typical edits that the data analyst might perform when writing code. To express the operations we define an editing context K which is similar to evaluation context C from figure 3, but allows sub-expressions appearing anywhere in the program.

We use the notation $\langle e \rangle$ to mark parts of expressions that are not recomputed during the edit; we write \bar{c} and \bar{e} for a list of commands and expressions, respectively. In some of the edit operations, we also specify an intermediate program that may be semantically different and only has a partial preview. This illustrates a typical way of working with code in a text editor using cut and paste. For example, in (let-intro-ins),

Foundations of a live data exploration environment

the analyst cuts a sub-expression e , replaces it with a variable x and then adds a let binding for a variable x and inserts the expression e from the clipboard. The (let-intro-del) operation captures the same edit, but done in a different order.

Theorem 8 proves that the operations given in figure 9 preserve the preview for a marked sub-expression. It relies on a lemma 13 given in appendix A.5 that generally characterizes one common kind of edits. Given two versions of a program that both contain the same sub-expression e , if the let bindings that define the values of variables used in e do not change, then the graph node assigned to e will be the same when binding the original and the updated program.

Theorem 8 (Preview reuse). *Given the sequence of expressions as specified in figure 9, if the expressions are bound in sequence and graph node cache updated as specified in figure 7, then the graph nodes assigned to the specified sub-expressions are the same.*

Proof. Cases (edit-let) and (edit-mem) are direct consequences of lemma 13; for (let-intro-var), the node assigned to x is the node assigned to e which is the same as before the edit from lemma 13. Cases (let-intro-ins) and (let-intro-del) are similar to (let-intro-var), but also require using induction over the binding of $K_c[e]$. Finally, cases (let-elim-ins) and (let-elim-del) are similar and also use lemma 13 together with induction over the binding of $K_c[x]$. \square

6.3 Empirical evaluation of efficiency

The key performance claim about our method of providing instant feedback is that it is more efficient than recomputing values for the whole program (or the current command) after every keystroke. In the previous section, we formally proved that this is true and gave examples of code edit operations that do not cause recomputation. In this section, we further support this claim with an empirical evaluation. The purpose of this section is not to precisely evaluate overheads of our implementation, but to compare how much recomputation different evaluation strategies perform.

For the purpose of the evaluation, we use a simple image manipulation library that provides operations for loading, greyscaling, blurring and combining images. We compare delays in updating the preview for three different evaluation strategies, while performing the same sequence of code edit operations. Using image processing as an example gives us a way to visualize the reuse of previously computed values. As in a typical data exploration scenario, the individual operations are relatively expensive compared to the overheads of building the dependency graph.

To avoid distractions when visualizing the performance, we update the preview after complete tokens are added rather than after individual keystrokes. Figure 10 shows the sequence of edits that we use to measure the delays in updating a instant preview. We first enter an expression to load, greyscale and blur an image (1) then introduce let binding (2) and add more operations (3). Finally, we extract one of the parameters into a variable (4). Most of the operations are simply adding code, but there are two cases where we modify existing code and change value of a parameter for blur and combine immediately after (1) and (3), respectively.

(1) Enter the following code and then change parameter of blur from 4 to 8:

```
image.load("shadow.png").greyScale().blur(4)
```

(2) Assign the result to a variable and start writing code for further operations:

```
let shadow = image.load("shadow.png").greyScale().blur(8)
shadow.combine
```

(3) Finish code to combine two images and change parameter of combine from 20 to 80:

```
let shadow = image.load("shadow.png").greyScale().blur(8)
shadow.combine(image.load("poppe.png"), 20)
```

(4) Extract the parameter of combine to a let bound variable:

```
let ratio = 80
let shadow = image.load("shadow.png").greyScale().blur(8)
shadow.combine(image.load("poppe.png"), ratio)
```

■ **Figure 10** Code edit operations that are used in the experimental evaluation

We implement the algorithm described in section 4 section 5 in a simple web-based environment that allows the user to modify code and explicitly trigger recomputation. It then measures time needed to recompute values for the whole program and displays the resulting image. If the parsing fails, we record only the time taken by parsing attempt. We compare the delays of three different evaluation strategies:

Call-by-value Following the semantics in section 3.2, all sub-expressions are evaluated before an expression. This is often wasteful. For example, we parse the expression `image.load("shadow.png").blur` as a member access with no arguments. The evaluation loads the image, but then fails because `blur` requires one argument.

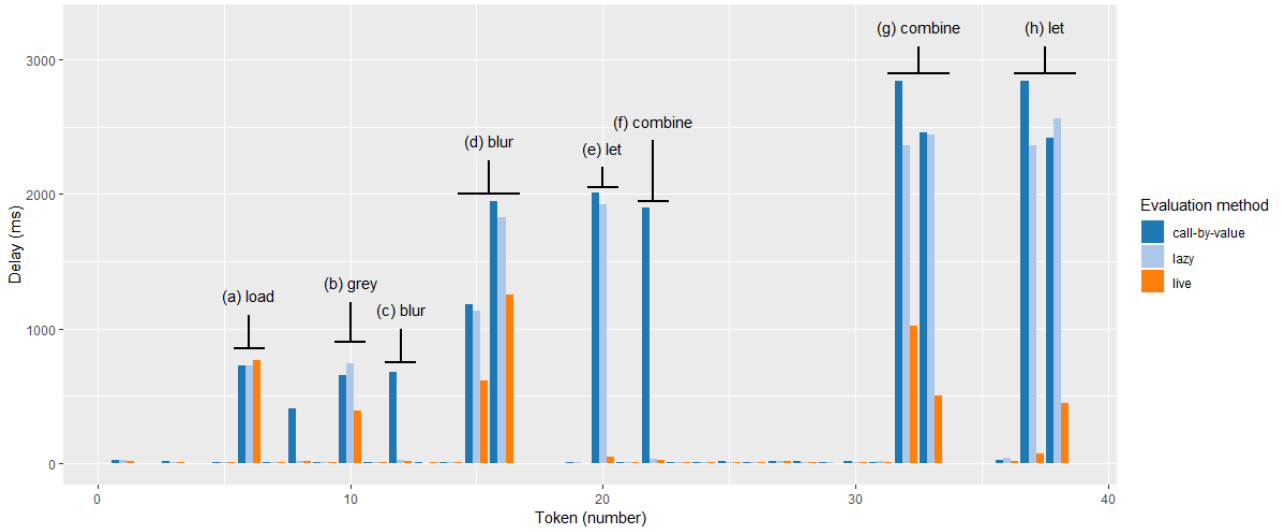
Lazy To address the wastefulness of call-by-value strategy, we simulate lazy evaluation by implementing a version of the image processing library where operations build a delayed computation and only evaluate it when rendering an image. Using this strategy, failing computations do not perform unnecessary work.

Live Finally, we use the algorithm described in section 4 section 5. The cache is empty at the beginning of the experiment and we update it after each token is added. This is the only strategy where evaluation does not start afresh after reparsing code.

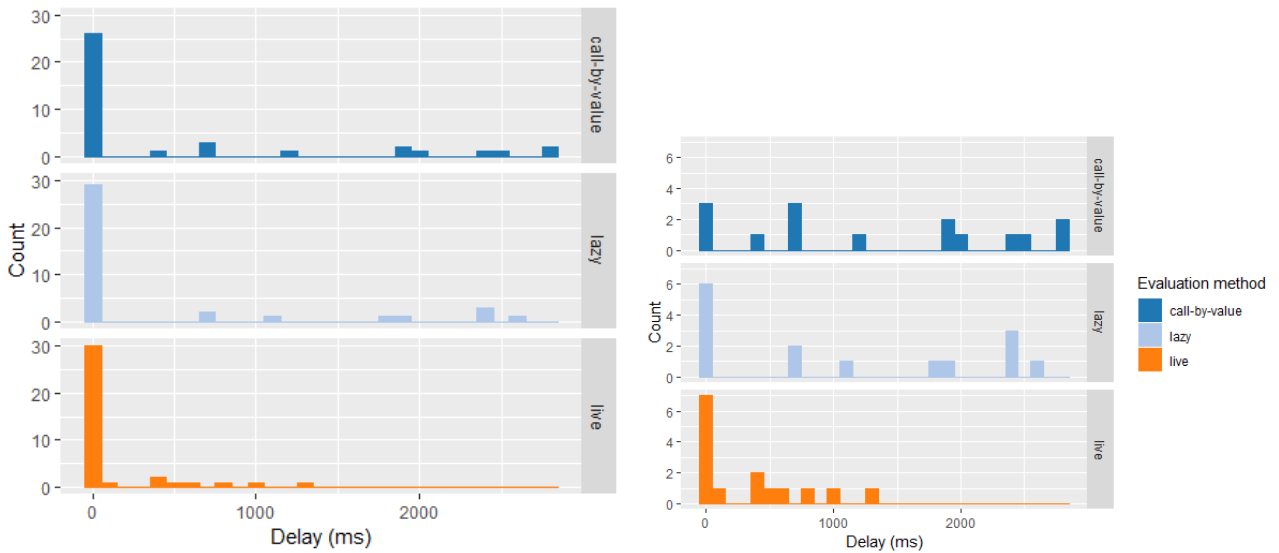
The experimental environment is implemented in F# and compiled to JavaScript using the Fable compiler. We run the experiments in Firefox (version 64.0.2, 32bit) on Windows 10 (build 1809, 64 bit) on Intel Core i7-7660U CPU with 16 GB RAM.

Figure 11 shows times needed to recompute previews after individual tokens are added, deleted or modified, according to the script in figure 10, resulting in 38 measurements. We mark a number of notable points in the chart:

Foundations of a live data exploration environment



■ **Figure 11** Time required to recompute the results of a sample program after individual tokens are added or modified for three different evaluations strategies.



■ **Figure 12** Distribution of delays incurred when updating previews. We show a histogram computed from all delays (left) and only from delays larger than 15 ms (right).

- a. Loading image for the first time incurs small extra overhead in the live strategy.
- b. Greyscaling using the live strategy does not need to re-load the image.
- c. Accessing the blur member without arguments causes delay for call-by-value.
- d. When varying the parameter of blur, the live strategy reuses the greyscaled image.
- e. Introducing let binding does not cause recomputation when using live strategy.
- f. As in (c), accessing a member without an argument only affects call-by-value.
- g. The live strategy is much faster when varying the parameter of combine.
- h. Introducing let binding, again, causes full recomputation for lazy and call-by-value.

A summarized view of the delays is provided in figure 12, which shows histograms illustrating the distribution of delays for each of the three evaluation methods. A large proportion of delays is very small (less than 15 ms) because the parser used in our experimental environment often fails (e.g. for unclosed parentheses). The histogram on the right summarizes only delays for edit operations where the delay for at least one of the strategies was over 15 ms. The histogram shows that the live strategy eliminates the longest delays (by caching partial results), with the exception of a few where the underlying operation takes a long time (such as blurring the image). The results would be even more significant with an error-recovering parser.

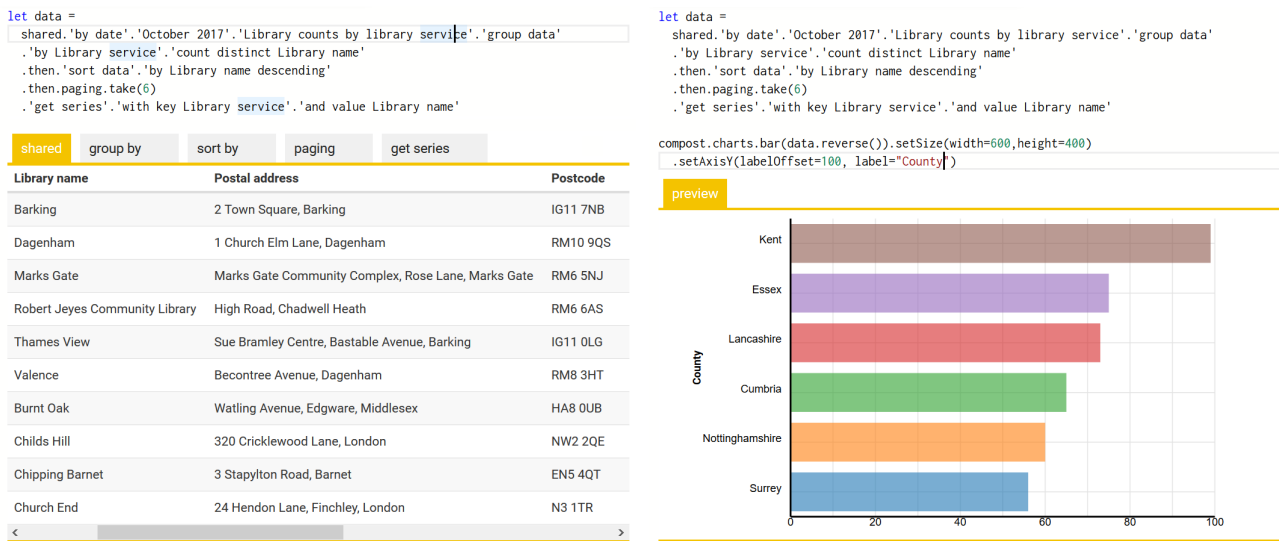
The purpose of our experiment is not to exactly assess the overhead of our implementation. Our goal is to illustrate how often can previously evaluated results be reused and the impact this has when writing code. The experiment presented in this section is small-scale, but it is sufficient for this purpose. When recomputing results after every edit using the *call-by-value* strategy, the time needed to update results grows continually. The *lazy* strategy removes the overhead for programs that fail, but keeps the same trend. Our *live* strategy reuses values computed previously. Consequently, expensive operations such as (d) and (g) in figure 11 are significantly faster, because they do not need to recompute operations done previously when writing the code. As shown in figure 12, there are almost no very expensive operations (taking over 1 second) in the *live* strategy in contrast to several in the other two strategies.

6.4 Transparent tools for data journalism

In section 2.1, we motivated our work by considering how journalists explore open data. In addition to the theoretical and experimental work presented in this paper, we also implemented an online data exploration environment, equipped with live editor for The Gamma language that provides instant feedback during coding. The environment uses the principles presented in this paper to build a more comprehensive system that allows users, such as journalists, to analyse, summarize and visualize open data. In this section, we briefly report on our experience with the system. Two screenshots shown in figure 13 illustrate a number of interesting features:

- The left screenshot uses a type provider for data aggregation [45]. Type providers are treated as an external library (with objects, members and reduction relation). Type providers also rely on type information to provide editor auto-complete, which we support by implementing type checking over a dependency graph (appendix C).

Foundations of a live data exploration environment



■ **Figure 13** Data analysis counting the number of libraries per county in the UK. We load and aggregate data (left) and then create a chart with a label (right). The example has been created using our tool discussed in section 6.4 by a non-programmer and is available at <http://gallery.thegamma.net/73/>.

- When displaying instant preview for code written using the data aggregation type provider (left), our environment generates tabs that show individual steps of the data transformation. A tab is selected based on the cursor position and shows a preview for the current sub-expression (e.g. raw data before grouping).
- Another external library provides support for charting (right). Here, the environment displays the result of evaluating the whole command. The screenshot shows a case where the user modifies parameters of the chart. Thanks to our live evaluation strategy, this is done efficiently without reevaluating the data transformation.

The environment is available at <http://gallery.thegamma.net>. A user study to evaluate the usability of the system from a human-computer interaction perspective is left for future work. As an anecdotal evidence, the code in figure 13 was developed by an attendee of a Mozilla Festival 2017 who had no prior programming experience.

7 Related and future work

Simple data exploration performed, for example, by journalists [20] is done either programmatically or using spreadsheets. The latter is easy but error-prone while the former requires expert programming skills. We aim to bring liveness of spreadsheets to programmatic data exploration. This requires extending recomputation as done in spreadsheets [52] to code written in an ordinary text editor.

Notebooks and data science tools. Visual data exploration tools are interactive [11, 24, 60] and some can export the workflow as a script [26], but data analysts who

prefer code typically resort to notebook systems such as Jupyter or R Markdown [5, 29]. Those are text-based, but have a limited model of recomputation. Users structure code in cells and manually reevaluate cells. Many notebook systems are based on the REPL (read-eval-print-loop) [16, 34] and do not track dependencies between cells, which can lead to well-documented inconsistencies [30, 46, 49].

Ideas such as dependency tracking and efficient recomputation exist in visual data exploration tools [11, 24, 60] and scientific workflow systems [6, 41]. The implementation techniques are related to our work, but we focus on text-based scripts. Tempe [13] focuses on streaming data, but provides a text-based scripting environment with automatic code update; its usability in contrast to REPLs has been empirically evaluated [12].

Live and exploratory programming. Live programming based on textual programs has been popularised by Victor [57, 58] and is actively developed in domains such as live coded music [1, 51]. The notions of exploratory and live programming have been extensively studied [50]. The notion of exploratory programming has recently been analysed from the perspective of human-computer interaction [28], which led to new tools [27], complementary to our instant previews. Kubelka, Robbes, and Bergel [31] review the use of live programming in a Smalltalk derived environment. Lighttable [19] and Chrome DevTools provide limited instant previews akin to those presented in this paper, but without well specified recomputation model. Finally, work on keeping state during code edits [8, 35] would be relevant for supporting streaming data.

A more principled approach can be used by systems based on structured editing [33, 42, 44, 55] where code is only modified via known operations with known effect on the computation graph (e.g. “extract variable” has no effect on the result; “change constant value” forces recomputation of subsequent code). This can be elegantly implemented using bi-directional lambda calculus [43], but it also makes us consider more human-centric abstractions [14, 15] further discussed in appendix D.

Incremental computation and dependency analysis. Work on self-adjusting and incremental computation [2, 23] handles recomputation when the program stays the same, but input changes. Most incremental systems, e.g. [3, 4, 22] evaluate the program and use programmer-supplied information to build a dependency graph, whereas our system uses static code analysis; [23] implement a small adaptive interpreter that treats code as changing input data, suggesting an implementation technique for live programming systems. Our use of dependency graphs [32] is static and first-order and can be seen as a form of program slicing [59], although our binding process is more directly inspired by Roslyn [40], which uses it for efficient background type-checking.

Semantics and partial evaluation. The evaluation of previews is a form of partial evaluation [10], done in a way that allows caching. This can be done implicitly or explicitly in the form of multi-stage programming [56]. Semantically, the evaluation of previews can be seen as a modality and delayed previews are linked to contextual modal type theory [39], formally modelled using comonads [17].

8 Summary

One of the aspects that make spreadsheets easier to use than programming tools is that they provide instant feedback. We aim to make programming tools as instant as spreadsheets. We described a number of key aspects of simple data analyses such as those done by journalists and then used our observations to build both theory and simple practical data analytics tools.

Our *data exploration calculus* is a simple formally tractable language for data exploration. The calculus captures key observations about simple data analyses. They rely on logic defined by external libraries, implement few abstractions and are written as lists of commands.

Our main technical contribution is a *instant preview* mechanism that efficiently evaluates code during editing and instantly provides a preview of the result. We allow users to edit code in unconstrained way in a text editor, which makes this particularly challenging. The key trick is to separate the process into a fast *binding phase*, which constructs a dependency graph and a slower *evaluation phase* that can cache results. This makes it possible to quickly parse updated code, reconstruct dependency graph and compute preview using previous, partially evaluated, results.

We evaluated our approach in three ways. First, we proved that our mechanism is correct and that it reuses evaluated values for many common code edit operations. Second, we conducted an experimental study that illustrates how often are previously evaluated results reused during typical programming scenario. Thirdly, we used our research as a basis for online data exploration environment, which shows the practical usability of our work.

Acknowledgements We thank Dominic Orchard, Stephen Kell, Roly Perera and Jonathan Edwards for many discussions about the work presented in the paper. Dominic Orchard provided invaluable feedback on earlier version of the paper. The paper also benefited from suggestions made by anonymous reviewers of The Programming Journal as well as reviewers of earlier versions of the paper. This work was partly supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1.

References

- [1] Samuel Aaron and Alan F. Blackwell. “From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages”. In: *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*. Edited by Paul Hudak and Conal Elliott. ACM. 2013, pages 35–46. DOI: 10.1145/2505341.2505346.
- [2] Umut A. Acar. “Self-adjusting Computation”. AAI3166271. PhD thesis. Pittsburgh, PA, USA, 2005. ISBN: 0-542-01547-1.

- [3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. “Adaptive Functional Programming”. In: *ACM Transactions on Programming Languages and Systems* 28.6 (Nov. 2006), pages 990–1034. ISSN: 0164-0925. DOI: 10.1145/1186632.1186634.
- [4] Umut A. Acar and Ruy Ley-Wild. “Self-adjusting Computation with Delta ML”. In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Edited by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Volume 5832. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 1–38. ISBN: 978-3-642-04652-0. DOI: 10.1007/978-3-642-04652-0_1.
- [5] JJ Allaire, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, Winston Chang, and Richard Iannone. *rmarkdown: Dynamic Documents for R*. 2016. URL: <https://github.com/rstudio/rmarkdown> (visited on 2020-01-31).
- [6] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. “Kepler: an extensible system for design and execution of scientific workflows”. In: *Scientific and Statistical Database Management*. Edited by Michael Hatzopoulos. IEEE. 2004, pages 423–424. DOI: 10.1109/SSDM.2004.1311241.
- [7] David Blood. *Recycling is broken – notebooks*. Nov. 2018. URL: <https://github.com/ft-interactive/recycling-is-broken-notebooks> (visited on 2019-02-01).
- [8] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. “It’s Alive! Continuous Feedback in UI Programming”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Edited by Hans Boehm and Cormac Flanagan. PLDI ’13. ACM SIGPLAN, June 2013. DOI: 10.1145/2491956.2462170.
- [9] David Raymond Christiansen. “Dependent type providers”. In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*. Edited by Jacques Carette and Jeremiah James Willcock. ACM. 2013, pages 25–34. DOI: 10.1145/2502488.2502495.
- [10] Charles Consel and Olivier Danvy. “Tutorial notes on partial evaluation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Edited by Mary Van Deusen and Bernard Lang. ACM. 1993, pages 493–501. DOI: 10.1145/158511.158707.
- [11] Andrew Crotty, Alex Galakatos, Emanuel Zraggen, Carsten Binnig, and Tim Kraska. “Vizdom: Interactive Analytics Through Pen and Touch”. In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pages 2024–2027. ISSN: 2150-8097. DOI: 10.14778/2824032.2824127.
- [12] Rob DeLine and Daniel Fisher. “Supporting exploratory data analysis with live programming”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Edited by Zhen Li, Claudia Ermel, and Scott Fleming. Oct. 2015, pages 111–119. DOI: 10.1109/VLHCC.2015.7357205.

Foundations of a live data exploration environment

- [13] Rob DeLine, Daniel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James Terwilliger, and John Wernsing. “Tempe: Live scripting for live data”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Edited by Zhen Li, Claudia Ermel, and Scott Fleming. Oct. 2015, pages 137–141. DOI: 10.1109/VLHCC.2015.7357208.
- [14] Jonathan Edwards. *Direct Programming*. June 2018. URL: <https://vimeo.com/274771188> (visited on 2019-02-09).
- [15] Jonathan Edwards. “Subtext: uncovering the simplicity of programming”. In: *ACM SIGPLAN Notices* 40.10 (2005), pages 505–518. DOI: 10.1145/1103845.1094851.
- [16] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. “DrScheme: A programming environment for Scheme”. In: *Journal of functional programming* 12.2 (2002), pages 159–182. DOI: 10.1017/S0956796801004208.
- [17] Murdoch J. Gabbay and Aleksandar Nanevski. “Denotation of contextual modal type theory (CMTT): Syntax and meta-programming”. In: *Journal of Applied Logic* 11.1 (2013), pages 1–29. DOI: 10.1016/j.jal.2012.07.002.
- [18] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvert, and Tarmo Uustalu. “Combining Effects and Coeffects via Grading”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. Edited by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pages 476–489. ISBN: 9781450342193. DOI: 10.1145/2951913.2951939. URL: <https://doi.org/10.1145/2951913.2951939>.
- [19] Chris Granger. *LightTable: A new IDE concept*. 2012. URL: <http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/> (visited on 2020-01-31).
- [20] Jonathan Gray, Lucy Chambers, and Liliana Bounegru. *The data journalism handbook: how journalists can use data to improve the news*. O’Reilly Media, Inc., 2012. ISBN: 978-1449330064.
- [21] Philip Guo. *Data science workflow: Overview and challenges*. Blog@CACM, Communications of the ACM. 2013. URL: <https://cacm.acm.org/blogs/blog-cacm/169199/fulltext> (visited on 2020-01-31).
- [22] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. “Incremental Computation with Names”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Edited by Aldrich Jonathan and Patrick Eugster. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pages 748–766. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814305.

- [23] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. “Adapton: Composable, Demand-driven Incremental Computation”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Edited by Michael O’Boyle and Keshav Pingali. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pages 156–166. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594324.
- [24] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. “Interactive data analysis: the Control project”. In: *Computer* 32.8 (Aug. 1999), pages 51–59. ISSN: 0018-9162. DOI: 10.1109/2.781635.
- [25] Leslie Hook and John Reed. *Why the world’s recycling system stopped working*. Sept. 2018. URL: <https://www.ft.com/content/360e2524-d71a-11e8-a854-33d6f82e62f8>.
- [26] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. “Wrangler: Interactive Visual Specification of Data Transformation Scripts”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Edited by Desney Tan, Geraldine Fitzpatrick, Carl Gutwin, Bo Begole, and Wendy Kellogg. CHI ’11. Vancouver, BC, Canada: ACM, 2011, pages 3363–3372. ISBN: 978-1-4503-0228-9. DOI: 10.1145/1978942.1979444.
- [27] Mary Beth Kery, Amber Horvath, and Brad Myers. “Variolite: Supporting Exploratory Programming by Data Scientists”. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. Edited by Gloria Mark, Susan Fussell, Cliff Lampe, Monica Schraefel, Juan Pablo Hourcade, Caroline Appert, and Daniel Wigdor. CHI ’17. Denver, Colorado, USA: Association for Computing Machinery, 2017, pages 1265–1276. ISBN: 9781450346559. DOI: 10.1145/3025453.3025626.
- [28] Mary Beth Kery and Brad A Myers. “Exploring exploratory programming”. In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Edited by Austin Henley, Peter Rogers, and Anita Sarma. IEEE, 2017, pages 25–29. DOI: 10.1109/VLHCC.2017.8103446.
- [29] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. “Jupyter Notebooks—a publishing format for reproducible computational workflows”. In: *20th International Conference on Electronic Publishing*. Edited by Fernando Loizides and Birgit Schmidt. 2016, pages 87–90. DOI: 10.3233/978-1-61499-649-1-87.
- [30] David Koop and Jay Patel. “Dataflow Notebooks: Encoding and Tracking Dependencies of Cells”. In: *9th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2017, Seattle, WA, USA, June 23, 2017*. Edited by Adam Bates and Bill Howe. 2017. DOI: 10.5555/3183865.3183888.

Foundations of a live data exploration environment

- [31] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. “The Road to Live Programming: Insights from the Practice”. In: *Proceedings of the 40th International Conference on Software Engineering*. Edited by Andrea Zisman and Sven Apel. ICSE '18. Gothenburg, Sweden: ACM, 2018, pages 1090–1101. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180200.
- [32] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. “Dependence graphs and compiler optimizations”. In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Edited by John White, Richard Jay Lipton, and Patricia Goldberg. ACM, 1981, pages 207–218. DOI: 10.1145/567532.567555.
- [33] Eyal Lotem and Yair Chuchem. *Lamdu Project*. 2018. URL: <https://github.com/lamdu/lamdu> (visited on 2020-01-31).
- [34] John McCarthy. “History of LISP”. In: *ACM SIGPLAN Notices* 13.8 (Aug. 1978), pages 217–223. ISSN: 0362-1340. DOI: 10.1145/960118.808387.
- [35] Sean McDirmid. “Living It up with a Live Programming Language”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. Edited by Richard Gabriel, David Bacon, Cristina Lopes, and Guy L. Steele Jr. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pages 623–638. ISBN: 9781595937865. DOI: 10.1145/1297027.1297073.
- [36] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012. ISBN: 978-1449319793.
- [37] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pages 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4.
- [38] Alan Mycroft, Dominic Orchard, and Tomas Petricek. “Effect systems revisited—control-flow algebra and semantics”. In: *Semantics, Logics, and Calculi*. Springer, 2016, pages 1–32. DOI: 10.1007/978-3-319-27810-0_1.
- [39] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. “Contextual modal type theory”. In: *ACM Transactions on Computational Logic (TOCL)* 9.3 (2008), page 23. DOI: 10.1145/1352582.1352591.
- [40] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. *The Roslyn Project, Exposing the C# and VB compiler’s code analysis*. Technical report. Microsoft, 2011. URL: <https://www.microsoft.com/en-us/download/details.aspx?id=27744> (visited on 2020-02-13).
- [41] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, and Peter Li. “Taverna: a tool for the composition and enactment of bioinformatics workflows”. In: *Bioinformatics* 20.17 (2004), pages 3045–3054. DOI: 10.1093/bioinformatics/bth361.

- [42] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. “Live Functional Programming with Typed Holes”. In: *PACMPL* 3.POPL (2019). DOI: 10.1145/3291622.
- [43] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. “Hazelnut: a bidirectionally typed structure editor calculus”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Edited by Giuseppe Castagna and Andrew Gordon. ACM. 2017, pages 86–99. DOI: 10.1145/3009837.3009900.
- [44] Roland Perera. “Interactive functional programming”. PhD thesis. University of Birmingham, 2013. URL: <https://etheses.bham.ac.uk/id/eprint/4209/> (visited on 2020-02-13).
- [45] Tomas Petricek. “Data Exploration through Dot-driven Development”. In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Edited by Peter Müller. Volume 74. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 21:1–21:27. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.21. URL: <http://drops.dagstuhl.de/opus/volltexte/2017/7261>.
- [46] Tomas Petricek, James Geddes, and Charles A. Sutton. “Wrattler: Reproducible, live and polyglot notebooks”. In: *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*. Edited by Melanie Herschel. 2018.
- [47] Tomas Petricek, Gustavo Guerra, and Don Syme. “Types from Data: Making Structured Data First-Class Citizens in F#”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Edited by Chandra Krintz and Emery Berger. PLDI ’16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pages 477–490. ISBN: 9781450342612. DOI: 10.1145/2908080.2908115.
- [48] Tomas Petricek, Dominic Orchard, and Alan Mycroft. “Coeffects: A Calculus of Context-Dependent Computation”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. Edited by Johan Jeuring and Manuel M. T. Chakravarty. ICFP ’14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pages 123–135. ISBN: 9781450328739. DOI: 10.1145/2628136.2628160.
- [49] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. “Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow”. In: *Workshop on the Theory and Practice of Provenance (TaPP)*. Edited by Paolo Missier and Jun Zhao. 2015, pages 155–167.
- [50] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. “Exploratory and Live, Programming and Coding”. In: *The Art, Science, and Engineering of Programming* 3.1 (2019). DOI: 10.22152/programming-journal.org/2019/3/1.

Foundations of a live data exploration environment

- [51] Charles Roberts, Matthew Wright, and JoAnn Kuchera-Morin. “Beyond Editing: Extended Interaction with Textual Code Fragments”. In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Edited by Edgar Berdahl and Jesse Allison. NIME 2015. Baton Rouge, Louisiana, USA: The School of Music, the Center for Computation, and Technology (CCT), Louisiana State University, 2015, pages 126–131. ISBN: 9780692495476.
- [52] Peter Sestoft. *Spreadsheet Implementation Technology: Basics and Extensions*. MIT Press, 2012. ISBN: 978-0262526647.
- [53] Don Syme. “Leveraging .net meta-programming components from F#: integrated queries and interoperable heterogeneous execution”. In: *Proceedings of the 2006 workshop on ML*. Edited by Andrew Kennedy and François Pottier. ACM, 2006, pages 43–54. DOI: 10.1145/1159876.1159884.
- [54] 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 the 2013 workshop on Data driven functional programming*. Edited by Evelyne Viegas, Karin Breitman, and Judith Bishop. ACM, 2013, pages 1–4. DOI: 10.1145/2429376.2429378.
- [55] Gerd Szwillus and Lisa Neal. *Structure-based editors and environments*. Academic Press, Inc., 1996. ISBN: 978-0126818901.
- [56] Walid Taha and Tim Sheard. “MetaML and multi-stage programming with explicit annotations”. In: *Theoretical computer science* 248.1 (2000), pages 211–242. DOI: 10.1016/S0304-3975(00)00053-0.
- [57] Bret Victor. *Inventing on Principle*. 2012. URL: <http://worrydream.com/InventingOnPrinciple> (visited on 2020-01-31).
- [58] Bret Victor. *Learnable programming: Designing a programming system for understanding programs*. 2012. URL: <http://worrydream.com/LearnableProgramming> (visited on 2020-01-31).
- [59] Mark Weiser. “Program slicing”. In: *Proceedings of the 5th international conference on Software engineering*. Edited by Leon George Stucki. IEEE Press, 1981, pages 439–449. DOI: 10.1109/TSE.1984.5010248.
- [60] Richard Wesley, Matthew Eldridge, and Pawel T. Terlecki. “An Analytic Data Engine for Visualization in Tableau”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. Edited by Timos Sellis, Renée Miller, Anastasios Kementsietsidis, and Yannis Velegrakis. SIGMOD ’11. Athens, Greece: ACM, 2011, pages 1185–1194. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989449.

A Details of proofs

A.1 Normalization for data exploration calculus

Theorem 9 (Normalization). *For all p , there exists n and o_1, \dots, o_n such that $p \rightsquigarrow^* o_1; \dots; o_n$.*

Proof. We define size of a program in data exploration calculus as follows:

$$\begin{aligned}
 \text{size}(c_1; \dots; c_n) &= 1 + \sum_{i=1}^n \text{size}(c_i) \\
 \text{size}(\text{let } x = t) &= 1 + \text{size}(t) \\
 \text{size}(e_0.m(e_1, \dots, e_n)) &= 1 + \sum_{i=0}^n \text{size}(e_i) \\
 \text{size}(\lambda x \rightarrow e) &= 1 + \text{size}(e) \\
 \text{size}(o) = \text{size}(x) &= 1
 \end{aligned} \tag{10}$$

The property holds because, first, both (let) and (external) decrease the size of the program and, second, a program is either fully evaluated, i.e. $o_1; \dots; o_n$ for some n or, it can be reduced using one of the reduction rules. \square

A.2 Let elimination for a program

Lemma 10 (Let elimination for a program). *Given any program p such that $p \rightsquigarrow^* o_1; \dots; o_n$ for some n and o_1, \dots, o_n then if $p \rightsquigarrow_{\text{let}} p'$ for some p' then also $p' \rightsquigarrow^* o_1; \dots; o_n$.*

Proof. The elimination of let binding transforms a program $c_1; \dots; c_{k-1}; \text{let } x = t; c_{k+1}; \dots; c_n$ to a program $c_1; \dots; c_{k-1}; t; c_{k+1}[x \leftarrow t]; \dots; c_n[x \leftarrow t]$. The reduction steps for the new program can be constructed using the steps of $p \rightsquigarrow^* o_1; \dots; o_n$. The new command t reduces to an object o using the same steps as the original term t in $\text{let } x = t$ but with context $C_c = -$ rather than $C_c = \text{let } x = -$; the terms t introduced by substitution also reduce using the same steps as before, but using contexts in which the variable x originally appeared. \square

A.3 Let elimination for a dependency graph

Lemma 11 (Let elimination for a dependency graph). *Given programs p_1, p_2 such that $p_1 \rightsquigarrow_{\text{let}} p_2$ and a lookup table Δ_0 then if $v_1; \dots; v_n, (V, E) = \text{bind-prog}_{\emptyset, \Delta_0}(p_1)$ and $v'_1; \dots; v'_n, (V', E') = \text{bind-prog}_{\emptyset, \Delta_1}(p_2)$ such that $\Delta_1 = \text{update}_{V, E}(\Delta_0)$ then for all i , $v_i = v'_i$ and also $(V, E) = (V', E')$.*

Proof. Assume $p_1 = c_1; \dots; c_{k-1}; \text{let } x = e; c_{k+1}; \dots; c_n$ and the let binding is eliminated resulting in $p_2 = c_1; \dots; c_{k-1}; e; c_{k+1}[x \leftarrow e]; \dots; c_n[x \leftarrow e]$. When binding p_1 , the case $\text{bind-prog}_{\Gamma, \Delta}(\text{let } x = e)$ is handled using (7) and the node resulting from binding e is added to the graph V, E . It is then referenced each time x appears in subsequent commands $c_{k+1}; \dots; c_n$. When binding p_2 , the node resulting from binding e is a primitive value or a node already present in Δ_1 (added by $\text{update}_{V, E}$) and is reused each time $\text{bind-expr}_{\Gamma, \Delta_1}(e)$ is called. \square

A.4 Term preview correctness

Theorem 12 (Term preview correctness). *Given a term t that has no free variables, together with a lookup table Δ obtained from any sequence of programs using `bind-prog` (figure 6) and `update` (figure 7), then let $\nu, (V, E) = \text{bind-expr}_{\emptyset, \Delta}(t)$. If $\nu \Downarrow p$ over a graph (V, E) then $p = o$ for some value o and $t \rightsquigarrow^* o$.*

Proof. When combining recursively constructed sub-graphs, the `bind-expr` function adds new nodes and edges leading from those new nodes. Therefore, an evaluation using \Downarrow over a sub-graph will also be valid over the new graph – the newly added nodes and edges do not introduce non-determinism to the rules given in figure 8.

We prove a more general property showing that for any e , its binding $\nu, (V, E) = \text{bind-expr}_{\emptyset, \Delta}(e)$ and any evaluation context C such that $C[e] \rightsquigarrow o$ for some o , one of the following holds:

- a. If $FV(e) = \emptyset$ then $\nu \Downarrow p$ for some p and $C[p] \rightsquigarrow o$
- b. If $FV(e) \neq \emptyset$ then $\nu \Downarrow \llbracket e_p \rrbracket_{FV(e)}$ for some e_p and $C[e_p] \rightsquigarrow o$

In the first case, p is a value, but it is not always the case that $e \rightsquigarrow^* p$, because p may be lambda function and preview evaluation may reduce sub-expression in the body of the function. Using a context C in which the value reduces to an object avoids this problem.

The proof of the theorem follows from the more general property. Using a context $C[-] = -$, the term t reduces $t \rightsquigarrow^* t' \rightsquigarrow_e o$ for some o and the preview p is a value o because $C[p] = p = o$. The proof is by induction over the binding process, which follows the structure of the expression:

- (1) `bind-expr` $_{\Gamma, \Delta}(e_0.m(e_1, \dots, e_n))$ – Here $e = e_0.m(e_1, \dots, e_n)$, ν_i are graph nodes obtained by induction for expressions e_i and $\{(\nu, \nu_0, \text{arg}(0)), \dots, (\nu, \nu_n, \text{arg}(n))\} \subseteq E$. From lookup inversion lemma 4, $\nu = \text{mem}(m, s)$ for some s .
 If $FV(e) = \emptyset$, then $\nu_i \Downarrow p_i$ for $i \in 0 \dots n$ and $\nu \Downarrow p$ using (`mem-val`) such that $p_0.m(p_1, \dots, p_n) \rightsquigarrow p$. From induction hypothesis and *compositionality* of external libraries (definition 2), it holds that for any C such that $C[e_0.m(e_1, \dots, e_n)] \rightsquigarrow o$ for some o then also $C[p_0.m(p_1, \dots, p_n)] \rightsquigarrow C[p] \rightsquigarrow o$.
 If $FV(e) \neq \emptyset$, then $\nu_i \Downarrow_{\text{lift}} \llbracket e'_i \rrbracket$ for $i \in 0 \dots n$ and $\nu \Downarrow \llbracket e'_0.m(e'_1, \dots, e'_n) \rrbracket_{FV(e)}$ using (`mem-expr`). From induction hypothesis and *compositionality* of external libraries (definition 2), it holds that for any C such that $C[e_0.m(e_1, \dots, e_n)] \rightsquigarrow o$ for some o then also $C[e'_0.m(e'_1, \dots, e'_n)] \rightsquigarrow o$.
- (2) `bind-expr` $_{\Gamma, \Delta}(e_0.m(e_1, \dots, e_n))$ – This case is similar to (1), except that the fact that $\nu = \text{mem}(m, s)$ holds by construction, rather than using lemma 4.
- (3) `bind-expr` $_{\Gamma, \Delta}(\lambda x \rightarrow e_b)$ – Here $e = \lambda x \rightarrow e_b$, ν_b is the graph node obtained by induction for the expression e_b and $(\nu, \nu_b, \text{body}) \in E$. From lookup inversion lemma 4, $\nu = \text{fun}(x, s)$ for some s . The evaluation can use one of three rules:
 - i. If $FV(e) = \emptyset$ then $\nu_b \Downarrow p_b$ for some p_b and $\nu \Downarrow \lambda x \rightarrow p_b$ using (`fun-val`). Let $e'_b = p_b$.
 - ii. If $FV(e_b) = \{x\}$ then $\nu_b \Downarrow \llbracket e'_b \rrbracket_x$ for some e'_b and $\nu \Downarrow \lambda x \rightarrow e'_b$ using (`fun-bind`).
 - iii. Otherwise, $\nu_b \Downarrow \llbracket e'_b \rrbracket_{x, \Gamma}$ for some e'_b and $\nu \Downarrow \llbracket \lambda x \rightarrow e'_b \rrbracket_{\Gamma}$ using (`fun-expr`).

For i.) and ii.) we show that a.) is the case; for iii.) we show that b.) is the case; that is for any C , if $C[\lambda x \rightarrow e_b] \rightsquigarrow o$ then also $C[\lambda x \rightarrow e'_b] \rightsquigarrow o$. For a given C , let $C'[-] = C[\lambda x \rightarrow -]$ and use the induction hypothesis, i.e. if $C'[e_b] \rightsquigarrow o$ for some o then also $C'[e'_b] \rightsquigarrow o$.

- (4) $\text{bind-expr}_{\Gamma, \Delta}(\lambda x \rightarrow e)$ – This case is similar to (3), except that the fact that $v = \text{fun}(x, s)$ holds by construction, rather than using lemma 4.
- (5) $\text{bind-expr}_{\Gamma, \Delta}(o)$ – In this case $e = o$ and $v = \text{val}(o)$ and $\text{val}(o) \Downarrow o$ using (val) and so the case a.) trivially holds.
- (6) $\text{bind-expr}_{\Gamma, \Delta}(x)$ – The initial Γ is empty, so x must have been added to Γ by case (3) or (4). Hence, $v = \text{var}(x)$, $v \Downarrow \llbracket x \rrbracket_x$ using (var) and so $e_p = e = x$ and the case b.) trivially holds.

□

A.5 Binding sub-expressions

Lemma 13 (Binding sub-expressions). *Assume we have programs p_1, p_2 such that $p_1 = c_1; \dots; c_k; K_c[e]; c_{k+1}; \dots; c_n$ and $p_2 = c'_1; \dots; c'_k; K'_c[e]; c'_{k+1}; \dots; c'_n$ and $I \subseteq \{1 \dots k\}$ such that $\forall i \in I. c_i = c'_i$ and for each $x \in \bigcup_{i \in I} FV(c_i) \cup FV(e)$ there exists $j \in I$ such that $c_j = \text{let } x = e$ for some e . Given any Δ , assume that the first program is bound, i.e. $v_1; \dots; v_n, (V, E) = \text{bind-prog}_{\emptyset, \Delta}(p_1)$, the cache is updated $\Delta' = \text{update}_{V, E}(\Delta)$ and the second program is bound, i.e. $v'_1; \dots; v'_n, (V', E') = \text{bind-prog}_{\emptyset, \Delta'}(p_2)$.*

Now, assume $v, G = \text{bind-expr}_{\Gamma, \Delta}(e)$ and $v', G' = \text{bind-expr}_{\Gamma', \Delta'}(e)$ are the recursive calls to bind e during the first and the second binding, respectively. Then, the graph nodes assigned to the sub-expression e are the same, i.e. $v = v'$.

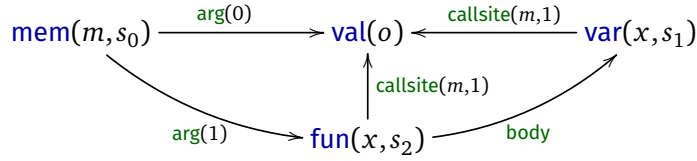
Proof. First, assuming that $\forall x \in FV(e). \Gamma(x) = \Gamma'(x)$, we show by induction over the binding process of e for the first program that the result is the same. In cases (1) and (3), the updated Δ' contains the required key and so the second binding proceeds using the same case. In cases (2) and (4), the second binding reuses the node created by the first binding using case (1) and (3), respectively. Cases (5) and (6) are the same.

Second, when binding let bindings in $c_1; \dots; c_k$, the initial $\Gamma = \emptyset$ during both bindings. Nodes added to Γ and Γ' for commands c_j such that $j \in I$ are the same and nodes added for remaining commands do not add any new nodes referenced from e and so $v = v'$ using the above. □

B Theories of delayed previews

The operational semantics presented in this paper serves two purposes. It gives a simple guide for implementing text-based live programming environments for data science and we use it to prove that our optimized way of producing instant previews is correct. However, some aspects of our mechanism are related to important work in semantics of programming languages and deserve to be mentioned.

Foundations of a live data exploration environment



■ **Figure 14** Dependency graph for $o.m(\lambda x \rightarrow x)$ with a newly added `callsite` edges.

The construction of delayed previews is related to meta-programming. Assuming we have delayed previews $\llbracket e_0 \rrbracket_x$ and $\llbracket e_1 \rrbracket_y$ and we invoke a member m on e_0 using e_1 as an argument. To do this, we construct a new delayed preview $\llbracket e_0.m(e_1) \rrbracket_{x,y}$. This operation is akin to expression splicing from meta-programming [53, 56].

The semantics of delayed previews can be more formally captured by Contextual Modal Type Theory (CMTT) [39] and comonads [17]. In CMTT, $[\Psi]A$ denotes that a proposition A is valid in context Ψ , which is similar to our delayed previews written as $\llbracket A \rrbracket_\Psi$. CMTT defines rules for composing context-dependent propositions that would allow us to express the splicing operation used in (mem-expr). In categorical terms, the context-dependent proposition can be modelled as a graded comonad [18, 38]. The evaluation of a preview with no context dependencies (built implicitly into our evaluation rules) corresponds to the counit operation of a comonad and would be explicitly written as $\llbracket A \rrbracket_\emptyset \rightarrow A$.

C Type checking

Instant previews give analysts quick feedback when they write incorrect code, but having type information is still valuable. First, it can help give better error messages. Second, types can be used to provide auto-complete – when the user types ‘.’ we can offer available members without having to wait until the value of the object is available.

Revised dependency graph. Type checking of small programs is typically fast enough that no caching is necessary. However, The Gamma supports *type providers* [9, 54], which can generate types based on an external file or a REST service call, e.g. [47]. For this reason, type checking can be relatively time consuming and can benefit from the same caching facilities as those available for instant previews.

Adding type checking requires revising the way we construct the dependency graph introduced in section 4. Previously, a variable bound by a lambda function had no dependencies. However, the type of the variable depends on the context in which it appears. Given an expression $o.m(\lambda x \rightarrow x)$, we infer the type of x from the type of the first argument of the member m . A variable node for x thus needs to depend on the call site of m . We capture that by adding an edge `callsite(m, i)` from x to o which indicates that x is the input variable of a function passes as the i^{th} argument to the m member of the expression represented by the target node. We also add `callsite(m, i)`

$$\begin{array}{c}
\text{(var)} \frac{(\text{var}(x, s), v, \text{callsite}(m, i)) \in E \quad v \vdash \{.., m : (\tau_1, \dots, \tau_k) \rightarrow \tau, ..\} \quad \tau_i = \tau' \rightarrow \tau''}{\text{var}(x, s) \vdash \tau'} \\
\text{(mem)} \frac{\forall i \in \{0 \dots k\}. (\text{mem}(m, s), v_i, \text{arg}(i)) \in E \quad v_0 \vdash \{.., m : (\tau_1, \dots, \tau_k) \rightarrow \tau, ..\} \quad v_i \vdash \tau_i}{\text{mem}(m, s) \vdash \tau} \\
\text{(fun)} \frac{\{(\text{fun}(x, s), v_b, \text{body}), (\text{var}(x, s), v_c, \text{callsite}(m, i))\} \subseteq E \quad v_c \vdash \{.., m : (\tau_1, \dots, \tau_k) \rightarrow \tau, ..\} \quad \tau_i = \tau' \rightarrow \tau'' \quad v_b \vdash \tau''}{\text{fun}(x, s) \vdash \tau' \rightarrow \tau''}
\end{array}$$

■ **Figure 15** Rules that define type checking of terms and expressions over a dependency graph (V, E)

as an edge from the node of the function. Figure 14 shows the revised dependency graph for $o.m(\lambda x \rightarrow x)$.

Type checking. The structure of typing rules is similar to the evaluation relation $v \Downarrow d$ defined earlier. Given a dependency graph (V, E) , we define typing judgements in the form $v \vdash \tau$. The type τ can be a primitive type, a function $\tau \rightarrow \tau$ or an object type $\{m_1 : \sigma_1, \dots, m_n : \sigma_n\}$ with member types $\sigma = (\tau_1, \dots, \tau_n) \rightarrow \tau$.

The typing rules for variables, functions and member access are shown in figure 15. When type checking a member access (mem), we find its dependencies v_i and check that the instance is an object with the required member m . The types of arguments of the member then need to match the types of the remaining (non-instance) nodes. Type checking a function (fun) and a variable (var) is similar. In both cases, we follow the `callsite` edge to find the member that accepts the function as an argument and use the type of the argument to check the type of the function or infer the type of the variable.

The results of type checking can be cached and reused in the same way as instant previews, although we leave out the details. A property akin to correctness (theorem 6) requires defining standard type checking over the structure of expressions, which we also omit for space reasons.

D Feedback-friendly abstraction

The data analysis by Financial Times in section 2.1 illustrates why notebook users often avoid abstraction. Wrapping code in a function makes it impossible to split code into cells and see results of intermediate steps. Instead, the analysis used a global variable with possible values in a comment.

Providing instant previews inside ordinary functions is problematic, because we do not have readily available values for input parameters and our mechanism for lambda functions only provides delayed previews inside body of a function. We believe that

Foundations of a live data exploration environment

extending the data exploration calculus with an abstraction mechanism that would support development with instant feedback is an interesting design problem and we briefly outline a possible solution here.

Data scientists often write code interactively using a sample data set and, when it works well, wrap it into a function that they then call on other data. Similarly, spreadsheet users often write equation in the first row of a table and then use the “drag down” operation to apply it to other rows. One way of adding similar functionality to the data exploration calculus is to label a sequence of commands such that the sequence can be reused later with different inputs:

$$\begin{aligned} p &= c_1; \dots; c_n \\ c &= \text{let } x = t \mid t \mid \text{lbl}: p \mid \text{lbl} \end{aligned}$$

We introduce two new kinds of commands: a labelled sequence of commands and a reference to a label. When evaluating, the command *lbl* is replaced with the associated sequence of commands *p* before any other reductions. Consequently, variables used in the labelled block are dynamically scoped and we can use let binding to redefine a value of a variable before invoking the block repeatedly. The correct use of dynamic scoping can be checked using *coeffacts* [48].

This minimalistic abstraction mechanism supports code reuse without affecting how instant previews are computed. Commands in a labelled block require variables to be defined before the block. Those define sample data for development and can be redefined before reusing the block. We intend to implement this mechanism in a future version of our data exploration environment (section 6.4).

Chapter 9

Wrattler: Reproducible, live and polyglot notebooks

Tomas Petricek, James Geddes, and Charles Sutton. 2018. Wrattler: Reproducible, live and polyglot notebooks. In *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*, Melanie Herschel (Ed.). USENIX Association.

Wrattler: Reproducible, live and polyglot notebooks

Tomas Petricek
University of Kent
The Alan Turing Institute
tomas@tomas.net

James Geddes
The Alan Turing Institute
jgeddes@turing.ac.uk

Charles Sutton
The University of Edinburgh
The Alan Turing Institute and Google
csutton@inf.ed.ac.uk

Abstract

Notebooks such as Jupyter became a popular environment for data science, because they support interactive data exploration and provide a convenient way of interleaving code, comments and visualizations. Alas, most notebook systems use an architecture that leads to a limited model of interaction and makes reproducibility and versioning difficult.

In this paper, we present Wrattler, a new notebook system built around provenance that addresses the above issues. Wrattler separates state management from script evaluation and controls the evaluation using a dependency graph maintained in the web browser. This allows richer forms of interactivity, an efficient evaluation through caching, guarantees reproducibility and makes it possible to support versioning.

1 Introduction

Notebooks [5, 15] are literate programming [6] systems that allow interleaving text, code and outputs. To aid reproducible, exploratory data science, notebook systems should provide:

Richer interaction model. Web browsers are increasingly powerful and allow moving parts of data exploration to the client-side. Notebooks should leverage this and give live previews when writing code to perform simple data exploration.

Transparent state management. The state maintained by a notebook should be transparent and accessible to external tools. This would allow versioning of state and development of tools that provide hints based on the notebook state.

Multiple languages and tools. A notebook should make it easy to combine multiple programming languages. A cell written in one language should be able to automatically access data frames defined in other languages.

Improved reproducibility. Changing code in a cell should invalidate results that depend on data frames defined in the cell. Reverting a change should immediately revert the result to the previous one and show it immediately using a cache.

Supporting these is a challenge that combines several research areas. We need programming language techniques to efficiently update live previews during editing, provenance

methods to track data dependencies and data representation that can be shared across languages.

Wrattler is a new notebook system that supports the above features. We follow a line of work combining provenance tracking with notebooks [7, 14], but rather than extending existing systems, we revisit two core aspects of the notebook architecture (Section 2). First, Wrattler uses a *dependency graph* to track provenance between cells and even function calls inside individual cells (Section 3.1). When a cell is changed, relevant parts of a graph are invalidated. This guarantees reproducibility and enables more efficient re-computation. Second, Wrattler introduces a *data store* that separates state management from code execution (Section 3.2). The data store handles versioning and simplifies the support for polyglot programming.

Together, these two changes to the standard architecture of notebook systems make Wrattler notebooks (Section 4) polyglot, reproducible (with an easy and reliable state rollback) and live (with efficient re-computation on change that enables live preview during data exploration).

2 Wrattler architecture

Standard notebook architecture consists of a *notebook* and a *kernel*. The kernel runs on a server, evaluates code snippets and maintains state they use. Notebook runs in a browser and sends commands to the kernel in order to evaluate cells selected by the user. As illustrated in Figure 1, Wrattler splits the server functionality between two components:

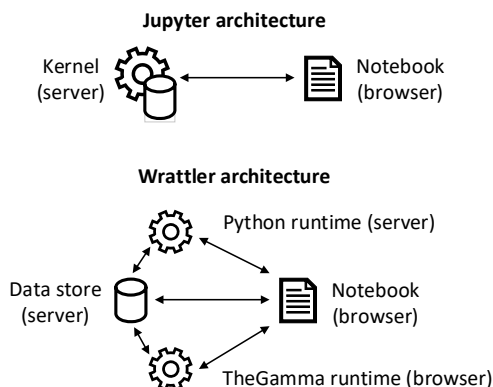


Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Language runtimes can run on the server-side (e.g. Python) or client-side (e.g. TheGamma).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

TaPP 2018, July 11–12, 2018, London, UK

Copyright remains with the owner/author(s).

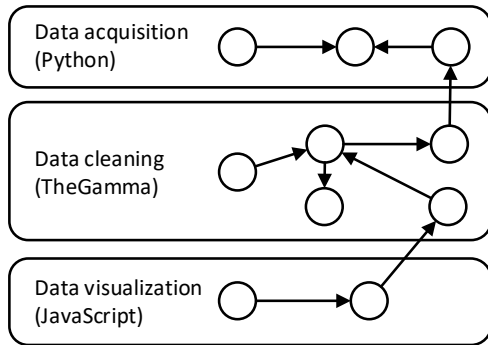


Figure 2. Dependency graph of a sample notebook: The first (Python) cell downloads data and exports the result as a data frame; the second (TheGamma) cell performs data cleaning and the third (JavaScript) cell creates a visualization. Language runtime for TheGamma runs in the browser and creates a fine-grained graph (which allows an efficient live previews), while Python and JavaScript runtimes create just one node for the whole source code.

Data store. Imported external data and results of running scripts are stored in the data store. The data store keeps version history and annotates data with metadata such as types, inferred semantics and provenance information.

Language runtimes. Code in notebook cells is evaluated by language runtimes. The runtimes read input data from and write results back to the data store. Wrattler supports language runtimes that run code on the server (similar to Jupyter), but also browser-based language runtimes.

Notebook. The notebook is displayed in a web browser and orchestrates all other components. The browser builds a dependency graph between cells or individual calls. It invokes language runtimes to evaluate code that has changed, and reads data from the data store to display results.

3 Wrattler components

Wrattler runs in the web browser and communicates with the data store and language runtimes that may run on the server or in the browser. An example of browser-based language runtime is TheGamma [12], discussed in Section 3.3.

3.1 Dependency graph

At runtime, Wrattler maintains a dependency graph that is composed from sub-graphs created by individual language runtimes. The graph is acyclic and a node can only depend on earlier nodes. Each node has a value which may be:

- A language-specific value that Wrattler does not understand. This is kept in the browser and re-computed when the notebook is re-opened.
- A data frame. Data frame is stored in the data store and browser keeps a reference (URL) of the frame. This is understood by all language runtimes and provides a way of exchanging data between multiple languages.

Figure 2 shows a sample dependency graph. Wrattler creates two nodes for each cell (representing the cell and its source code) and a node for each data frame exported by a cell (e.g. the rightmost node in the first cell). Nodes in subsequent cells may depend on data frames exported by earlier cells.

Wrattler treats data frames in a special way. They are stored in data store and each language runtime is responsible for loading them into a native language representation (e.g. pandas in Python and array of records in JavaScript).

Dependency graph construction. The dependency graph is updated after every code change. Wrattler invokes individual language runtimes to parse each cell. Language runtimes that run in the browser (e.g. TheGamma) produce a fine-grained syntax tree. The result of parsing the whole notebook is then a list of elements obtained for each cell.

Wrattler then walks over the syntax tree and binds a dependency graph node to each syntactic element using a process described in Figure 3. The *antecedents* of a node are the nodes that it depends on. This typically includes inputs for an operation or instance on which a member access is performed.

Checking and evaluation. Nodes in the dependency graph can be annotated with information such as the evaluated value of the syntactic element that the node represents. An important property of the binding process (Figure 3) is that, if there is no change in antecedents of a node, binding will return the same node as before. As a result, previously evaluated values attached to nodes in the graph are reused.

Wrattler re-evaluates parts of the dependency graph on demand and the displayed results and visualizations always reflect the current source code in the notebook. When the evaluation of a cell is requested, Wrattler recursively evaluates all the antecedents of the node and then evaluates the value of the node. The evaluation is delegated to a language runtime associated with the language of the node:

1. For Python nodes, the language runtime sends the source code, together with its dependencies, to a server that retrieves the dependencies and evaluates the code.
2. For TheGamma and JavaScript nodes, the language runtime collects values of the dependencies and runs the operation that the node represents in the browser.

3.2 Data store

The data store enables communication between individual Wrattler components and provides a way for persistently storing input data. Data frames stored in the data store are associated with the hash produced by the binding process outlined in Figure 3 and are immutable. When the notebook changes, new nodes with new hashes are created and appended to the data store. This means that language runtimes can cache them and avoid fetching data from data store each time they need to evaluate a code snippet.

```

procedure bind(cache, syn) =
  let h = hash({kind(syn)} ∪ antecedents(syn))
  if not contains(cache, h) then
    let n = fresh node
    value(n), hash(n) ← Unevaluated, h
    set(cache, h, n)
  lookup(cache, h)

```

Figure 3. When binding a graph node to a syntactic element, Wrattler first computes a set of hashes that uniquely represent the node. This includes hash of the kind of the node (e.g. the source code of a Python node or member name in TheGamma) and hashes of all antecedents. If a node with a given hash does not exist in cache, a new node is created. We set its hash, indicate that its value has not been evaluated and add it to the cache.

External inputs imported into Wrattler notebooks (such as downloaded web pages) are stored as binary blobs. Data frames are stored in JSON format (as an array of records), but we intend to use a suitable database in the future. During the binding process (Section 3.1), a language runtime identifies imported and exported data frames for each cell (e.g. by static analysis of the code). Those are then represented as hashes (keys) referring to a location in the data store.

The data store also supports a mechanism for annotating data frames with semantic information. Columns can be annotated with primitive data types (date, floating-point number) and semantic annotation indicating their meaning (address or longitude and latitude). Columns, rows and individual cells of the data frame can also be annotated with custom metadata such as their data source or accuracy.

In addition to storing the raw data, the data store also persistently stores the current and multiple past versions of the dependency graph constructed from the notebook (saved by an explicit checkpoint). This makes it possible to analyse the history of a notebook and track how data is transformed by the computation in a notebook.

3.3 TheGamma script

The Wrattler architecture supports languages that can be parsed and evaluated in the browser. To illustrate this, we integrated Wrattler with TheGamma [12], a simple browser-based language for data exploration.

The Figure 4 shows TheGamma cell in Wrattler during editing. The example uses broadband speed data published by the UK government [10] and calculates average download speed in urban and rural areas, respectively. TheGamma supports a rich interactive model in two ways:

- The script is evaluated on-the-fly during editing and a live preview is shown (below the code editor).
- All code can be written using autocomplete that offers available members (representing aggregation operations). Rather than writing code, user repeatedly selects one of the offered members (which are provided by a type provider [18] running in the browser).

For the purpose of this paper, the most important aspect of TheGamma is that scripts can be parsed and evaluated in the browser. This allows more interactive style of data exploration without round-trips to re-evaluate modified code.

4 Properties of Wrattler

The Wrattler architecture outlined in Section 2 allowed us to develop a prototype system with a number of properties that are difficult to obtain with traditional notebooks.

4.1 Reproducible, live and smart

The two most important aspects of the Wrattler architecture are that it separates the state from the language runtime (using a data store) and that it keeps a dependency graph based on the current notebook source code (on the client). The provenance information that is available thanks to this architecture enable a number of properties.

Reproducibility. The evaluation outputs displayed in Wrattler notebook always reflect the current source code. When code changes, Wrattler updates the dependency graph and hides invalidated visualizations. Because the data store caches earlier results, it is always possible to go back without re-evaluating the whole notebook.

Refactoring. The dependency graph allows us to implement notebook refactoring. For example, it is possible to extract only code necessary to produce a given visualization. For code written in TheGamma, this extracts individual operations; for Python or JavaScript, we can currently extract code at cell-level granularity.

Live previews. The dependency graph makes it possible to give live previews during development. When code changes, only values for new nodes in the graph need to be calculated. The fine-grained structure of the dependency graph for TheGamma makes it possible to update previews instantly.

Polyglot. Sharing the state via data store makes it possible to combine multiple language runtimes, as long as they support sharing data via data frames. In our prototype, this includes R, JavaScript and TheGamma script, but the extensibility model allows adding further languages.

4.2 Wrattler prototype

A prototype implementation of the Wrattler system is available on GitHub (<http://github.com/wrattler>). The prototype implements language runtimes for TheGamma script (Section 3.3), R and JavaScript. It builds a dependency graph (Section 3.1) and uses it to evaluate results of cells.

The data store (Section 3.2) stores data in Microsoft Azure in JSON format. Support for meta-data annotations and big data is not yet implemented. Storing notebook state in the data store also allowed us to develop an integration with Data diff [17], which provides data cleaning recommendations.

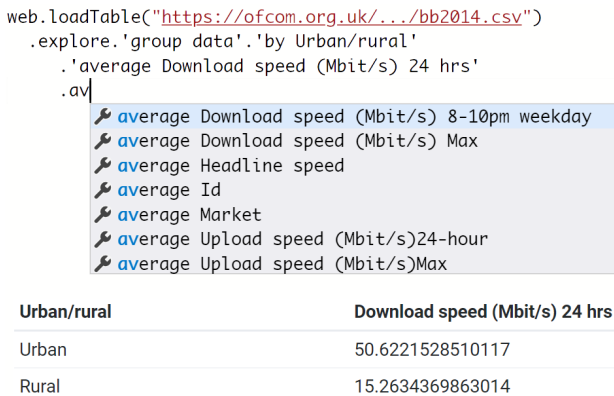


Figure 4. TheGamma script that downloads and aggregates UK government data, running in Wrattler notebook with a live preview.

5 Related work

The work in this paper directly follows the work on IPython and Jupyter systems [5, 15]. Wrattler shares many properties with those and aims to address some of their limitations. To address reproducibility, some Jupyter extensions and systems such as R markdown lock cells after evaluation.

Dataflow notebooks [7] attach unique hashes to cell evaluations. This allows the user to refer to dependencies explicitly and, in effect, construct a dependency graph manually. Scientific workflow systems [1, 11] manage evaluation over a dependency graph similarly to Wrattler, but allow editing it directly via a GUI, rather than through code in a notebook.

The noWorkflow project [14] links the two approaches by instrumenting Jupyter kernel with a mechanism for capturing provenance based on light-weight annotations. Vizer [4] focuses on integrating notebooks with spreadsheet-like interface. It internally uses a data store component similar to ours, but does not keep dependency graph on the client.

Our binding process is inspired by Roslyn [9] and extends an earlier work on TheGamma [13]. It is similar to methods used in live programming languages [3, 8], incremental compilation [16] and partial evaluation [2]. Wrattler adapts those methods to a notebook environment.

6 Summary

This paper presents early work on Wrattler – a new notebook system for data science that makes notebooks reproducible, live and polyglot. The properties of Wrattler are enabled by provenance information that is maintained thanks to two changes to the standard architecture of notebook systems.

First, Wrattler separates the state management from code execution. This allows versioning, polyglot notebooks and integration of third-party tools that can work directly with the data store. Second, Wrattler keeps a dependency graph on the client (web browser) and uses it to control evaluation. This guarantees reproducibility and allows faster feedback during development.

Acknowledgments

The authors would like to acknowledge the funding provided by the UK Government’s Defence & Security Programme in support of the Alan Turing Institute and the EPSRC grant EP/N510129/1. We thank to our colleagues Chris Williams, Zoubin Ghahramani and Ian Horrocks and attendees of a recent AIDA project workshop.

References

- [1] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management*. IEEE, 423–424.
- [2] Olivier Danvy. 1999. Type-directed partial evaluation. In *Partial Evaluation*. Springer, 367–411.
- [3] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. *ACM SIGPLAN Notices* 40, 10 (2005), 505–518.
- [4] Juliana Freire, Boris Glavic, Oliver Kennedy, and Heiko Mueller. 2016. The Exception That Improves the Rule. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. ACM, 7:1–7:6.
- [5] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks—a format for reproducible computational workflows.. In *ELPUB*. 87–90.
- [6] Donald Ervin Knuth. 1992. *Literate programming*. Center for the Study of Language and Information Stanford.
- [7] David Koop and Jay Patel. 2017. Dataflow Notebooks: Encoding and Tracking Dependencies of Cells. In *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*. USENIX Association.
- [8] Sean McDirmid. 2007. Living it up with a live programming language. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 623–638.
- [9] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. 2011. The Roslyn Project, Exposing the C# and VB compiler’s code analysis. *White paper, Microsoft* (2011).
- [10] Ofcom. 2018. Open data. Available online at <https://www.ofcom.org.uk/research-and-data/data/opendata>. (2018).
- [11] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. 2004. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20, 17 (2004), 3045–3054.
- [12] Tomas Petricek. 2017. Data exploration through dot-driven development. In *Proceedings of ECOOP*, Vol. 74. Schloss Dagstuhl.
- [13] Tomas Petricek. 2018. Design and implementation of a live coding environment for data science. Unpublished draft. Available online at <http://tomasp.net/academic/drafts/live>. (2018).
- [14] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. 2015. Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow. In *Workshop on the Theory and Practice of Provenance (TaPP)*. 155–167.
- [15] M Ragan-Kelley, F Perez, B Granger, T Kluyver, P Ivanov, J Frederic, and M Bussonnier. 2014. The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication.. In *AGU Fall Meeting Abstracts*.
- [16] Mayer D Schwartz, Norman M Delisle, and V S Begwani. 1984. Incremental compilation in Magpie. *SIGPLAN Not.* 19, 6 (1984), 122–131.
- [17] Charles Sutton, Tim Hobson, James Geddes, and Rich Caruana. 2018. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. *Proceedings of KDD*. (2018).
- [18] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. 2013. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of DDFP*. ACM, 1–4.

Part IV

Publications: Iterative prompting

Chapter 10

The Gamma: Programmatic data exploration for non-programmers

Tomas Petricek. 2022. The Gamma: Programmatic Data Exploration for Non-programmers. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2022, Rome, Italy, September 12-16, 2022*, Paolo Bottoni, Gennaro Costagliola, Michelle Brachman, and Mark Minas (Eds.). IEEE, 1–7. <https://doi.org/10.1109/VL/HCC53370.2022.9833134>

The Gamma: Programmatic Data Exploration for Non-programmers

Tomas Petricek

University of Kent, UK and Charles University, Czech Republic

tomas@tomasp.net

Abstract—Data exploration tools based on code can access any data source, result in reproducible scripts and encourage users to verify, reuse and modify existing code. Unfortunately, they are hard to use and require expert coding skills. Can we make data exploration tools based on code accessible to non-experts?

We present The Gamma, a novel text-based data exploration environment that answers the question in the affirmative. The Gamma takes the idea of code completion to the limit. Users create transparent and reproducible scripts without writing code, by repeatedly choosing from offered code completions.

The Gamma is motivated by the needs of data journalists and shows that we may not need to shy away from code for building accessible, reproducible and transparent tools that allow a broad public to benefit from the rise of open data.

Index Terms—data exploration, data journalism

I. INTRODUCTION

Despite the advances on visual tooling, programmatic data exploration remains the choice of expert analysts. It is flexible, offers greater reusability and leads to transparent analyses. The design of a programmatic data exploration tool that would be accessible to data journalists poses a number of design challenges. First, the tool needs to have a low barrier to entry to support first-time users without training. Second, it needs to support multiple data sources in a uniform way to allow transfer of knowledge across domains. Finally, users need to be able to learn by looking at existing data analyses.

We present The Gamma, a text-based data exploration tool for non-experts that is based on a single, easy to understand interaction principle. It provides a uniform access to data tables, graph databases and data cubes and leads to transparent analyses that can be easily reproduced, encouraging learning and critical engagement with data.

The Gamma is based on *iterative prompting*, which turns code completion from a programmer assistance tool into a non-expert programming mechanism that allows users to construct all valid data exploration scripts just by repeatedly choosing an item from a list of offered options. The design favors *recognition over recall* and allows non-programmers to construct entire scripts without first learning to code. Yet, the result remains a transparent and reproducible script. A crucial feature is that iterative prompting only offers operations that are valid in a given context and that it offers all such operations; it is both correct and complete.

The Gamma focuses on tasks that a data journalist may want to complete (Figure 1). The user accesses data available in a

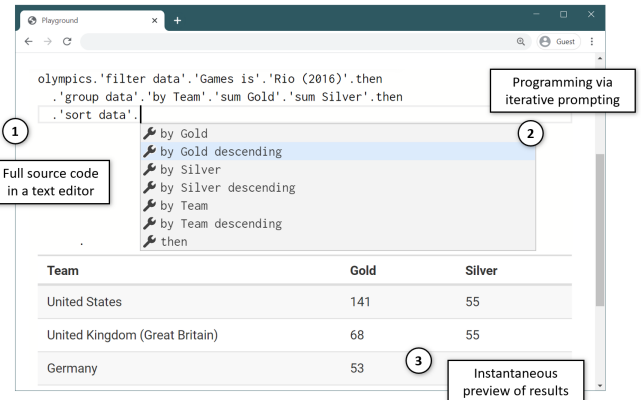


Fig. 1: Obtaining teams with the greatest number of gold medals from Rio 2016 Olympics: (1) Reproducible The Gamma script; (2) contextual iterative prompting offering ways of sorting the data; (3) an instant preview of results.

structured format. They make several experiments to find an interesting insight, e.g. by applying different aggregations or filters. They visualize the results using a table or a chart before publishing their analysis. The Gamma makes such programmatic data exploration simple enough for non-programmers. Scraping and cleaning of messy data or building custom data visualizations is outside of the scope of our work, but exposing such functionality using iterative prompting is an interesting and worthwhile future challenge.

In this paper, we describe and evaluate the design principles behind The Gamma project:

- We introduce the iterative prompting principle in The Gamma (Section III) and show how it can be used for querying of distinct data sources including data tables, graph databases and data cubes (Section IV).
- We illustrate the expressiveness of the system through a case study (Section V) and evaluate it through a user study (Section VI), confirming that non-programmers can use The Gamma to construct non-trivial data queries.
- We reflect how our design lowers barriers to entry, supports learning without experts and offers a complete and correct program construction method (Section VII).

The Gamma is available at <http://thegamma.net>, both as an open-source library and a hosted data exploration service.

II. RELATED WORK

We aim to make recent advances on information-rich programming [1] available to non-programmers [2], [3], in the context of data journalism [4]. Our work features a novel combination of characteristics in that our iterative prompting interaction principle is centered around code, but reduces the conceptual complexity of coding to a single basic kind of interaction.

Code Completion for Data Science: The Gamma utilizes type providers [1], [5], which integrate external data into a static type system. This enables auto-completion [6], which we turn into a tool for non-programmers. Similar systems based on machine learning and domain specific languages [7], [8] do not guarantee completeness, i.e. it is unclear whether the user can create all possible scripts. Approaches based on natural language are effective [9], [10], [11], but hide the underlying structure and do not help users understand the exact operations performed. Code completion based on machine learning [12], [13] also exists for general-purpose programming languages used by data scientists such as Python [14], but this focuses on providing assistance to expert programmers.

Notebooks and Business Intelligence Tools: Notebooks such as Jupyter [15] are widely used data exploration environments for programmers. The Gamma targets non-experts, but could be integrated with a multi-language notebook system [16]. Spreadsheets, business intelligence [17], [18] and other visual data analytics tools [19], [20] do not involve programming, but require mastering a complex GUI. In contrast, in The Gamma, all operations can be completed through a single kind of interaction. Several systems [21], [22], [23] record interactions with the GUI as a script that can be modified by the user. Unlike in The Gamma, the generated code does not guide the user in learning how to use the system.

Easier Programming Tools: Many systems aim to make programming easier. Victor [24] inspired work on live environments environments [25], [26], [27] that help programmers understand how code relates to output; exploratory systems [28], [29] assist with completing open-ended tasks; and system combining code with visualization also exists for graph querying [30]. The Gamma is live in that our editor gives an instant preview of the results. To avoid difficulties with editing code as text, some systems use structured editors [31], [32], [33], [34], [35]. Many systems simplify programming by offering high-level abstractions, e.g. for interactive news articles [36], statistical analyses [37], data visualization [38], [39]. The Gamma provides high-level abstractions for data querying, but supporting other tasks remains future work.


Programming without Writing Code: In programming by example [40], used for example in spreadsheets [41], [42], the user gives examples of desired results. In direct manipulation [43], a program is specified by interacting with the output. This has been used in the visual domain [44], but also for data querying [45], [46], [47]. Direct manipulation can also support data exploration by letting users partially edit queries, e.g. by changing quantifiers as in DataPlay [48].

III. OVERVIEW

The Gamma is a text-based system that allows non-experts to explore data using iterative prompting – by repeatedly selecting an item from an auto-complete list. The study presented in Section VI confirms that the kind of data exploration shown in the next section can be successfully done by non-experts.

We introduce The Gamma by walking through a typical data exploration task. A data journalist from Kent is exploring travel expense claims by members of the House of Lords published by the UK government [49]. After importing the CSV file through a web interface, the environment is initialized with code that refers to the imported data as `expenses` using the type provider for tabular data (Section IV). The journalist types `.` (dot) to start exploring the data:


```
1 expenses.
```



```
✎ drop columns  
✎ filter data  
✎ group data  
✎ paging  
✎ sort data
```

The type provider offers a list of operations that the journalist can perform. To find House of Lords members from Kent, the journalist chooses `filter data`. She is then offered a list of columns based on the schema of the CSV file and chooses `County is`. The completion lists counties in the data set:

```
1 expenses.'filter data'.'County is'.
```



```
✎ Hull  
✎ Invernesshire  
✎ Kent  
✎ Kincardineshire
```

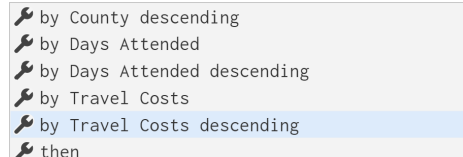
The journalist chooses `Kent`. The Gamma evaluates the code on-the-fly and shows a preview of results.

```
1 expenses.'filter data'.'County is'.Kent
```

Name	County	Days Attended	Travel Costs
Lord Astor of Hever	Kent	7	85
Lord Freud	Kent	1	0
Lord Harris of Peckham	Kent	3	0

The journalist decides to compare travel costs. She finishes specifying the filtering condition by choosing `then` and is offered the same list of querying operations as in the first step. She selects `sort data` and is offered a list of sorting options:

```
1 expenses.'filter data'.'County is'.Kent.then.  
2 'sort data'.
```



```
✎ by County descending  
✎ by Days Attended  
✎ by Days Attended descending  
✎ by Travel Costs  
✎ by Travel Costs descending  
✎ then
```

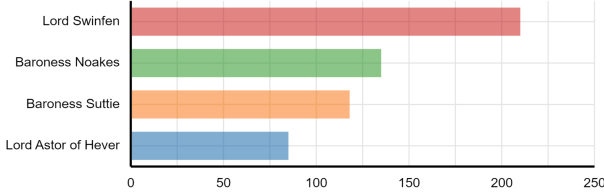
The journalist chooses `then` and is, again, offered the list of querying operation. She uses `paging` to get the top 4 records, which requires typing `4` as the argument. She then uses the

get series operation to obtain a data series associating travel expenses with a name, which is automatically visualized:

```

1 expenses.'filter data'.'County is'.Kent.then
2   .'sort data'.'by Travel Costs descending'.then
3   .paging.take(4).'get series'
4   .'with key Name'.'and value Travel Costs'

```



The code is not unlike an SQL query, except that the whole script is constructed using iterative prompting, by repeatedly selecting one of the offered members. Those represent both operations, such as sort by and arguments, such as Kent. The only exception is when the analyst needs to type the number 4 to specify the number of items to take.

IV. SYSTEM DESCRIPTION

A program in The Gamma is a sequence of commands that can be either a variable declarations or an expression that evaluates to a value. An expression is a reference to a data source followed by a chain of member accesses. Each expression has a type that is used to generate options in auto-completion. A type defines a list of members that, in turn, have their own types. The types are not built-in, but are generated by type providers for individual data sources. The syntax and semantics of the language has been described elsewhere [50].

A new data source can be supported by implementing a *type provider*, which defines a domain specific language for exploring data of a particular kind. A type provider generates object types with members (such as paging or Kent) that are accessed via iterative prompting. We outline type providers for exploring data cubes (inspired by Syme et al. [1]), tabular data (formalized elsewhere [52]), and graph databases.

Data Cube Provider: Data cubes are multi-dimensional arrays of values. For example, the World Bank collects a range of indicators about many countries each year while the UK government expenditure records spending for different government services, over time, with different adjustments:

```

1 worldbank.byCountry.'United States'.
2   'Climate Change'.'CO2 emissions (kt)'
3
4 expenditure.byService.Defence.inTermsOf.GDP

```

The dimensions of the worldbank cube are countries, years and indicators, whereas the dimensions of expenditure are government services, years and value type (adjusted, nominal, per GDP). Figure 2a illustrates how the provider allows users to slice the data cube. Choosing byCountry.'United States', restricts the cube to a plane and 'CO2 emissions (kt)' then gives a series with years as keys and emission data as values. Similarly, we could first filter the data by a year or an indicator. The same mechanism is used to select UK government spending on defence in terms of GDP.

Graph Database Type Provider: Graph databases store nodes representing entities and relationships between them. The following example explores a database of Doctor Who characters and episodes. It retrieves all enemies of the Doctor that appear in the Day of the Moon episode:

```

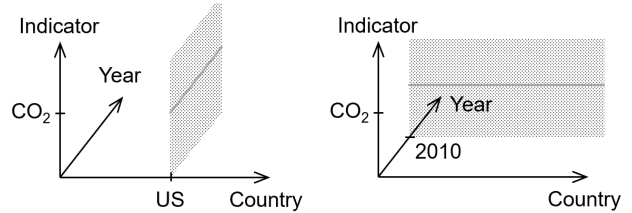
1 drwho.Character.Doctor.'ENEMY OF'.'[any]'
2   .'APPEARED IN'.'Day of the Moon'

```

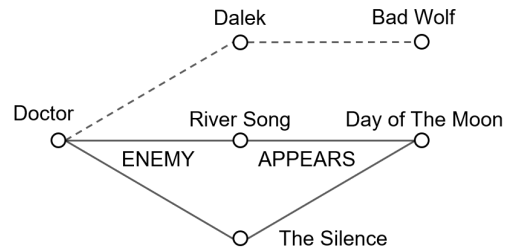
We start from the Doctor node and then follow two relationships. We use 'ENEMY OF'.'[any]' to follow links to all enemies of the Doctor and then specify 'APPEARED IN' to select only enemies that appear in a specific episode. The query is illustrated in in Figure 2b. The members are generated from the data; ENEMY OF and APPEARED IN are labels of relations and Doctor and Day of the Moon are labels of nodes. The [any] member defines a placeholder that can be filled with any node with the specified relationships. The results returned by the provider is a table of properties of all nodes along the specified path, which can be further queried and visualized.

Tabular Data Provider: Unlike the graph and data cube providers, the type provider for tabular data does not just allow selecting a subset of the data, but it can be used to construct SQL-like query. Consider the example of querying expense claims from Section III, which filters and then sorts the data.

When using the provider, the user specifies a sequence of operations. Members such as 'filter data' or 'sort data' determine the operation type. Those are followed by members that specify operation parameters. For example, when filtering data, we first select the column and then choose a desired value. Unlike SQL, the provider only allows users to choose from pre-defined filtering conditions, but this is sufficient for constructing a range of practical queries.



(a) Exploring World Bank data using the data cube type provider, users choose values from two dimensions to obtain a data series.



(b) To query graph data, the user specifies a path through the data, possibly with placeholders to select multiple nodes.

Fig. 2: Type providers for exploring cube and graph data.

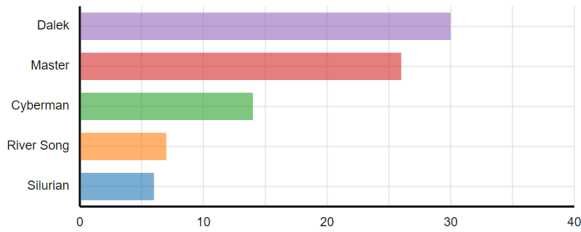


Fig. 3: Who does the Dr Who fight most frequently?

V. CASE STUDY

The Gamma aims to simplify programmatic data exploration while keeping enough expressive power to allow users to create interesting data explorations. To show what can be achieved by interactive prompting, we present a case study that explores a graph database with Dr Who series data.¹

The following constructs a chart (Figure 3) of top Dr Who villains by the number of episodes in which they appear. This case is interesting as it combines the graph database provider for fetching the data with the tabular data provider:

```

1 drWho.Character.Doctor.'ENEMY OF'. '[any]'
2   .'APPEARED IN'. '[any]'. explore
3   .'group data'. 'by Character name'
4   .'count distinct Episode name'. then
5   .'sort data'. 'by Episode name descending'. then
6   .paging.take(8). 'get series'
7   .'with key Character name'
8   .'and value Episode name'

```

Line 1 use the graph provider to find all paths linking the Doctor with any character linked via ENEMY OF, followed by any episode linked by APPEARED IN. This produces a table that can be analysed using the tabular data provider by selecting explore. For each character (the villain) we count the number of distinct episodes. The result is shown in Figure 3. Despite performing a sophisticated data analysis that involves a graph database query, followed by an SQL-like data aggregation, the code can be constructed using iterative prompting, with the exception of the numbers in paging.

VI. USER STUDY

Data exploration environments are complex systems that do not yield to simple controlled experimentation [54]. Rather than comparing our work with other tools, we evaluate whether The Gamma can be successfully used by non-programmers.

We performed a between-subjects study to assess whether non-programmers are able to complete a simple data exploration task using The Gamma. We recruited 13 participants (5 male, 8 female) from a business team of a research institute working in non-technical roles (project management, communications). Only one participant (#12) had prior programming experience. We split participants into 4 groups and asked each group to complete a different task. We gave participants a brief overview of The Gamma. The participants then worked for 30

minutes, after which we conducted a semi-structured group interview. We offered guidance if participants were unable to progress for more than 5 minutes. The four tasks were:

- *Expenditure*. Participants were shown the *worldbank* data cube and were asked to compare UK spending on ‘Public order and safety’ and ‘Defence’ using another data cube.
- *Lords*. Participants were shown *worldbank* and were asked to use the *expenses* data table provider to sort London House of Lords members by their travel costs.
- *Worldbank*. Participants were given a minimal iterative prompting demo and a code sample using *worldbank*. They were asked to solve another *worldbank* task.
- *Olympics*. Participants were given a demo using *olympics* that did not involve grouping. They were asked to solve a problem involving grouping and aggregation.

Our primary hypothesis was that non-programmers will be able to use The Gamma to explore data. This was tested by all four tasks for one of the supported data sources.

The tasks *expenditure* and *lords* further test if knowledge can be transferred between different data sources by using one sources in the introduction and another in the task; *worldbank* explores whether users can learn how to use a data source from just code samples; and *lords* lets us study to what extent participants form a correct mental model of the more complex query language used in the tabular data source.

Can non-programmers explore data with The Gamma? All participants were able to complete, at least partially, a non-trivial data exploration task and only half of them required further guidance. Participants spent 10–25 minutes (average 17) working with The Gamma and 12 out of 13 completed the task; 6 required assistance, but 3 of those faced the same issue related to operations taking arguments (discussed later).

A number of participants shared positive comments in the group interviews. Participant #3 noted that “*this is actually pretty simple to use,*” participant #2 said that The Gamma alleviated their unease about code: “*for somebody who does not do coding or programming, this does not feel that daunting.*” and participant #5 suggested that the system could be used as an educational tool for teaching critical thinking with data.

How users learn The Gamma? There is some evidence that knowledge can be transferred between different data sources. In *expenditure* and *lords*, participants were able to complete tasks after seeing a demo using another data source. Participant #2 “*found it quite easy to translate what you showed us in the demo to the new dataset.*”. However, the *lords* task has been more challenging as it involves a more complex data source.

There is also some evidence that, once a user understands iterative prompting, they can learn from just code samples. All three participants were able to complete the *worldbank* task, where they were given printed code samples, but no demo using any data source. When discussing suitable educational materials for The Gamma, participant #7 also confirmed that “*a video would just be this [i.e. a code sample] anyway.*”

¹See: <http://gallery.thegamma.net/87/>. We also used The Gamma for projects exploring the UK government expenditure, activities of a research institute and Olympic medal winners, available at <http://turing.thegamma.net> and <http://rio2016.thegamma.net>

How users understand complex query languages? The tabular type provider uses a member then to complete the specification of a current operation, for example when specifying a list of aggregation operations. Two participants (#12 and #13) initially thought that then is used to split a command over multiple lines, but rejected the idea after experimenting. Participant #12 then correctly concluded that it “allows us to chain together the operations” of the query. While iterative prompting allows users to start exploring new data sources, the structures exposed by more complex data sources have their own further design principles that the users need to understand.

What would make The Gamma easier to use? Three participants (#11, #12, #13) struggled to complete a task using the tabular data source, because they attempted to use operation that takes a numerical parameter and thus violates the iterative prompting principle. This could be avoided by removing such operations or by hiding them under an “advanced” tab.

The Gamma uses an ordinary text editor and most participants had no difficulty navigating around code, making edits or deleting fragments, which is harder in a structure editor. Some participants used the text editor effectively, e.g. leveraging copy-and-paste. However, two participants struggled with indentation and a syntax error in an unrelated command. This could likely be alleviated through better error reporting.

VII. DISCUSSION

As a text-based programming environment for non-programmers, The Gamma examines an unexplored point in the design space of tools for data exploration. It has been particularly motivated by the use of data in journalism. The Gamma has the potential to enable journalists to make factual claims backed by data more commonplace and enable wider audience to engage with such claims, satisfying the *importance* criteria [54] for advancing the state of the art. It also satisfies a number of design goals important in the data journalism context.

Learning without experts: Our design aims to make The Gamma suitable for users who cannot dedicate significant amount of time to learning it in advance and may not have access to experts, satisfying the *empowering new participants* criteria [54]. This is supported in two ways.

First, the iterative prompting principle makes it easy for users to start experimenting. The user needs to select an initial data source and then repeatedly choose an item from a list of choices. This is easier to use than a command line or a REPL (read-eval-print-loop) interface, because it follows the *recognition over recall* usability heuristic. The users are not required to recall and type a command. They merely need to select one from a list of options.

Second, the resulting code serves as a trace of how the analysis was created. It provides the user with all information that they need to recreate the program, not just by copying it, but also by using iterative prompting. Such design has been called *design for percolation* [55] and it supports learnability. In Excel, studied by Sarkar [55], users learn new features when their usage is apparent in a spreadsheet, e.g. different functions

in formulas, but learning how to use a wizard for creating charts is hard because the operation does not leave a full trace in the spreadsheet.

Lowering barriers to entry: Data exploration has a certain irreducible essential complexity [56]. To make a system usable, this complexity needs to be carefully stratified. The Gamma uses a two level structure. The first level consists of the language itself with the iterative prompting mechanism. The second level consists of the individual members generated by a type provider. This can be seen as a *domain specific language*, embedded in The Gamma language. Although the complexity of individual domain specific languages differs, the user can always start exploring through iterative prompting, even when faced with an unfamiliar data source.

In tackling complexity, The Gamma satisfies two criteria proposed by Olsen [54]: *generality* in that it can be used uniformly with a wide range of data sources, and *expressive leverage* in that it factors out common aspects of different data queries into the core language (first level) and leaves the specifics of each data source to the second level.

Correctness and completeness: An important characteristic of our design is that the iterative prompting mechanism is both *correct* and *complete* with respect to possible data exploration scripts. The two properties are a consequence of the fact that a program is formed by a chain of operations and that the auto-completion leverages a static type system. When invoking iterative prompting at the end of a well-typed script, a selected option, which is a valid object member, is added to the end of the script, resulting in another well-typed script. This distinguishes our system from auto-completion based on machine learning, which may offer members not valid in a given context. Auto-completion lists offered via iterative prompting contain all available members and so the user can construct all possible scripts. Two exceptions to completeness in our current design are the let binding and specifying numerical parameters as in `take(5)`.

VIII. CONCLUSIONS

Exploring data in a programming environment that makes the full source code available increases transparency, reproducibility and empowers users to ask critical questions about the data analysis. But can we make those features accessible to non-programmers? In this paper, we presented The Gamma, a simple data exploration environment for non-programmers that answers this question in the affirmative.

The Gamma is based on a single interaction principle, *iterative prompting*. It can be used to complete a range of data exploration tasks using tabular data, data cubes and graph databases. The design lowers the barrier to entry for programmatic data exploration and makes it easy to learn the system independently through examples and by experimentation. We implemented The Gamma, made it available as open source and conducted a user study, which lets us conclude that The Gamma can be used by non-programmers to construct non-trivial data exploration scripts.

ACKNOWLEDGMENTS

We thank to May Yong and Nour Boulahcen for their contributions to The Gamma type providers. The author is also grateful to Don Syme, James Geddes, Jonathan Edwards and Roly Perera for numerous discussions about data science tooling and type providers, as well as Luke Church for discussions about human-computer interaction and Clemens Klokose for numerous suggestions on framing of this paper. Anonymous reviewers of this and earlier versions of the paper also provided valuable feedback. This work was partly supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1 and by a Google Digital News Initiative grant.

REFERENCES

- [1] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek, "Themes in information-rich functional programming for internet-scale data sources," in *Proceedings of Workshop on Data Driven Functional Programming*. ACM, 2013, pp. 1–4.
- [2] B. A. Myers, A. J. Ko, and M. M. Burnett, "Invited research overview: end-user programming," in *Extended Abstracts Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI '06*. ACM, 2006, pp. 75–80. [Online]. Available: <https://doi.org/10.1145/1125451.1125472>
- [3] B. A. Nardi, *A small matter of programming: perspectives on end user computing*. MIT press, 1993.
- [4] J. Gray, L. Chambers, and L. Bounegru, *The data journalism handbook: how journalists can use data to improve the news*. O'Reilly, 2012.
- [5] T. Petricek, G. Guerra, and D. Syme, "Types from data: Making structured data first-class citizens in F#," in *Proceedings of Conference on Programming Language Design and Implementation*, ser. PLDI '16. ACM, 2016, pp. 477–490.
- [6] G. E. Kaiser and P. H. Feiler, "An architecture for intelligent assistance in software development," in *Proceedings of the 9th International Conference on Software Engineering*, ser. ICSE '87. Washington, DC, USA: IEEE Computer Society Press, 1987, p. 180–188.
- [7] J. Heer, J. M. Hellerstein, and S. Kandel, "Predictive interaction for data transformation," in *CIDR*, 2015.
- [8] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, "Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011, pp. 65–74.
- [9] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang, "Eviza: A natural language interface for visual analysis," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST '16*. ACM, 2016, pp. 365–377. [Online]. Available: <https://doi.org/10.1145/2984511.2984588>
- [10] X. Rong, S. Yan, S. Oney, M. Dontcheva, and E. Adar, "Codemend: Assisting interactive programming with bimodal embedding," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST '16*. ACM, 2016, pp. 247–258.
- [11] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein, "Iris: A conversational agent for complex tasks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*. ACM, 2018, p. 473.
- [12] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM International Symposium on Foundations of Software Engineering*. ACM, 2009.
- [13] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*. ACM, 2014, pp. 419–428. [Online]. Available: <https://doi.org/10.1145/2594291.2594321>
- [14] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '19*. ACM, 2019, pp. 2727–2735. [Online]. Available: <https://doi.org/10.1145/3292500.3330699>
- [15] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks—a publishing format for reproducible computational workflows," in *20th International Conference on Electronic Publishing*, F. Loizides and B. Schmidt, Eds., 2016, pp. 87–90.
- [16] T. Petricek, J. Geddes, and C. A. Sutton, "Wrattler: Reproducible, live and polyglot notebooks," in *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018.*, M. Herschel, Ed., 2018.
- [17] R. Wesley, M. Eldridge, and P. T. Terlecki, "An analytic data engine for visualization in tableau," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 2011, pp. 1185–1194.
- [18] Microsoft Corporation. (2020) Microsoft power bi. [Online]. Available: <https://powerbi.microsoft.com/en-us/>
- [19] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, "Interactive data analysis: the control project," *Computer*, vol. 32, no. 8, pp. 51–59, 8 1999.
- [20] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska, "Vizdom: Interactive analytics through pen and touch," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 2024–2027, Aug. 2015.
- [21] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *VLDB*, vol. 1, 2001, pp. 381–390.
- [22] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: Interactive visual specification of data transformation scripts," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 3363–3372.
- [23] A. Satyanarayan and J. Heer, "Lyra: An interactive visualization design environment," in *Computer Graphics Forum*, vol. 33, no. 3, 2014, pp. 351–360.
- [24] B. Victor. (2012) Inventing on principle. [Online]. Available: <http://worrydream.com/InventingOnPrinciple>
- [25] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape, "Exploratory and live, programming and coding," *The Art, Science, and Engineering of Programming*, vol. 3, no. 1, 2019.
- [26] J. Kubelka, R. Robbes, and A. Bergel, "The road to live programming: Insights from the practice," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1090–1101.
- [27] C. Granger. (2012) Lighttable: A new IDE concept. [Online]. Available: <http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/>
- [28] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting exploratory programming by data scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, p. 1265–1276.
- [29] M. B. Kery and B. A. Myers, "Exploring exploratory programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, A. Henley, P. Rogers, and A. Sarma, Eds. IEEE, 2017, pp. 25–29.
- [30] E. Adar, "GUESS: a language and interface for graph exploration," in *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006*. ACM, 2006, pp. 791–800. [Online]. Available: <https://doi.org/10.1145/1124772.1124889>
- [31] G. Szwillus and L. Neal, *Structure-based editors and environments*. Academic Press, Inc., 1996.
- [32] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, "Live functional programming with typed holes," *PACMPL*, vol. 3, no. POPL, 2019.
- [33] E. Lotem and Y. Chuchem. (2018) Lamdu project. [Online]. Available: <https://github.com/lamdu/lamdu>
- [34] J. Edwards, "Subtext: uncovering the simplicity of programming," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 505–518, 2005.
- [35] —. (2018, 6) Direct programming. [Online]. Available: <https://vimeo.com/274771188>
- [36] M. Conlen and J. Heer, "Idyll: A markup language for authoring and publishing interactive articles on the web," in *The 31st Annual ACM Symposium on User Interface Software and Technology, UIST '18*. ACM, 2018, pp. 977–989. [Online]. Available: <https://doi.org/10.1145/3242587.3242600>
- [37] E. Jun, M. Daum, J. Roesch, S. Chasins, E. Berger, R. Just, and K. Reinecke, "Tea: A high-level language and runtime system for automating statistical analysis," in *Proceedings of the 32nd Annual ACM UIST Symposium 2019*. ACM, 2019, pp. 591–603.

- [38] A. Satyanarayan, K. Wongsuphasawat, and J. Heer, “Declarative interaction design for data visualization,” in *The 27th Annual ACM Symposium on User Interface Software and Technology, UIST '14*. ACM, 2014, pp. 669–678. [Online]. Available: <https://doi.org/10.1145/2642918.2647360>
- [39] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, “Vega-lite: A grammar of interactive graphics,” *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 341–350, 2016.
- [40] H. Lieberman, *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
- [41] S. Gulwani, W. R. Harris, and R. Singh, “Spreadsheet data manipulation using examples,” *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.
- [42] V. Le and S. Gulwani, “Flashextract: a framework for data extraction by examples,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 542–553.
- [43] E. L. Hutchins, J. D. Hollan, and D. A. Norman, “Direct manipulation interfaces,” *Human-Computer Interaction*, vol. 1, no. 4, pp. 311–338, 1985.
- [44] B. Hempel, J. Lubin, and R. Chugh, “Sketch-n-sketch: Output-directed programming for SVG,” in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST '19*. ACM, 2019, pp. 281–292. [Online]. Available: <https://doi.org/10.1145/3332165.3347925>
- [45] B. Shneiderman, C. Williamson, and C. Ahlberg, “Dynamic queries: Database searching by direct manipulation,” in *Conference on Human Factors in Computing Systems, CHI '92*. ACM, 1992, pp. 669–670. [Online]. Available: <https://doi.org/10.1145/142750.143082>
- [46] I. Bretan, R. Nilsson, and K. S. Hammarstrom, “V: a visual query language for a multimodal environment,” in *Conference on Human Factors in Computing Systems, CHI '94*, C. Plaisant, Ed. ACM, 1994, pp. 145–147. [Online]. Available: <https://doi.org/10.1145/259963.260174>
- [47] M. Derthick, J. Kolojechick, and S. F. Roth, “An interactive visual query environment for exploring data,” in *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology, UIST '97*. ACM, 1997, pp. 189–198. [Online]. Available: <https://doi.org/10.1145/263407.263545>
- [48] A. Abouzied, J. M. Hellerstein, and A. Silberschatz, “Dataplay: interactive tweaking and example-driven correction of graphical database queries,” in *The 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12*. ACM, 2012, pp. 207–218.
- [49] UK Parliament. (2021) Members’ allowances and expenses. [Online]. Available: <https://www.parliament.uk/mps-lords-and-offices/members-allowances/house-of-lords/holallowances/>
- [50] T. Petricek, “Foundations of a live data exploration environment,” *Art Sci. Eng. Program.*, vol. 4, no. 3, p. 8, 2020. [Online]. Available: <https://doi.org/10.22152/programming-journal.org/2020/4/8>
- [51] Microsoft Corporation. (2021) Monaco editor. [Online]. Available: <https://microsoft.github.io/monaco-editor/>
- [52] T. Petricek, “Data exploration through dot-driven development,” in *31st European Conference on Object-Oriented Programming*, 2017.
- [53] G. Myre. (2021) If michael Phelps were a country, where would his gold medal tally rank? [Online]. Available: <https://www.npr.org/sections/thetorch/2016/08/14/489832779/>
- [54] D. R. O. Jr., “Evaluating user interface systems research,” in *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, UIST '07*. ACM, 2007, pp. 251–258. [Online]. Available: <https://doi.org/10.1145/1294211.1294256>
- [55] A. Sarkar and A. D. Gordon, “How do people learn to use spreadsheets? (work in progress),” in *Proceedings of the 29th Annual Conference of the Psychology of Programming Interest Group (PPIG 2018)*, Sep. 2018, pp. 28–35.
- [56] J. Brooks, F.P., “No silver bullet essence and accidents of software engineering,” *Computer*, vol. 20, no. 4, pp. 10–19, april 1987.
- [57] J. Cheney, S. Chong, N. Foster, M. I. Seltzer, and S. Vansummeren, “Provenance: a future history,” in *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '09*. ACM, 2009, pp. 957–964.

Chapter 11

AI Assistants: A framework for semi-automated data wrangling

Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. 2023. AI Assistants: A Framework for Semi-Automated Data Wrangling. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 9295–9306. <https://doi.org/10.1109/TKDE.2022.3222538>

AI Assistants: A Framework for Semi-Automated Data Wrangling

Tomas Petricek, Gerrit J.J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, Christopher K. I. Williams

Abstract—Data wrangling tasks such as obtaining and linking data from various sources, transforming data formats, and correcting erroneous records, can constitute up to 80% of typical data engineering work. Despite the rise of machine learning and artificial intelligence, data wrangling remains a tedious and manual task. We introduce *AI assistants*, a class of semi-automatic interactive tools to streamline data wrangling. An AI assistant guides the analyst through a specific data wrangling task by recommending a suitable data transformation that respects the constraints obtained through interaction with the analyst.

We formally define the structure of AI assistants and describe how existing tools that treat data cleaning as an optimization problem fit the definition. We implement AI assistants for four common data wrangling tasks and make AI assistants easily accessible to data analysts in an open-source notebook environment for data science, by leveraging the common structure they follow. We evaluate our AI assistants both quantitatively and qualitatively through three example scenarios. We show that the unified and interactive design makes it easy to perform tasks that would be difficult to do manually or with a fully automatic tool.

Index Terms—Data Wrangling, Data Cleaning, Human-in-the-Loop

1 INTRODUCTION

WHILE most *research* in data science focuses on novel methods and clever algorithms, the *practice* is dominated by the realities of working with messy data. Surveys [1], [2] indicate that up to 80% of data engineering is spent on *data wrangling*, a tedious process of transforming data into a format suitable for analysis, which includes parsing, making sense of encodings, merging datasets, and correcting erroneous records. Data wrangling prevents both organizations and individuals from applying machine learning and represents an enormous cost, both in terms of wasted time and in terms of missed opportunities.

Despite attempts to address this issue [3], [4], data wrangling remains hard to automate, because it often involves special cases that require human insight. An automatic tool can easily confuse interesting outliers for uninteresting noise in cases where a human would immediately spot the difference. This makes incorporating human understanding into the process crucial. A major advance in the practice of data wrangling therefore requires semi-automated tools that integrate automatic methods with human insight, allow the analyst to review cleaning operations before applying them, and follow a unified interface that makes it easy to use a wide range of tools during data wrangling.

- T. Petricek (tomas@tomasp.net), Charles University, Prague, Czechia (work done while at University of Kent and The Alan Turing Institute)
- G.J.J. van den Burg (gertjanvandenburger@gmail.com), Amazon, UK (work done prior to joining Amazon, in The Alan Turing Institute)
- A. Nazabal (alfredonazabal@gmail.com) Amazon Development Centre Scotland, Edinburgh (work done prior to joining Amazon, in The Alan Turing Institute).
- T. Ceritli (taha.ceritli@eng.ox.ac.uk), University of Oxford, UK (work done in University of Edinburgh and The Alan Turing Institute)
- E. Jiménez-Ruiz (ernesto.jimenez-ruiz@city.ac.uk) City, University of London, UK and University of Oslo, Norway
- C. K. I. Williams (ckiw@inf.ed.ac.uk) University of Edinburgh and The Alan Turing Institute, UK

1.1 Background

Data wrangling is most often done manually using a combination of programmatic and graphical tools. Jupyter and RStudio are popular environments used for programmatic data cleaning. They are used alongside libraries that implement specific functionality such as parsing CSV files or merging datasets [5], [6] and general data transformation functions provided, e.g., by Pandas [7] and Tidyverse [8].

Trifacta [9] and OpenRefine [10] are complete graphical data wrangling systems that consist of myriad tools for importing and transforming data, which are accessible through different user interfaces or through a scriptable programmatic interface. Finally, spreadsheet applications such as Excel and business intelligence tools like Tableau [11] are often used for manual data editing, reshaping, and especially visualization [12]. The above general-purpose systems are frequently complemented by ad-hoc tools such as Tabula [13], which extracts tables from PDF documents.

1.1.1 Semi-automatic data wrangling

Some of the most practical tools along the entire data wrangling pipeline partially automate a specific tedious data wrangling task. To merge datasets, Trifacta [9] and datadiff [6] find corresponding columns using machine learning. To transform textual data and tables, Excel [14] employs programming-by-example to parse semistructured data, LearnPADS [15] automatically generates programmatic data processing routines, and many tools exist to semi-automatically detect duplicate records in databases [16].

A common theme in data wrangling tools that utilize machine learning, including those listed above, is that they allow the analyst to review and influence the results. The interaction between a human and a computer in such data wrangling systems follows a number of common patterns:

- *Onetime interaction.* A tool makes a best guess, but allows the analyst to manually edit the proposed data transformation. Examples include LearnPADS [15] and dataset merging in Trifacta [9] and datadiff [6].
- *Live previews.* Environments like Jupyter, Trifacta [9], and The Gamma [17] provide live previews, allowing the analyst to check the results and tweak parameters of the operation they are performing before moving on.
- *Iterative.* A tool re-runs inference after each interaction with a human to refine the result. For example, in Predictive Interaction [18] the analyst repeatedly selects examples to construct a data transformation.
- *Question-based.* A system repeatedly asks the human questions about data and uses the answers to infer and refine a general data model. Examples include data repair tools such as UGuide [19], [20].

The interaction pattern that combines human inputs and automatic inference is also known as mixed-initiative interfaces [21], [22] in the context of graphical user interfaces, and as human-in-the-loop data analytics (HILDA) [22], [23] in the context of data science. However, both of these are general patterns, rather than specific technical frameworks.

1.1.2 Issues and limitations

The emerging class of semi-automatic data wrangling tools have the potential to dramatically simplify data wrangling because they combine the automation and scalability of machine learning with crucial human insight. However, this development has been hindered by two main issues.

First, semi-automatic data wrangling tools lack a common structure. Notions such as mixed-initiative user interfaces and human-in-the-loop are too general and do not provide a specific technological framework that a tool implementation could follow. Moreover, many tools only exist in one specific environment or programming language, forcing the analyst to repeatedly switch between tools. They may, for example, need to export data from Trifacta to a CSV file, run a particular R or Python script and then import data back. This is not without risk, as intermediate data formats may accidentally corrupt data.

Second, the way analysts interact with such tools can vary significantly. Consequently, users have to learn how to interact with each new tool using whatever mechanism it supports, be it a graphical user interface, a program library, or a command-line script. Moreover, most semi-automatic data wrangling tools accept only limited forms of human input. The *onetime interaction* pattern of interaction prevails and only a few systems [18], [24] follow the flexible *iterative* pattern. Even then, the way of specifying feedback in such systems is often specialized and tied to the problem domain.

1.2 Contributions

We present the notion of an *AI assistant*, a common structure for building semi-automatic data wrangling tools that incorporate human feedback. AI assistants capture the *iterative* pattern of interaction where a human user repeatedly provides insights about the problem and a computer performs automatic inference. The design addresses the issues with semi-automatic data wrangling tools described above.

First, the AI assistant framework allows for a wide range of semi-automatic data wrangling tools that can integrate human feedback. For analysts, this makes using AI assistants easy as they can complete a variety of data wrangling tasks through a uniform user interface. For tool developers, this makes building AI assistants easier, because any AI assistant can be readily used from JupyterLab and potentially other data wrangling systems.

Second, the notion of an AI assistant defines a simple uniform mechanism for iteratively providing feedback to the assistants. An AI assistant makes an initial best guess and then it repeatedly offers the analyst a list of options that they can choose from in order to guide the next iteration of the automatic process.

The remainder of this paper is structured as follows:

- We introduce AI assistants by example in Section 2, looking at how the datadiff AI assistant simplifies merging data from inconsistent datasets.
- We define the structure of AI assistants formally in Section 3 and show how tools solving an optimization problem fit the definition.
- We present four AI assistants in Section 4 (for parsing, merging, type inference, and semantic type prediction), that each restructure an existing non-interactive tool as an interactive AI assistant.
- We evaluate our approach in Section 5 qualitatively, by discussing three scenarios where automatic tools would fail, and quantitatively, by evaluating how many interactions are needed to complete a wrangling task.

While we may not entirely eliminate the 80% of time data scientists spend on data wrangling, our framework provides a pathway to the future where data analysts leverage the advances in AI for the most time-consuming aspect of their job. The four AI assistants we develop illustrate the benefits that a rich ecosystem of AI assistants would provide.

2 MOTIVATION

To give an overview of how AI assistants work, we discuss the data wrangling task of merging multiple incompatible datasets, using the UK broadband quality data [25], published by the UK communications regulator Ofcom.

The regulator collects data annually, but the formats of the files are inconsistent over the years. The order of columns changes, some columns are renamed, and new columns are added. We take the 2014 dataset and select six interesting columns (latency, download and upload speed, time needed to load a sample page, country, and whether the observation is from an urban or a rural area). We then want to find corresponding columns in the 2015 dataset.

The 2015 dataset has 66 different columns so finding corresponding columns manually would be tedious. Instead, we can use the automatic datadiff tool [6], which matches columns by analyzing the distributions of the data in each column. Datadiff generates a list of *patches* that reconcile the structure of the two datasets. A patch describes a single data transformation to, for example, reorder columns or recode a categorical column according to an inferred mapping. Datadiff is available as an R function that takes two datasets and several hyperparameters that affect the likelihood of

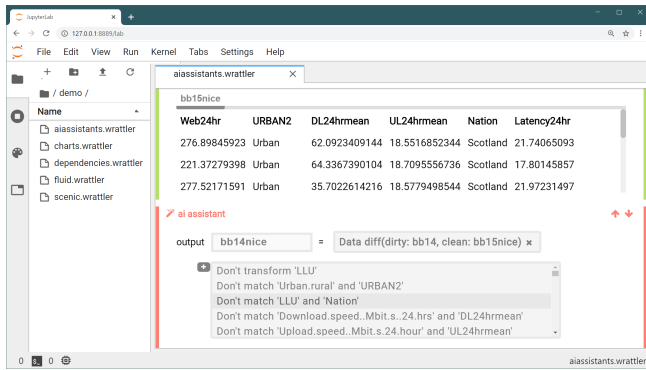


Fig. 1. Using the datadiff AI assistant in JupyterLab to semi-automatically merge data from two sources, parsed by an earlier R script.

the different types of patches. Datadiff correctly matches five out of six columns, but it incorrectly attempts to match a column representing Local-loop unbundling (LLU) to a column representing UK countries. This happens as datadiff allows the recoding of categorical columns, and seeks to match them based on the relative frequencies in the two columns. Consequently, the inferred transformation includes a patch to recode the Cable, LLU, and Non-LLU values to Scotland, Wales, and England. To correct this, we could either manually edit the resulting list of patches, or tweak the likelihood of the *recode* patch. Such parameter tuning is typical for real-world data wrangling, but finding the values that give the desired result can be hard.

The semi-automatic datadiff AI assistant presented in this paper enables the analyst to guide the inference process by specifying human insights in the form of constraints. The AI assistant first suggests an initial set of patches with one incorrect mapping. After the analyst chooses one of the offered constraints, shown in Figure 1, datadiff runs again and presents a new solution that respects the specified constraints until, after two more simple interactions, it reaches the correct solution (see Section 5.1.1 for details).

The example illustrates the interaction pattern at the core of AI assistants. At each step, the assistant analyzes the data and recommends the best data transformation. It previews the transformed data and offers a range of constraints that may be sorted by their estimated fit. The analyst can then accept the result or choose another constraint to refine the outcome. The behaviour is controlled through comprehensible constraints rather than opaque numerical parameters.

As discussed in Appendix D (see supplemental files), we make AI assistants available in JupyterLab, which allows analysts to combine text and equations with code and outputs such as charts. We introduce a new cell type that leverages the common structure of AI assistants to provide a unified user interface (see Figure 1) for accessing any AI assistant.

3 THEORY OF AI ASSISTANTS

The notion of an *AI assistant* formally captures a pattern of interaction between a semi-automatic data wrangling tool and a data analyst. The precise definition distinguishes AI assistants from more general notions such as human-in-the-loop data analytics, and it facilitates the development of concrete AI assistants discussed in Section 4.

3.1 Formal model

Our definition uses the algebraic approach [26] and presents AI assistants as a formal mathematical entity that consists of several operations, modeled as mathematical functions between different sets. For reference, a glossary of symbols used in the paper can be found in Table 5 (supplement).

Every AI assistant is defined by three operations that work with expressions e , past human interactions H , input data X , and output data Y . While AI assistants share a common structure, the language of expressions e that an assistant produces, the notion of human interactions H , and the notion of X and Y can differ between assistants.

We refer to e as expressions, following the programming research tradition, but expressions e can also be thought of as data cleaning scripts. As we will see from our concrete examples, the input data X is typically one or more data tables and the output data Y is typically a single table, often annotated with meta-data such as column types.

Definition 3.1 (AI assistant). Given expressions e , input data X , output data Y , and human interactions H , an *AI assistant* $(H_0, f, best, choices)$ is a tuple where H_0 is a set denoting an empty human interaction and f , $best$ and $choices$ are operations such that:

- $f(e, X) = Y$
- $best_X(H) = e$
- $choices_X(H) = (H_1, H_2, H_3, \dots, H_n)$.

The operation f transforms an input dataset X into an output dataset Y according to the expression e . The operation $best_X$ recommends the best expression for a given input dataset X , respecting past human interactions H . Finally, the operation $choices_X$ generates a sequence of options $H_1, H_2, H_3, \dots, H_n$ that the analyst can choose from (for instance through the user interface illustrated in Figure 1). When interacting with an assistant, the selected human interaction H is passed back to $best_X$ in order to refine the recommended expression. Note that the sequence of human interactions given by $choices_X$ may be sorted, starting with the one deemed the most likely. To initialize this process, the AI assistant defines an empty human interaction H_0 .

The interesting AI logic can be implemented in either the $best_X$ operation, the $choices_X$ operation, or both. The f operation is typically straightforward. It merely executes the inferred cleaning script. Both $best_X$ and $choices_X$ are parameterized by input data X , which could be the actual input or a smaller representative subset, such as coresets [27], to make working with the assistant more efficient.

The logic of working with AI assistants is illustrated in Figure 2. When using the assistant, we start with the empty interaction H_0 . We then iterate until the human analyst accepts a proposed data transformation. In each iteration, we first invoke $best_X(H)$ to get the best expression e^* respecting the current human insights captured by H . We then invoke $f(e^*, X)$ to transform the input data X according to e^* and obtain a transformed output dataset Y . After seeing a preview of Y , the analyst can either accept or reject the recommended expression e^* . In the latter case, we generate a list of possible human interactions $H_1, H_2, H_3, \dots, H_n$ using $choices_X(H)$ and ask the analyst to pick an option H_i (where $i \in \{1, \dots, n\}$). We use this choice as a new human interaction H and call the AI assistant again.

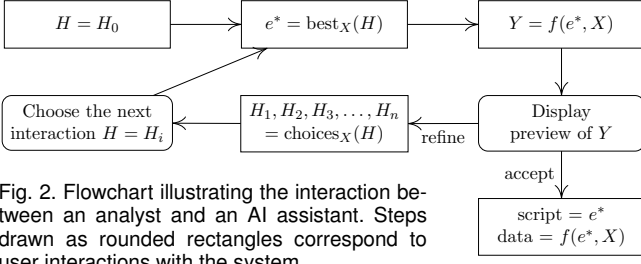


Fig. 2. Flowchart illustrating the interaction between an analyst and an AI assistant. Steps drawn as rounded rectangles correspond to user interactions with the system.

Having a unified structure for AI assistants means that we can separate the development of individual AI assistants from the development of tools that use them. Our Jupyter-Lab implementation facilitates access to any AI assistant that adheres to the interface captured by Definition 3.1.

3.2 Example

To provide intuition behind the operations, we return to the semi-automatic datadiff AI assistant introduced in Section 2 and presented in full in Section 4.1. In case of datadiff, an expression e is a list of patches. Input data X is a pair of data tables comprising a reference dataset and an input dataset. The output data Y is the input dataset, transformed to the format of the reference dataset. Finally, human interactions H are lists of constraints that restrict what expressions are permissible. An example constraint, discussed earlier, prevents the matching of particular columns.

The most interesting aspect of the assistant is the $best_X$ operation. It takes a sample input X together with a trace of human interactions H , which is a list of constraints. It then finds the best way to match the columns from the two datasets, utilizing the algorithm from the original datadiff [6], but respecting the constraints. The result is a list of patches, which is returned as an expression e . The $choices_X$ operation generates a list of choices $H_1, H_2, H_3, \dots, H_n$. An individual choice is obtained by taking the constraints specified earlier and adding one additional constraint that restricts some aspect of the recommended script, e.g., recoding of a column that was recommended in the expression e . Finally, f applies the list of patches to the input data.

In datadiff, the clever algorithmics are done in the $best_X$ operation, while $choices_X$ is simpler. It generates constraints in a simple hard-coded way, although a more elaborate AI assistant could rank these constraints.

3.3 Optimization perspective

Our definition of an AI assistant is purposefully general, but the way most AI assistants recommend cleaning scripts is based on the optimization of an objective function. They attempt to find the best cleaning script for a given problem from a set of possible cleaning scripts. The best script is determined by an objective function Q that scores expressions based on how well they clean the specified input data.

As before, we write e for expressions (cleaning scripts) that an AI assistant recommends and H for human interactions. The operation $best_X(H)$ solves an optimization problem based on an objective function Q_H to identify the best expression e^* from a set E_H of possible expressions. Note that both Q_H and E_H are parameterized by human

interactions, meaning that interaction with the tool can affect both the optimization objective and the set of permitted expressions. More formally, given Q_H and E_H where:

- $Q_H(X, e)$ is an objective function that assigns a score to an expression e , applied to input data X , taking into account human interaction H ;
- E_H is a set of permitted expressions with respect to human interaction H ,

we define $best_X$ as solving an optimization problem:

$$best_X(H) = \arg \max_{e \in E_H} Q_H(X, e)$$

The objective function Q needs to be defined individually for each AI assistant. It typically uses a measure of how clean the data is after applying the expression e . For datadiff, Q is computed as a sum of distance measures between the empirical distributions of the corresponding columns [6]. The optimization based on Q is also implemented individually for each AI assistant and is discussed in the next section.

The fact that both E_H and Q_H are parameterized by H makes the definition more flexible. One human interaction can entirely prevent the assistant from generating certain expressions (by removing them from E_H), while another can make a particular expression less desirable (by decreasing the score assigned to it by Q_H). For example, human interactions in datadiff restrict the set of allowed expressions E_H , but do not affect the objective function Q_H .

4 PRACTICAL AI ASSISTANTS

In this section, we show how to turn four existing non-interactive data wrangling tools into interactive AI assistants. An example of a newly developed assistant for outlier detection is discussed in Appendix C (supplemental files).

4.1 datadiff: Merging mismatched data tables

We start by revisiting the datadiff AI assistant. The original R package [6] implements a function that returns the inferred best list of patches. We modify the package to support restricting the optimization using constraints specified by the analyst, and use it as the basis for an interactive AI assistant. The following formal model explains how the AI assistant uses the underlying optimization and generates choices that the analyst can use to control the assistant.

Formal definition of datadiff. The input data for the assistant is a pair of data tables T_i, T_r representing the input and reference datasets, respectively. The expression e is a sequence of patches and human interactions H are lists of constraints that restrict what patches can be generated. Patches P and constraints c are defined as:

$$\begin{aligned}
 P &= \text{recode}(k, [v_1 \mapsto v'_1, \dots]) \mid \text{linear}(k, a, b) \mid \\
 &\quad \text{delete}(k) \mid \text{insert}(k) \mid \text{permute}(\pi) \\
 c &= \text{nomatch}(k, l) \mid \text{notransform}(k) \mid \text{match}(k, l)
 \end{aligned}$$

The $\text{recode}(k, [v_1 \mapsto v'_1, \dots])$ patch transforms categorical values in a column k by replacing old values v_i with new values v'_i , $\text{linear}(k, a, b)$ transforms values v in a numerical column k using a linear transformation $v \cdot a + b$, $\text{insert}(k)$ inserts a new column at an index k , $\text{delete}(k)$ removes a column at an index k , and $\text{permute}(\pi)$ reorders columns according to a permutation π .

Interacting with the assistant results in a list of constraints: `nottransform(k)` prevents recoding or linear transformation in a column k , while `nomatch(k, l)` and `match(k, l)` prevent or enforce matching of columns k from the input dataset to the column l in the reference dataset.

The operations of `datadiff` follow the optimization-based framework where $best_X(H)$ finds a list of patches from the set E_H that maximizes the objective function Q :

$$best_X(H) = \arg \max_{e \in E_H} Q(X, e) \text{ where } E_H = \{(P_1, \dots, P_L) \in E \mid \forall i \in 1 \dots L. \text{valid}_H(P_i)\}$$

The objective function, discussed below, is not affected by human interaction, but human interactions do limit the set of allowed expression E_H . This is captured by the valid_H predicate defined in Appendix A (see supplemental files). Briefly, a list of patches P is valid if it does not recode or rescale any column specified in `norecode` and if the permutation given in `permute(π)` is compatible with the `match` and `nomatch` constraints.

The $choices_X$ operation, also given in Appendix A, offers constraints based on the best patch set obtained from calling $best_X$. Each human interaction adds one additional constraint to the current set of constraints H . The constraints allow the analyst to override some aspect of the generated patch. For any `recode` or `linear` patch, we offer the `nottransform` constraint to block the transformation. For any matched columns, we offer `nomatch`, and for any columns that were not automatically matched we offer the `match` constraint to manually match them. In the example discussed above, the assistant recommends an incorrect `recode` patch. The first interaction offered by $choices_X$ is to add the `nottransform` constraint to prevent this matching.

Objective function optimization. We use the same objective function Q as non-interactive `datadiff` [6]. Given the input and reference datasets and a set of patches to apply, the objective function sums the distances between the distributions of the matched columns, using the Kolmogorov-Smirnov statistic for numerical columns and the total variation (TV) statistic for categorical columns.

The optimization algorithm employed in `datadiff` first computes the optimal patch for all pairs of columns producing a cost matrix. The optimal matching is then determined by running the Hungarian algorithm [28]. Our modification incorporates the constraints specified by the user by not applying recoding where prevented by a constraint and by setting the cost of columns that should or should not be matched to zero and infinity, respectively. Details and performance considerations can be found in Appendix B.

Example of using datadiff. Suppose we have two data tables and we want to transform the input table T_i on the left to match the format of the reference table T_r on the right. The following shows the header and the first three rows:

City, Name, Count	Name, City
Cardiff, Alice, 1	Joe, London
Cardiff, Bob, na	Jane, Edinburgh
Edinburgh, Bill, 2	Jim, London

The original `datadiff` recommends three patches: `delete(3)`, `permute(2, 1)` and `recode(2, ["Cardiff" \mapsto "London"])`.

`Datadiff` correctly infers that we need to drop the `Count` column and that the order of `Name` and `City` has been switched. It erroneously infers that the encoding of a categorical column `City` has been changed. This would be useful for pairs of values like "true", "false" and "yes", "no", but it is incorrect in the case of cities.

Using the interactive `datadiff`, the analysts can specify the `nottransform(2)` constraint, which will prevent `datadiff` from generating the `recode` patch for the column 2. The interactive AI assistant makes such an intervention easy, because it offers the constraint via the $choices_X$ operation and the analyst can simply select it from a drop-down menu.

4.2 CleverCSV: Parsing tabular data files

While parsing CSV files in the standard format [29] is easy, parsing a file with non-standard column separators and other formatting parameters often requires human insight. `CleverCSV` [5] is an automatic tool that uses a data consistency measure to determine formatting parameters, called a "dialect", consisting of the delimiter (e.g., `,`), quote (e.g., `"`) and escape characters (e.g., `\`). We adapt `CleverCSV` into an interactive AI assistant that allows the analyst to guide the tool in case the automatic detection fails.

Formal definition of CleverCSV. `CleverCSV` is an optimization-based assistant that takes a single string, representing the CSV file, as the input data X . The objective function $Q(X, e)$ is defined by the data consistency measure discussed below, expressions e represent dialects, and $Y = f(e, X)$ denotes the result of parsing the file using a given dialect. Human interactions place constraints on the characters that are considered for each parameter and can either fix a dialect parameter to a specific value or block a character from being considered:

$$\begin{aligned} e &= (\text{is_delimiter}(d), \text{is_quote}(q), \text{is_escape}(a)) \\ c &= \text{fix_delimiter}(d) \mid \text{fix_quote}(q) \mid \text{fix_escape}(a) \\ &\quad \mid \text{not_delimiter}(d) \mid \text{not_quote}(q) \mid \text{not_escape}(a) \end{aligned}$$

The operations that define the `CleverCSV` AI assistant follow the same structure as those of `datadiff` and are shown in Appendix A (see supplemental files). The $best_X$ operation optimizes the objective function $Q(X, e)$ over a set E_H , consisting of dialects compatible with the current constraints H . The $choices_X(H)$ operation can take advantage of the consistency score $Q(X, e)$ computed for each dialect under consideration to sort the suggested constraints.

Objective function optimization. The objective function Q_H for the AI assistant does not depend on user interactions and uses the consistency measure of non-interactive `CleverCSV` [5]. The measure is calculated by parsing the input file using a potential dialect and taking the product of two scores: the "pattern score" that captures how regular the structure of the parsed data is (i.e., does the resulting table have the same number of cells in each row?), and the "type score" that captures the proportion of cells that have an identifiable data type. The optimization involves iterating over each possible dialect allowed by the constraints H , and identifying the one that maximizes the objective function. Further details on the optimization and runtime performance of `CleverCSV` can be found in Appendix B.

Example of using CleverCSV. While the automatic dialect detection proposed in [5] achieves 97% accuracy, one type of failure arises when there are *two* delimiters that result in consistent row lengths and interpretable cells:

```
"{"name": "John", "age": "28"}", 22:34:00, 01:16:40
{"name": "Sara", "age": "26"}", 18:28:02, 19:32:37
{"name": "Bill", "age": "31"}", 02:51:34, 10:14:58
{"name": "Jane", "age": "18"}", 13:06:36, 16:59:47
```

A dialect with colon (:) as the column separator maximizes the consistency measure even though comma (,) is the correct separator. This happens because splitting the data on the colon character results in regular row lengths and because the JSON syntax in the first column is an unknown data type for CleverCSV. The correct dialect receives the second-highest consistency score and it differs from the chosen dialect only in the delimiter character. This can be corrected with single interaction. In fact, $choices_X(H)$ could automatically propose the constraint `fix_delimiter(,)` first.

4.3 ptype: Inferring column types

After parsing data, the next step is often to identify the data types for each column. This becomes challenging in the presence of missing and anomalous data. The probabilistic type detection package ptype [30] uses a Probabilistic Finite-State Machine model to solve this problem with an overall accuracy of 93%, but lower for data types like dates. We recast ptype as an interactive AI assistant that allows the data analyst to correct errors in those situations.

Formal definition of ptype. For simplicity, we consider input data X with just a single column. The expression e represents inferred information for the column and consists of the inferred column type τ and sets of values which (conditional on that type) are deemed missing and anomalous. Human interactions H allow the analyst to constrain the type (τ), missing values (u), and anomalous values (v):

$$e = (\tau, \{u_1, \dots, u_k\}, \{v_1, \dots, v_l\})$$

$$c = \text{not_type}(\tau) \mid \text{not_missing}(u) \mid \text{not_anomaly}(v)$$

The `not_type(τ)` constraint marks τ as an incorrect column type, while `not_missing(u)` and `not_anomaly(v)` prevent ptype from treating values u, v as missing and anomalous.

Probabilistic model. The objective function for ptype is derived from a probabilistic model that views expressions e as parameters of the transformation f and human interactions H as a meta-parameter that adjusts the likelihood of values in the parameter space. The $Q_H(X, e)$ function is derived from two probability distributions:

- $p_H(X|e)$ denotes the likelihood of the input data X given an expression e , which represents a type alongside with missing and anomalous values
- $p_H(e)$ is a distribution over the expressions, representing prior beliefs about probabilities of expressions, i.e., types with missing and anomalous values

The probability distributions are written as p_H because, in general, a human interaction can change the shape of the distribution as well as its parameters. In the case of ptype, human interactions do not affect the probability distributions, but are used later when selecting the solution from a distribution over types.

The objective of a probabilistic AI assistant such as ptype is to maximize the posterior probability distribution of the set of expressions given the data. This is obtained from the prior distribution over the expressions $p(e)$ and the likelihood model $p(X|e)$ using Bayes' rule:

$$Q_H(X, e) = p_H(e|X) \propto p_H(X|e)p_H(e)$$

The operation $best_X(H)$ then takes the type with the highest probability according to $Q_H(X, e)$ that is compatible with the constraints specified by the user. If there are no constraints, this is the maximum a posteriori (MAP) solution. If the MAP solution is incorrect, the analyst can choose the `not_type` constraint to obtain the next most likely value.

Implementation. Since non-interactive ptype [30] infers the posterior distribution, our interactive tool only needs to select the most likely solution compatible with the specified constraints. Interestingly, this is a general approach which can be implemented for any tool based on a probabilistic framework, regardless of the particular problem it solves.

When generating constraints, $choices_X$ allows the analyst to mark a type as incorrect, but also to mark values inferred as anomalous or missing as valid. This forces the assistant to choose the best type that considers them as normal. The formal model of the logic is given in Appendix A.

4.4 ColNet: Semantic type prediction

Annotating data with semantic information can further assist data analysis. Tools like OpenRefine [10] and ColNet [31] automatically annotate tabular data with semantic types such as `dbo:Company` and `dbo:Person` obtained from a knowledge graph [32] such as DBpedia [33]. This may fail when data contain ambiguous values such as "Apple" or "Virgin" or values that do not exist in the knowledge graph (e.g., non-famous people [34]). We present an AI assistant based on ColNet (currently under development) that lets the analyst resolve such errors.

Formal definition of ColNet. ColNet is an optimization-based AI assistant, but it has a different structure than datadiff and CleverCSV. For simplicity, we consider a single-column input X formed by a set of values v_i . When inferring the semantic type, ColNet uses a set of samples S_1, S_2, \dots, S_n which each contain several individual values from the input data. The sampling method is discussed in [31].

The expression produced by the assistant is a single semantic type σ , to be attached to the dataset. The analyst can influence the result by specifying a list of constraints. The constraints `is_type(S, σ)` and `not_type(S, σ)` override the automatically inferred type for a given sample S .

In contrast to the constraints used in ptype, the constraints used by ColNet override the type of individual samples, rather than the overall type of a column. A constraint does not fix a type of the column, but merely provides a hint regarding one of several samples.

Objective function optimization. Non-interactive ColNet [31] pre-trains a Convolutional Neural Network (CNN) model for each (relevant) semantic type in the knowledge graph and fine-tunes the model with information from the column to be annotated. The CNN is then used to rank the possible semantic types obtained by querying the knowledge graph.

Given a set of samples \mathbb{S} from a given column, non-interactive ColNet predicts a score p_S^σ in $[0, 1]$ for each sample $S \in \mathbb{S}$ and semantic type σ . The score indicates the likelihood that values in S have a type σ . ColNet then averages scores over given samples (i.e., $p_S^\sigma = \frac{1}{|\mathbb{S}|} \sum_{S \in \mathbb{S}} p_S^\sigma$) and chooses the semantic type σ with the largest score. In interactive ColNet, human interactions affect the scoring of samples. Assuming p_S^σ is the score given by non-interactive ColNet, the interactive AI assistant uses $q_{H,S}^\sigma$ defined as:

$$q_{H,S}^\sigma = \begin{cases} 1 & \text{when } \text{is_type}(S, \sigma) \in H \\ 0 & \text{when } \text{not_type}(S, \sigma) \in H \\ p_S^\sigma & \text{otherwise} \end{cases}$$

The objective function $Q_H(X, e)$ is defined by the overall score $q_{H,S}^\sigma$, computed as the average of scores of individual samples, i.e., $q_{H,S}^\sigma = \frac{1}{|\mathbb{S}|} \sum_{S \in \mathbb{S}} q_{H,S}^\sigma$.

The best_X operation searches for a semantic type s (from a knowledge graph G) that maximizes the objective function Q_H . The constraints offered by choices_X allow the analyst to mark any of the samples $S \in \mathbb{S}$ as either having or not having a predicted type and are generated as follows:

$$\text{choices}_X(H) = \{\text{is_type}(S, \sigma), \text{not_type}(S, \sigma) \mid S \in \mathbb{S}, \sigma \in G, p_S^\sigma \geq \epsilon\}$$

To offer only relevant types, the constraint generation can be limited to types with a score greater than a threshold ϵ .

Example of using ColNet. One of the columns in the broadband quality data (Section 2) includes company names Virgin, BT, Sky, and Vodafone. Non-interactive ColNet predicts the semantic types (with an associated score in parentheses): `dbo:Work` (0.6), `dbo:Company` (0.5) and `dbo:Person` (0.4).

The correct type `dbo:Company` is not in the top position. This case is complex due to the use of acronyms (BT) and ambiguous entries (Virgin). In the case of Virgin, the expected entity is `dbr:Virgin_media`, but ColNet also finds `dbr:Virgin_of_the_Rocks` (a painting of type `dbo:Work`) and `dbr:Mary_mother_of_Jesus` (type `dbo:Person`).

To resolve the ambiguity regarding Virgin and obtain the expected semantic type, the analyst can specify a constraint `is_type({"Virgin"}, dbo:Company)`, which fixes the semantic type for the sample $S = \{\text{"Virgin"}\}$. This constraint indirectly decreases the likelihood that the types `dbo:Work` or `dbo:Person` will be inferred as the best semantic type.

5 EVALUATION

The previous section shows that the notion of an AI assistant captures a wide range of practical semi-automatic data wrangling tools. In this section, we evaluate the specific AI assistants that were presented. In Section 5.1, we use three scenarios to compare our tools with the state of the art systems. In Section 5.2, we quantify how many human interactions are needed to obtain the correct result with AI assistants in cases where the state of the art automatic tool fails. Performance is discussed in Appendix B (see supplement).

For the evaluation, we use real-world datasets from various sources with manually identified ground truth (CleverCSV, ptype) or synthetic dataset with ground truth known by construction (datadiff). A summary of datasets used can be found in Table 4 in the Appendix (see supplemental files).

TABLE 1
Comparing Ofcom broadband quality data for 2014 and 2015.

Name ('15)	Col ('15)	Name ('14)	Col ('14)
UL24hrmean	18	Upload (Mbit/s)24-hour	13
Web24hr	32	Web page (ms)24-hour	28
DL24hrmean	14	Download (Mbit/s) 24 hrs	10
URBAN2	10	Urban/rural	4
Nation	11	N/A	N/A
Latency24hr	30	Latency (ms)24-hour	24

5.1 Data wrangling scenarios

We first consider four real-world data wrangling scenarios based either on a problem from the literature [5], [30], [35] or earlier data analyses done by the authors.

5.1.1 datadiff: Merging Broadband data

For datadiff, we expand the example from Section 2. The analyst wants to analyze the change in broadband quality and needs to merge data for years 2014 and 2015. She selects six columns from 2015 and uses datadiff to find corresponding columns from 2014. Table 1 shows the relevant column names and indices, which have changed between years. Note that "Nation" (a categorical column with values England, Wales, Scotland) has been added in 2015. The analyst obtains the correct result after three interactions:

- 1) datadiff matches `Nation` with `LLU`, a categorical column with three values (LLU, non-LLU and Cable). The analyst chooses "Don't match LLU and Nation".
- 2) datadiff matches `Nation` with `Urban.rural`, another categorical column with three values. The analyst selects "Don't match Urban.rural and Nation".
- 3) datadiff matches `Nation` with `Technology`, yet another categorical column with three values. The analyst chooses "Don't match Technology and Nation".
- 4) datadiff correctly identifies that `Nation` has no corresponding column in 2014 and generates an *insert* patch to add a new empty column.

In all three interactions, the analyst immediately notices that there is one incorrectly matched column and selects a `nomatch` constraint. In non-interactive datadiff [6], the analyst would have to manually edit the initial set of patches (returned as an R object) or tweak one of the datadiff hyperparameters. Either of those is more complex than choosing three constraint with informative labels.

In Trifacta [9], the same task can be solved by using the "Add Union" operation. Here, the analyst chooses the 2015 dataset, selects the desired 6 columns and then adds the 2014 dataset. Choosing "auto align" invokes a proprietary algorithm that attempts to find matching columns using column names, column types, and similarity between sampled data.

At the time of writing, the algorithm aligned two of the columns ("UL24hrmean" and "Latency24hr") and provided no mapping for the remaining four that have to be matched manually using a graphical user interface. In other words, Trifacta is less successful in guessing the initial matching but, more importantly, it also only implements the *onetime interaction* model where analyst invokes the automatic tool once, but then has to correct all errors manually.

5.1.2 CleverCSV: Parsing large and messy CSV files

Dialect detection can seem a trivial task, but large CSV files can hide problems that are difficult to detect manually. We consider two scenarios: one where CleverCSV infers the dialect correctly and one where a single human interaction is needed.

First, consider the Internet Movie Database file¹, which contains descriptive statistics for 14,762 movies. A few rows and columns from the file look as follows:

```

1 fn,title,imdbRating
2 titles01/tt0015864,Goldrausch (1925),8.3
3 titles01/tt0017136,Metropolis (1927),8.4
4 titles01/tt0017925,Der General (1926),8.3
5 titles02/tt0080388,Atlantic City\,USA (1980),7.4

```

In this case, CleverCSV infers the correct dialect fully automatically. The standard R and Python functions fail to identify the escape character (`\`) which is used for movies with a comma in the title (line 5) and load 15,190 and 13,928 rows, respectively. Trifacta [9] also does not correctly handle the escape character. It assumes the file has three columns due to the first row and silently merges the additional data into the last column. OpenRefine [10] instead adds a fourth column due to the fact that the last row contains three delimiters. Such failures can be very time consuming to address and neither Trifacta nor OpenRefine provide straightforward mechanisms to mitigate this problem.

In cases when CleverCSV does not automatically detect the correct dialect, the AI assistant shows a preview of the parsed CSV file to the analyst, who can steer CleverCSV in the right direction. The following shows a few lines of a CSV file that contains filenames and RGB color codes²:

```

1 1894_0.jpg 51,47,45 87,88,86 110,112,110
2 1895_0.jpg 37,25,24 87,59,47 105,88,88
3 1895_1.jpg 48,34,46 80,51,58 98,80,88
4 1901_0.jpg 45,46,55 100,96,91 115,139,129
5 1901_1.jpg 45,46,48 71,66,61 98,97,94

```

For this file, CleverCSV predicts the comma as the delimiter even though the tab character is used. The analyst notices the issue easily thanks to the provided preview and can fix the parsing through a single interaction: by choosing the `fix_delimiter(\t)` constraint to set the correct delimiter.

For the same file, OpenRefine chooses underscore as the delimiter, whereas Trifacta uses the comma character. While in this case the user can select the correct delimiter in OpenRefine, this is not the case in Trifacta, where additional manual interaction is needed to get the data to a usable state.

5.1.3 ptype: Annotating the Cylinder Bands dataset

For the type inference task, we consider the Cylinder Bands dataset from the UCI repository [36]. The file contains data on process defects known as “cylinder bands” in rotogravure printing. When analyzing the file, `ptype` fails to correctly identify the type for some columns of this dataset.

For example, the “ESA Amperage” column contains mostly the 0 value (480 out of 540 entries) and a small number of other values (0.5, 4, 6, ?). The initial type offered by `ptype` is Boolean with 0.5, 4 and 6 incorrectly treated as

1. From: <https://www.kaggle.com/orgesleka/imdbmovies>.
2. Available at: <https://github.com/victordiaz/color-art-bits->

TABLE 2
Interactions required to solve a wrangling task for each AI assistant.

AI assistant (dataset)	Number of interactions					Average
	0	1	2	3	4+	
datadiff (UCI)	0.52	0.20	0.12	0.00	0.18	3.25
datadiff (without Iris)	0.63	0.22	0.15	0.00	0.00	1.40
CleverCSV (GitHub)	0.20	0.70	0.05	0.04	0.00	1.17
ptype (Various)	0.33	0.51	0.16	0.00	0.00	1.24

anomalies and ? correctly identified as missing data. This is perhaps unsurprising given the dominance of 0 values.

The analyst can obtain the correct type through a single interaction, by choosing “ESA Amperage is not Boolean”, which adds the `not_type(Boolean)` constraint. The assistant then returns the correct, second most likely, data type `Float` with no anomalies and ? as the missing data indicator.

State of the art tools face similar issues. Trifacta labels the “ESA Amperage” column with the integer type rather than float. It considers 4 and 6 valid values, but 0.5 and ? are treated as *mismatched values*. The analyst needs to change the assigned type to float through the user interface, by clicking on the integer sign and then selecting the float type. This interaction is specific to type inference in Trifacta and requires familiarity with the graphical user interface.

OpenRefine does not directly address column-type inference. Instead, it separately infers the type for each entry. It correctly identifies the data type for the entries of “ESA Amperage” as it uses the numeric label rather than separate float and integer types. However, user interaction is required for many other columns in the same dataset that are labeled correctly by both `ptype` and Trifacta. For example, the “grain screened” column represents a Boolean with values `yes` and `no`. Here, `ptype` and Trifacta correctly infer the type as `Boolean`, whereas OpenRefine treats the values as text. Changing the assigned type to `Boolean` converts both `yes` and `no` to `false`. To get the correct types, the analyst first needs to replace all values of `yes` with `true`.

5.2 Empirical evaluation

For optimization-based AI assistants, we can evaluate how many interactions are needed to arrive at the correct result. As each assistant solves a different task, we need to use a different dataset for each. Table 2 shows the results; for each AI assistant, we show the fraction of cases that requires a specific number of interactions. The “Average” column shows the average over the cases where *some* human interaction is required. The datasets used are discussed below.

datadiff. Following the original *datadiff* evaluation [6] we use a synthetic dataset obtained by corrupting five datasets from the UCI repository [36] (Abalone, Adult, Bank, Car, Iris). To corrupt a file, we randomly reorder columns and apply two other randomly chosen corruptions.

The corruptions include inserting a numerical column (with values from a uniform distribution $U(0, 1)$), inserting a categorical column (with two evenly distributed values), deleting a random column, recoding a categorical column and applying a linear transformation (with a from $U(-0.5, 0.5)$ and b from $U(-2\bar{v}, 2\bar{v})$ where \bar{v} is the mean of

the values in the column). We apply the corruption to a randomly selected half of the data and attempt to reconcile the two halves using `datadiff`. When `datadiff` does not produce the expected result, we repeatedly add `nomatch` constraints to prevent incorrect matchings inferred by `datadiff`.

Our corruptions and dataset are more challenging than those used previously [6]. We note that `datadiff` performs poorly on one of the five datasets (Iris), so the table shows results for all five datasets as well as for the remaining four (without Iris). `Datadiff` requires no human interaction in 52% and 63% cases, respectively. Our evaluation models the case where the analyst can easily spot an error and inform the assistant, but the number of interactions could be reduced further by choice of the explicit match constraint.

CleverCSV. To evaluate `CleverCSV`, we revisit the failure cases of the non-interactive `CleverCSV` [5] and count interactions needed to find the correct dialect. We apply the assistant on 255 files from a corpus of CSV files extracted from GitHub where the dialect was detected incorrectly in [5]. We focus on this selection as the 97% of cases where `CleverCSV` detects the dialect correctly are not relevant here. Since the dialect considered for the CSV file consists of three components, the maximum number of interactions is three.

For 20% of the 255 files no interaction is needed to find the correct dialect. This can be attributed to improvements in `CleverCSV` since publication of [5]. For the majority of files (70%) a single interaction was needed, with an average of only 1.17. This illustrates that as the human provides a constraint to the AI assistant, the limits on the search space allow `CleverCSV` to quickly arrive at the correct answer.

ptype. To evaluate the `ptype` AI assistant, we consider 43 (out of 610) data columns where the types were not inferred correctly by the non-interactive `ptype` [30], using a corpus obtained from various sources including the UCI repository [36] and open government data sources. To guide the assistant, we iteratively add the `not_type` constraint.

Although `ptype` recognizes 11 data types, we focus on 5 primitive types (Boolean, integer, floating-point number, date, string) to allow comparison with other tools. Even with 5 types, identifying them correctly by hand remains difficult, because `ptype` also detects anomalous and missing values, which may not be easy to notice for a human analyst.

Of the 43 data columns, no interaction is needed for 33% of cases. As with `CleverCSV`, this is due to recent improvements in `ptype`. A single interaction was needed for a majority of files (51%). In those cases, the assistant arrives at the correct answer by choosing the second most likely type. The remaining columns require two interactions, resulting in an average of 1.24. Note that rejecting the offered type is a simpler interaction than directly selecting a type, which would reduce the maximum number of interactions to one.

Summary. The quantitative evaluation of three AI assistants demonstrates that many data wrangling tasks can be solved much more efficiently by creating opportunities where the human analyst can nudge the tool in the right direction. This approach obviates the need for tedious data manipulation in spreadsheet applications or case-specific wrangling scripts, and is significantly easier to implement than fully automatic tools that need to cover numerous edge cases.

6 RELATED WORK

The problem of data wrangling has been studied by both practitioners [1], [2], [37] and academics [12], [38], [39], [40]. These studies repeatedly mention the problems that motivated our work. We believe that *interactivity*, and *uniformity* are crucial. Interactivity allows incorporating crucial human insights, and a common structure makes it possible for the analyst to easily access a wide range of tools.

Programming and analytic systems. Data wrangling is often done programmatically in the R and Python languages, using libraries such as `Tidyverse` and `Pandas` [7], [8] in notebook systems like `RStudio` and `Jupyter`. Our AI assistants are available for the `Jupyter` platform through the `Wrattler` extension [41], which enables `polyglot` programming.

Spreadsheets and business intelligence tools such as `Tableau` and `Power BI` provide a complex set of features for data analytics, often used through a complex graphical user interface, while more focused data wrangling tools like `Trifacta` and `OpenRefine` [9], [10] provide similar environments focused on data cleaning. As discussed in Section 5, those tools address many of the specific problems addressed by AI assistants, but lack uniformity and rarely implement the powerful *iterative* interaction model.

Data wrangling and repair tools. A number of tools attempt to solve a specific data wrangling problem automatically, including the tools extended in this paper [5], [6], [30], [31] as well as tools for data imputation [42], deduplication [16], and parsing [15]. These tools often achieve a high accuracy, but they lack an easy-to-use mechanism for incorporating critical human insights in cases where the automatic answer is incorrect. Automatic tools can also be guided by a manually written domain-specific data model, as in `PClean` [43].

A few systems utilize the flexible *iterative* interaction pattern to suggest possible data transformations using machine learning. `Proactive wrangling` [24] suggests data transformations to improve data structure based on a metric and offers those to the user. In `Predictive Interaction` [18], inputs provided by the analyst are used to generate a ranked list of predictions from which the analyst can choose or, alternatively, provide further inputs. This is similar to how AI assistants work, but the way of specifying feedback is domain-specific, e.g., highlighting substrings in textual data.

The problem of incorporating user input has been extensively studied in data repair tools for databases [44], [45]. Tools for enforcing functional database dependencies [19], [20], [46] work by asking analysts questions about the data and using the answers to improve the model used for data repair. Such tools could be recast as AI assistants; they complement our examples in that they focus on working with databases whereas our focus is on less structured data.

Programming language approaches. Numerous programmatic tools offer a small domain-specific language for a particular data wrangling task such as statistical analysis or data visualization [47], [48]. A small domain-specific language is also at the core of semi-automatic tools such as `LearnPADS` and `Predictive Interaction` [15], [18]. AI assistants follow the same approach in that the expressions of individual AI assistants form small languages that are easy to understand.

AI assistants can also be seen as a form of code completion. This typically focuses on offering available operations, possibly using machine learning to rank the recommendations [49]. Type providers [50], [51] are closer to our approach in that the recommendations are generated programmatically, similar to our *choices* operation.

Human-computer interaction. Two interaction techniques used in data wrangling tools are direct manipulation and programming-by-example. In the former, a program is specified by directly interacting with the output. This has been used for data analysis and querying [52], [53], [54], [55], as well as data wrangling [45]. In the latter, the user gives examples of desired results, for example, to specify data transformations in spreadsheets [14]. This results in an *iterative* interaction mechanism, but one where the analyst needs to specify more complex inputs as opposed to just choosing from a list of options. Novel human-interaction techniques for data wrangling also include touch-based editing [56], natural language [57], and conversational agents [58].

Human in the loop. Our work contributes to the emerging field of human-in-the-loop data analytics [23], [59]. AI assistants particularly implement the “efficient correction” pattern [60]. We focus on supporting an individual analyst, but a range of systems involve multiple users in addressing data wrangling problems. In [61], data cleaning problems are solved by assigning the tasks that cannot be automated to human “detectors” and “repairers” and several data cleaning tools rely on crowdsourcing [62], [63], [64].

7 FUTURE WORK

Allowing AI assistants to accept richer user inputs would let us support programming-by-example. Programming-by-example can be seen as ranking programs in an underlying DSL that are consistent with a given set of training examples [65], fitting well with our optimization-based AI assistant structure. Alternatively, focusing on probabilistic AI assistants would let the system leverage additional information, such as the distribution of possible cleaning scripts, allowing users to choose a desired solution more effectively. The usability of AI assistants could also be improved by offering possible choices in a more structured way than as a flat list.

The AI assistants presented in this paper solve individual data wrangling problems, but a typical data wrangling workflow involves a combination of tools. An interesting direction for future work is to recommend the entire data wrangling workflow, composed of multiple AI assistant invocations. This could be done by repeatedly predicting the next step as in [18], [24]. Closer interaction between AI assistants could also lead to better results. For example, the consistency measure used by CleverCSV could incorporate information obtained from *p*type or ColNet, while *datadiff* could prefer matching columns with the same data type, as inferred by *p*type or ColNet.

Finally, AI assistants do not currently learn from past user interaction. Using the interactions with human analysts to improve the models underlying the AI assistants as well as learning the ways in which AI assistants are composed could provide valuable information for improving the accuracy of the inference done by the assistants.

8 CONCLUSION

Data wrangling is notoriously tedious and hard to automate. It has eluded the recent rise of AI because large datasets hide corner cases that require human insight. We have introduced the notion of AI assistants, which captures the structure of semi-automatic, interactive tools for data wrangling. We showed how the definition captures common types of tools and makes them easy to use from notebook systems. We developed four concrete AI assistants that are flexible interactive versions of existing non-interactive tools.

This paper makes two claims. First, we argue that the structure of AI assistants is a suitable abstraction for interactive data wrangling tools. Second, we argue that our interactive AI assistants are more practical than fully automatic tools. To support the first claim, we present AI assistants that cover a wide range of data wrangling tasks including parsing, merging mismatched datasets, type inference, and the inference of semantic information. To support our second claim, we discuss three real-world case studies where a fully automatic tool does not give the desired result, together with an empirical evaluation that showed that users can typically solve a wrangling task with 1-3 simple interactions.

While we cannot hope to reduce to zero the 80% of the time that data analysts spend on data wrangling solely with what we have described above, we believe that our framework provides the right pathway. A growing ecosystem of interactive unified AI assistants would allow data analysts to fully leverage recent AI advances for the most tedious and time-consuming aspect of their job and pave the way for more equitable access to data science and machine learning.

ACKNOWLEDGMENTS

The work was supported in part by EPSRC grant EP/N510129/1 to The Alan Turing Institute. TP and CW would like to acknowledge the funding provided by the UK Government’s Defence & Security Programme in support of The Alan Turing Institute. The work of EJR was also supported in part by the SIRIUS Centre for Scalable Data Access (Research Council of Norway, project 237889) and TC was supported by a PhD studentship from The Alan Turing Institute (EPSRC grant TU/C/000018).

The work on AI assistants would not be possible without work on Wrattler by May Yong and Nick Barlow. We benefited from discussions with colleagues in The Alan Turing Institute, especially Charles Sutton and James Geddes. We would also like to highlight the contribution of Jiaoyan Chen, the researcher behind the non-interactive ColNet. Last but not least, we thank the anonymous reviewers for their comments that have helped improve the paper.

REFERENCES

- [1] Crowdfunder, “Data science report,” 2016, accessed November 2018. [Online]. Available: <http://visit.figure-eight.com/data-science-report.html>
- [2] Kaggle, “The state of data science & machine learning,” 2017, accessed September 2018. [Online]. Available: <http://kaggle.com/surveys/2017>
- [3] X. He, K. Zhao, and X. Chu, “AutoML: A survey of the state-of-the-art,” *Knowledge-Based Systems*, vol. 212, p. 106622, 2021.
- [4] T. De Bie, L. De Raedt, J. Hernández-Orallo, H. H. Hoos, P. Smyth, and C. K. I. Williams, “Automating data science: Prospects and challenges,” *Commun. ACM*, vol. 65, no. 3, p. 76–87, 2022.

- [5] G. J. J. Van den Burg, A. Nazábal, and C. Sutton, "Wrangling messy CSV files by detecting row and type patterns," *Data Mining and Knowledge Discovery*, vol. 33, no. 6, pp. 1799–1820, 2019.
- [6] C. A. Sutton, T. Hobson, J. Geddes, and R. Caruana, "Data Diff: Interpretable, executable summaries of changes in distributions for data wrangling," in *24th ACM SIGKDD Conference*, 2018.
- [7] W. McKinney, "pandas: a foundational Python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, 2011.
- [8] H. Wickham, M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester *et al.*, "Welcome to the Tidyverse," *Journal of Open Source Software*, vol. 4, no. 43, p. 1686, 2019.
- [9] Trifacta, "Trifacta – data wrangling software and tools," 2021, accessed July 2022. [Online]. Available: <https://www.trifacta.com>
- [10] A. Delpuch, D. Huynh, T. Morris, S. Mazzocchi, and contributors, "OpenRefine: A free, open source, powerful tool for working with messy data," 2021. [Online]. Available: <https://openrefine.org/>
- [11] R. Wesley, M. Eldridge, and P. T. Terlecki, "An analytic data engine for visualization in Tableau," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11, T. Sellis, R. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds. ACM, 2011, pp. 1185–1194.
- [12] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. Van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono, "Research directions in data wrangling: Visualizations and transformations for usable and credible data," *Information Visualization*, vol. 10, no. 4, pp. 271–288, 2011.
- [13] M. Aristarán, "Tabula," 2021. [Online]. Available: <https://github.com/tabulapdf/tabula>
- [14] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.
- [15] K. Fisher, D. Walker, and K. Q. Zhu, "LearnPADS: automatic tool generation from ad hoc data," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, 2008, pp. 1299–1302.
- [16] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 1, pp. 1–16, 2007.
- [17] T. Petricek, "Foundations of a live data exploration environment," *Art Sci. Eng. Program.*, vol. 4, no. 3, p. 8, 2020.
- [18] J. Heer, J. M. Hellerstein, and S. Kandel, "Predictive interaction for data transformation," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2015.
- [19] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas, "Guided data repair," *Proceedings of the VLDB Endowment*, vol. 4, no. 5, pp. 279–289, 2011.
- [20] S. Thirumuruganathan, L. Berti-Equille, M. Ouzzani, J.-A. Quiane-Ruiz, and N. Tang, "UGuide: User-guided discovery of FD-detectable errors," in *Proceedings of the ACM International Conference on Management of Data*, ser. SIGMOD '17, 2017, pp. 1385–1397.
- [21] E. Horvitz, "Principles of mixed-initiative user interfaces," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1999, pp. 159–166.
- [22] J. Williams, C. Negreanu, A. D. Gordon, and A. Sarkar, "Understanding and inferring units in spreadsheets," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2020, pp. 1–9.
- [23] A. Doan, "Human-in-the-loop data analysis: A personal perspective," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '18. ACM, 2018.
- [24] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, "Proactive wrangling: mixed-initiative end-user programming of data transformation scripts," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011, pp. 65–74.
- [25] Ofcom, "Open data," 2018. [Online]. Available: <https://www.ofcom.org.uk/research-and-data/data/pendata>
- [26] R. Bird and O. de Moor, *The Algebra of Programming*. Prentice-Hall, 1996.
- [27] O. Bachem, M. Lucic, and A. Krause, "Practical coresets constructions for machine learning," *arXiv preprint arXiv:1703.06476*, 2017.
- [28] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, 1955.
- [29] Y. Shafraanovich, "Common format and MIME type for comma-separated values (CSV) files," Internet Requests for Comments, Tech. Rep. RFC 4180, 2005.
- [30] T. Ceritli, C. K. I. Williams, and J. Geddes, "ptype: Probabilistic type inference," *Data Mining and Knowledge Discovery*, vol. 34, no. 3, pp. 870–904, 2020.
- [31] J. Chen, E. Jiménez-Ruiz, I. Horrocks, and C. Sutton, "ColNet: Embedding the semantics of web tables for column type prediction," in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 29–36.
- [32] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. d. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier *et al.*, "Knowledge graphs," *Synthesis Lectures on Data, Semantics, and Knowledge*, vol. 12, no. 2, pp. 1–257, 2021.
- [33] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer *et al.*, "DBpedia—a large-scale, multilingual knowledge base extracted from Wikipedia," *Semantic web*, vol. 6, no. 2, pp. 167–195, 2015.
- [34] E. Jiménez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, and K. Srinivas, "SemTab 2019: Resources to Benchmark Tabular Data to Knowledge Graph Matching Systems," in *The Semantic Web - 17th International Conference (ESWC)*, 2020, pp. 514–530.
- [35] A. Nazábal, C. K. I. Williams, G. Colavizza, C. R. Smith, and A. Williams, "Data engineering for data analytics: A classification of the issues, and case studies," *CoRR*, vol. abs/2004.12929, 2020. [Online]. Available: <https://arxiv.org/abs/2004.12929>
- [36] K. Bache and M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: <archive.ics.uci.edu/ml>
- [37] T. Dasu and T. Johnson, *Exploratory Data Mining and Data Cleaning*, 1st ed. John Wiley & Sons, Inc., 2003.
- [38] S. Krishnan, D. Haas, M. J. Franklin, and E. Wu, "Towards reliable interactive data cleaning: a user survey and recommendations," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '16. ACM, 2016.
- [39] M. I. Gorinova, K. Prince, S. Meakins, A. Vuylsteke, M. Jones, and A. F. Blackwell, "The end-user programming challenge of data wrangling," in *Proceedings of the 27th annual workshop of the Psychology of Programming Interest Group (PPIG)*, 2016, pp. 140–149.
- [40] A. Paleyes, R.-G. Urma, and N. D. Lawrence, "Challenges in deploying machine learning: a survey of case studies," *ACM Computing Surveys*, 2020.
- [41] T. Petricek, J. Geddes, and C. A. Sutton, "Wrattler: Reproducible, live and polyglot notebooks," in *10th USENIX Workshop on Theory and Practice of Provenance (TaPP 2018)*, 2018.
- [42] A. Nazábal, P. M. Olmos, Z. Ghahramani, and I. Valera, "Handling incomplete heterogeneous data using VAEs," *Pattern Recognition*, vol. 107, p. 107501, 2020.
- [43] A. Lew, M. Agrawal, D. Sontag, and V. Mansinghka, "PClean: Bayesian data cleaning at scale with domain-specific probabilistic programming," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2021, pp. 1927–1935.
- [44] A. Assadi, T. Milo, and S. Novgorodov, "DANCE: data cleaning with constraints and experts," in *IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 1409–1410.
- [45] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *Proc. of the 27th Intl. Conference on Very Large Data Bases*. Morgan Kaufmann, 2001, pp. 381–390.
- [46] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for capturing data inconsistencies," *ACM Transactions on Database Systems*, vol. 33, no. 2, pp. 1–48, 2008.
- [47] E. Jun, M. Daum, J. Roesch, S. Chasins, E. Berger, R. Just, and K. Reinecke, "Tea: A high-level language and runtime system for automating statistical analysis," in *Proceedings of the 32nd Annual ACM UIST Symposium 2019*, F. Guimbretière, M. Bernstein, and K. Reinecke, Eds. ACM, 2019, pp. 591–603.
- [48] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE Tran. on Vis. and Comp. Graphics*, vol. 23, no. 1, pp. 341–350, 2016.
- [49] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM International Symposium on Foundations of Software Engineering*. ACM, 2009, pp. 213–222.
- [50] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek, "Themes in information-rich functional programming for internet-scale data sources," in *Proceedings of Workshop on Data Driven Functional Programming*, ser. DDFP '13. ACM, 2013, pp. 1–4.
- [51] T. Petricek, "Data exploration through dot-driven development," in *31st European Conference on Object-Oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

- [52] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, "Interactive data analysis: the Control project," *Computer*, vol. 32, no. 8, pp. 51–59, 1999.
- [53] B. Shneiderman, C. Williamson, and C. Ahlberg, "Dynamic queries: Database searching by direct manipulation," in *Conference on Human Factors in Computing Systems, CHI '92*, P. Bauersfeld, J. Bennett, and G. Lynch, Eds. ACM, 1992, pp. 669–670.
- [54] M. Derthick, J. Kolojechick, and S. F. Roth, "An interactive visual query environment for exploring data," in *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology, UIST '97*, G. G. Robertson and C. Schmandt, Eds. ACM, 1997, pp. 189–198.
- [55] A. Abouzied, J. M. Hellerstein, and A. Silberschatz, "Dataplay: interactive tweaking and example-driven correction of graphical database queries," in *The 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12*, R. Miller, H. Benko, and C. Latulipe, Eds. ACM, 2012, pp. 207–218.
- [56] A. Crotty, A. Galakatos, E. Zraggen, C. Binnig, and T. Kraska, "Vizdom: Interactive analytics through pen and touch," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 2024–2027, 2015.
- [57] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang, "Eviza: A natural language interface for visual analysis," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST '16*, J. Rekimoto, T. Igarashi, J. O. Wobbrock, and D. Avrahami, Eds. ACM, 2016, pp. 365–377.
- [58] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein, "Iris: A conversational agent for complex tasks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*. ACM, 2018, p. 473.
- [59] G. Li, "Human-in-the-loop data integration," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 2006–2017, 2017.
- [60] S. Amershi, D. S. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. T. Iqbal, P. N. Bennett, K. Inkpen, J. Teevan, R. Kikin-Gil, and E. Horvitz, "Guidelines for Human-AI interaction," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, S. A. Brewster, G. Fitzpatrick, A. L. Cox, and V. Kostakos, Eds. ACM, 2019, p. 3.
- [61] E. K. Rezig, M. Ouzzani, A. K. Elmagarmid, W. G. Aref, and M. Stonebraker, "Towards an end-to-end human-centric data cleaning framework," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '19. ACM, 2019.
- [62] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang, "NADEEF: a commodity data cleaning system," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 541–552.
- [63] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye, "Katara: A data cleaning system powered by knowledge bases and crowdsourcing," in *Proc. of the 2015 ACM SIGMOD Intl. Conference on Management of Data*, 2015, pp. 1247–1261.
- [64] J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "Crowder: Crowdsourcing entity resolution," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1483–1494, 2012.
- [65] S. Gulwani, "Programming by examples (and its applications in data wrangling)," in *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.



heterogeneous data preprocessing and recommender systems.

Alfredo Nazabal is an applied scientist working in the Amazon Development Center in Edinburgh. Before that he was a Postdoctoral researcher in The Alan Turing Institute in London where he worked developing deep generative models for data analytics problems. He obtained his PhD in the University Carlos III of Madrid, where he developed and applied Machine Learning algorithms for Human Activity Recognition. His main research interests include deep generative models, unsupervised learning,



Gerrit J.J. van den Burg is an applied scientist at Amazon. Previously, he was a postdoctoral researcher at The Alan Turing Institute in London, UK, where he worked on automating the manual parts of data science using machine learning. He obtained a PhD from the Erasmus University Rotterdam in The Netherlands, during which he focused on developing algorithms for multiclass classification and sparse regularization.



Taha Ceritli is a postdoctoral research assistant at the University of Oxford. His current research focus is machine learning for healthcare. He previously received his PhD from the University of Edinburgh on probabilistic data type inference, which was carried out at and supported by The Alan Turing Institute.



Ernesto Jiménez-Ruiz is a Lecturer in Artificial Intelligence at City, University of London (UK), and a researcher at the University of Oslo (Norway). He previously held a Senior Research Associate position at The Alan Turing Institute in London (UK) and a Research Assistant position at the University of Oxford (UK). His research interests focus on the application of Semantic Web technology to Data Science workflows and the combination of Knowledge Representation and Machine Learning techniques.



Tomas Petricek is a Lecturer in School of Computing at University of Kent, UK. His research focuses on making programming easier, trustworthy and more accessible. Previously, he worked on tools for data science in The Alan Turing Institute and functional programming language F# in Microsoft Research. He holds PhD from University of Cambridge where he developed theory of context-aware programming languages.



Chris Williams is Professor of Machine Learning in the School of Informatics, University of Edinburgh. His main areas of research are in visual object recognition and image understanding, models for understanding time-series, AI for data analytics, unsupervised learning, and Gaussian processes. He obtained his MSc (1990) and PhD (1994) at the University of Toronto, under the supervision of Geoffrey Hinton. He was at Aston University 1994-1998, and has been at the University of Edinburgh since 1998.

APPENDIX A FORMAL DEFINITIONS

This appendix provides further details of the formal models of the AI assistants discussed in the main text. The complete descriptions provided here enhance the reproducibility of our work and make it possible to reimplement AI assistant as presented and evaluated in this paper. A summary of symbols used in the formalization can be found in Table 5.

datadiff. The definition of the datadiff AI assistant given in Section 4.1 shows patches, constraints, and the $best_X$ operation. It omits the $choices_X$ operation and the $valid$ predicate which identifies patches that are valid for a given set of human interactions H . The operations of the datadiff assistant, including the $valid$ predicate, are defined as:

$$best_X(H) = \arg \max_{e \in E_H} Q(X, e) \text{ where} \\ E_H = \{(P_1, \dots, P_k) \in E \mid \forall i \in 1 \dots k. \text{valid}_H(P_i)\}$$

$\text{valid}_H(P)$ such that

$\text{valid}_H(\text{permute}(\pi))$ iff

$$(\forall \text{match}(i, j) \in H. (i, j) \in \pi) \wedge \\ (\forall \text{nomatch}(i, j) \in H. (i, j) \notin \pi)$$

$\text{valid}_H(\text{recode}(i, [\dots]))$ iff $\text{nottransform}(i) \notin H$

$\text{valid}_H(\text{linear}(i, a, b))$ iff $\text{nottransform}(i) \notin H$

$\text{valid}_H(\text{delete}(i))$

$\text{valid}_H(\text{insert}(i, d))$

$choices_X(H) =$

$$\{H \cup \{\text{nottransform}(i)\} \mid \forall i. \text{recode}(i, [\dots]) \in e\} \cup \\ \{H \cup \{\text{nottransform}(i)\} \mid \forall i. \text{linear}(i, a, b) \in e\} \cup \\ \{H \cup \{\text{nomatch}(i, j)\} \mid \text{permute}(\pi) \in e, \forall i, j. (i, j) \in \pi\} \cup \\ \{H \cup \{\text{match}(i, j)\} \mid \text{permute}(\pi) \in e, \forall i, j. (i, j) \notin \pi\} \\ \text{where } e = best_X(H)$$

ptype. In the ptype AI assistant, $best_X$ is obtained by taking the maximum a posteriori of the posterior probability distribution of the set of expressions determined by the past human interactions. As in the case of datadiff and CleverCSV, this is defined using the $valid$ predicate. The $choices_X$ operation allows the analyst to reject an inferred type and mark an inferred missing or anomalous value as non-missing or non-anomalous. More formally, the operations of ptype are defined as follows:

$$best_X(H) = \arg \max_{e \in E_H} p_H(X|e) p_H(e) \text{ where} \\ p_H(X|e) = p(X|e) \text{ and } p_H(e) = p(e) \\ E_H = \{e \in E \mid \text{valid}_H(e)\}$$

$\text{valid}_H(\tau, V_m, V_a)$ iff

$$(\text{not_type}(\tau') \in H \implies \tau' \neq \tau) \vee \\ (\text{not_missing}(v) \in H \implies v \notin V_m) \vee \\ (\text{not_anomaly}(v) \in H \implies v \notin V_a)$$

$$choices_X(H) = \{H \cup \{\text{not_type}(t)\}\} \cup \\ \{H \cup \{\text{not_missing}(v_j)\} \mid j \in J\} \cup \\ \{H \cup \{\text{not_anomaly}(w_k)\} \mid k \in K\}$$

where $best_X(H) =$

$$(\text{type}(t), \text{missing}\{v_j\}_{j \in J}, \text{anomaly}\{w_k\}_{k \in K})$$

CleverCSV. The definition of the CleverCSV AI assistant closely follows the example of datadiff. The $best_X$ operation uses the pattern of the optimization-based AI assistants. The $choices_X$ operation allows the analyst to reject a component of a currently inferred dialect or to explicitly choose a specific character for a dialect component. As before, the $valid$ predicate determines what is a valid dialect given past human interactions. Formally:

$$best_X(H) = \arg \max_{e \in E_H} Q(X, e) \text{ where} \\ E_H = \{e \in E \mid \text{valid}_H(e)\}$$

$\text{valid}_H(\text{is_delimiter}(d), \text{is_quote}(q), \text{is_escape}(a))$ iff

$$(\text{fix_delimiter}(d') \in H \implies d' = d) \vee \\ (\text{fix_quote}(q') \in H \implies q' = q) \vee \\ (\text{fix_escape}(a') \in H \implies a' = a) \vee \\ (\text{block_delimiter}(d') \in H \implies d' \neq d) \vee \\ (\text{block_quote}(q') \in H \implies q' \neq q) \vee \\ (\text{block_escape}(a') \in H \implies a' \neq a)$$

$choices_X(H) =$

$$\{H \cup \{\text{not_delimiter}(c_d)\}, H \cup \{\text{not_quote}(c_q)\}, \\ H \cup \{\text{not_escape}(c_e)\}\} \cup \\ \{H \cup \{\text{delimiter}(c)\} \mid \forall c. C\} \cup \\ \{H \cup \{\text{quote}(c)\} \mid \forall c. C\} \cup \{H \cup \{\text{escape}(c)\} \mid \forall c. C\} \\ \text{where } (\text{delimiter}(c_d), \text{quote}(c_q), \text{escape}(c_e)) = best_X(H).$$

ColNet. The definition of ColNet differs from the other examples in that human interactions affect the objective function $Q_H(X, e)$ rather than the set of possible expressions E_H . The definition of the objective function is discussed in Section 4.4. The following provides a full definition of both of the operations of the AI assistant for completeness:

$$c = \text{not_type}(S, \sigma) \mid \text{is_type}(S, \sigma)$$

$$p_S^\sigma = \text{As defined in non-interactive ColNet}$$

$$q_{H,S}^\sigma = \begin{cases} 1 & \text{when } \text{is_type}(S, \sigma) \in H \\ 0 & \text{when } \text{not_type}(S, \sigma) \in H \\ p_S^\sigma & \text{otherwise} \end{cases}$$

$$q_{H,S}^\sigma = \frac{1}{|S|} \sum_{S \in S} q_{H,S}^\sigma$$

$$best_X(H) = \arg \max_{\sigma \in G} Q_H(X, \sigma) \text{ where}$$

$$Q_H(X, \sigma) = q_{H,S}^\sigma$$

$choices_X(H) =$

$$\{\text{is_type}(S, \sigma), \text{not_type}(S, \sigma) \mid S \in S, \sigma \in G, p_S^\sigma \geq \epsilon\}$$

APPENDIX B PERFORMANCE CONSIDERATIONS

In general, interactive AI assistants obtained by adapting an existing non-interactive tool (datadiff, ptype, CleverCSV, ColNet) invoke the optimization logic of the non-interactive tool, with small modifications, each time the user interacts with the assistant. The performance is thus comparable to the performance of the base tools [5], [6], [30], [31]. In some cases, however, the interactive AI assistant can reuse previously computed results and perform more efficiently.

In this section, we briefly discuss performance in typical real-world scenarios as well as algorithmic complexity of the optimization and possible performance improvements for the four AI assistants discussed in the paper.

datadiff. The datadiff AI assistant uses the algorithm from non-interactive datadiff [6]. This works in two phases. In the first phase, the algorithm determines a cost matrix C_{ij} by finding the best patch between each pair of columns. In the second phase, the algorithm uses the Hungarian algorithm to find the best bipartite matching. The interactive AI assistant requires two modifications. First, after computing C_{ij} , we set C_{ij} to 0 for each $\text{match}(i, j)$ constraint and to $+\infty$ for each $\text{nomatch}(i, j)$ constraint. Second, we modify the logic for finding the best patch to not use a recoding or linear transformation when `norecode` is specified.

The algorithmic complexity of the first phase is $O(n^2)$ in terms of the number of columns n , while the algorithmic complexity of the second phase is $O(n^3)$. For real-world datasets, however, most of the time is spent in the first phase, generating and evaluating possible pairwise patches. Reconciling the full broadband quality dataset for 2014 (31 columns) and 2015 (71 columns) takes 35 seconds on a recent computer.³ Reconciling the full 2014 (31 columns) with filtered 2015 (6 columns) dataset, as done in the case study in Section 5.1, takes 5 seconds.

This makes the current implementation useable for smaller datasets. There are two ways in which the performance could easily be improved. First, the cost matrix (phase one) could be determined on the first run and then cached. This does not change between runs and would significantly improve the performance on subsequent interactions. Second, the cost matrix could be determined based on a sample of the full dataset, possibly improving the initial cost matrix in background after offering the first recommendation.

pctype. The pctype AI assistant computes the posterior probabilities over data types based on non-interactive pctype [30] and updates the initially assigned types according to the user feedback. This is achieved by storing the posterior probability distributions rather than re-running non-interactive pctype. Thus, assuming that the number of known column types and therefore the maximum number of user corrections is constant, the complexity of the pctype AI assistant becomes identical to the complexity of non-interactive pctype.

The computational bottleneck in type inference via pctype is the calculation of probability distribution assigned for a data column x by the k th PFSM denoted by $p(x|t = k)$. This calculation is carried out by taking into account only unique data entries, for efficiency. Denoting the u th unique data value by x_u , the computation of $p(x_u|t = k)$ is done via the PFSM Forward algorithm. This has complexity $O(M_k^2 L)$, where M_k is the number of hidden states in the k th PFSM, and L is the maximum length of the data entries. Therefore, the overall complexity of the inference becomes $O(UKM^2L)$, where U is the number of unique data entries,

K is the number of types, and M is the maximum number of hidden states in the PFSMs.

Notice that the complexity depends on data through U and L , and does not necessarily increase with the number of rows. The runtime for non-interactive pctype has been shown to scale linearly with the number of unique values U , handling around $10K$ unique values per second [30]. This makes the pctype AI assistant feasible in practice. It would be possible to further improve its performance by parallelizing the computations. For instance, the calculation of the probabilities assigned for unique data values can be calculated independently.

CleverCSV. The CleverCSV AI assistant uses the non-interactive algorithm of [5] to identify the optimal formatting dialect for a given CSV file, X . The possible dialects are generated by CleverCSV prior to the optimization, and are based on the set, \mathcal{C} , of unique characters in the file. The optimization proceeds by computing for each dialect a “pattern score” that captures how regular the structure of the parsed data is (i.e., whether the resulting table has the same number of cells in each row), and a “type score” that captures the proportion of cells in the parsed file with an identifiable data type. The product of these two scores forms the objective function to be maximized. Since the type score is in the range $[0, 1]$, computing it can be skipped for dialects with a small value for the pattern score (see [5]).

We distinguish three components of CleverCSV: constructing potential dialects, computing pattern scores, and computing type scores. Constructing the dialects can be done naively in $O(|\mathcal{C}|)$ time, with $|\mathcal{C}|$ denoting the number of elements in the set \mathcal{C} . In [5] two pruning steps are discussed to remove unlikely dialects, which increases the complexity for constructing potential dialects to $O(|X| \cdot |\mathcal{C}|^3)$. Theoretically, the number of potential dialects is on the order of $|\mathcal{C}|^3$. Computing the pattern score for each dialect is linear in the size of the input file, $O(|X|)$. If we write \mathcal{T} for the set of data types in the type score, then computing this score can be done in $O(|X| \cdot |\mathcal{T}|)$, as the number of cells in the parsing result is linear in the size of the input. Combining these results gives a worst-case runtime complexity of $O(|X| \cdot |\mathcal{T}| \cdot |\mathcal{C}|^3)$. However, as discussed in [5], the number of potential dialects is in practice proportional to $|\mathcal{C}|$, giving a practical runtime complexity of $O(|X| \cdot |\mathcal{T}| \cdot |\mathcal{C}|)$. The median runtime for the files in Section 5.2 is 0.018 seconds.

Human interaction with the CleverCSV AI assistant provides constraints on the dialects considered for the file. By storing the value of the objective function for each dialect in a lookup table, interactions with the AI assistant need only update the allowed dialects in this table, resulting in interactions that are linear in the number of dialects.

ColNet. The non-interactive version of ColNet trains a CNN classifier for each (relevant) semantic type in the knowledge graph. The training is split into two phases [31]: pre-training and fine-tuning. The pre-training is performed using the information from the knowledge graph (typically a large set of samples) while the fine-tuning is computed with the data from the column to be annotated (typically a small set of samples).

As described in [31], the classifiers were implemented in

³ Laptop with Intel Core i7-1185G7 processor, 15GB RAM, running inside Docker container on Windows 11 OS.

Tensorflow and the pre-trained phase for each classifier was completed within 2 minutes on a workstation with Xeon CPU E5-2670. The computation time for the fine-tuning phase was in the order of seconds. The interactive version of ColNet relies on the same training phases, where the pre-training can be run offline for each knowledge graph. Fine-tuning only needs to be run before the first human interaction and it is done using a sample drawn from the input data. The sample size can thus be adapted to meet given performance goals. For the cases of very large knowledge graphs, one could also focus only on a subset of relevant types.

The constraints used by ColNet, as described in Section 4.4, directly affect the score associated to a semantic type for the involved sample and has an impact on the overall score of a semantic type for the column. Constraints obtained during the user interaction could also be used to further fine-tune the involved classifiers and thus adapt their scores. This would affect the performance, but could potentially improve the quality of recommendations.

APPENDIX C OUTLIER AI ASSISTANT

The AI assistants discussed so far are examples of tools based on sophisticated machine learning methods. Such tools allow the analyst to tackle the most challenging data wrangling tasks. However, data analysts also regularly need to complete more mundane tasks, such as identifying outlier values based on standard deviation, removing exact duplicates or correcting simple typos. Such mundane tasks would typically be done without dedicated tool support. However, the fact that AI assistants are very easy to build makes it practical to develop dedicated interactive tools to support mundane tasks that are based on a simple algorithms.

Formal definition

We first discuss a simplified formal model of an AI assistant for removing outlier values based on the m -sigma rule. Given a sequence of values x_1, \dots, x_n with a mean \bar{x} and a standard deviation σ , the assistant identifies values outside of the interval $(\bar{x} - m\sigma, \bar{x} + m\sigma)$ for a multiplier m specified by the analyst. It then offers the values outside of the range to the analyst who can choose which of those should be removed from the dataset. For simplicity, we describe a version of the assistant where the input is a sequence of values, corresponding to a data table with a single column.

The outlier assistant is not optimization-based. It offers potential outliers as a result of the $choices_X$ operation. The user can then choose values to be removed. A human interaction H is thus a set of values selected by the user. The expressions are likewise just sets of values to be removed. The $best_X$ operation does not perform any inference and simply returns the values selected by the user. The f operation then actually removes the values from the dataset. Assuming O is a set of outliers o_1, \dots, o_n such that $o_i \in X$ and $o_i \leq \bar{x} - m\sigma$ or $o_i \geq \bar{x} + m\sigma$, the assistant is defined as:

$$\begin{aligned} f(e, X) &= \{x_i \in X \mid x_i \notin e\} \\ best_X(H) &= H \\ choices_X(H) &= H \cup \{o_1\}, \dots, H \cup \{o_k\} \\ &\text{where } o_1, \dots, o_k = O \setminus H \end{aligned}$$

The expression e is a set of values that the user selected for removal. To apply the expression, the f function removes all values from X that are also in e . Since human interactions H and expressions e are the same, the $best_X$ function simply returns the human interaction H it receives as an argument as the best cleaning script. Finally, the $choices_X$ operation takes previously selected values to be removed H . It generates a list of choices by taking all outlier values that are not already selected, i.e., $O \setminus H$, and adds each to the already selected outliers to be removed.

The value of this example is two-fold. First, it implements a simple yet practical operation that data scientists in the real world actually use. For example, the anomaly detection in the Tundra Traits case study discussed in [35] uses this approach with an 8σ threshold. Second, the example shows the flexibility of our definition. It supports optimization-based AI assistants, but also more manual ones such as the Outlier assistant described here.

Removing aggregates

To illustrate the usefulness of simple AI assistants, we developed a practical version of the AI assistant for outlier detection based on the simple theoretical model presented above. The assistant can be used for removing outlier rows, for example when working with datasets that combine raw and aggregate data. This example illustrates the possibilities of the AI assistant ecosystem. It is a simple assistant that solves a specific problem, but does so very effectively.

The assistant takes a data table with a mix of numerical and categorical columns. It identifies rows that contain numerical outliers (using a simple m -sigma rule) and collects values of categorical columns in those rows. The user can choose any of those as conditions for filtering rows in the dataset. The user can choose to remove all rows where a selected categorical column has a particular value that has been found among the outlier rows.

Consider data on aviation incidents published by Eurostat⁴ (Table 3). Each row shows the number of people injured in accidents that involve an airplane registered in a country specified by `c_regis` that occurred in a country given in the `c_geo` column. However, the dataset also contains aggregate rows. The last row in the sample shows the total number of injuries in the EU, which is obtained as a sum of all the other rows (some not shown). Such aggregate rows are not uncommon in real-world datasets, and can significantly affect an analysis if they are not identified.

To work with the data, the analyst first wants to remove the aggregate rows. When she invokes the AI assistant for outlier detection on the aviation accidents dataset, she gets four recommendations related to the `c_regis` column and three recommendations related to the `c_geo` column. The assistant offers a choice of transformations that remove rows where `c_regis` is EU28, FR, CH or NEASA and rows where `c_geo` is EU28, OTH or FR. With two human interactions, the analyst can choose the desired two filters and remove all aggregates (either of the columns has a value EU28) from the dataset. The other choices are not relevant, but indicate regions that are worthy of further investigation, e.g., France

4. <https://ec.europa.eu/eurostat/web/transport/data/main-tables>

TABLE 3
Subset of Eurostat data on aviation accidents.

c_regis	c_geo	2017	2016	2015	2014
UK	CZ	0	0	0	0
UK	IT	0	0	1	0
UK	SE	0	0	0	0
UK	UK	3	0	2	2
EU28	EU28	18	7	22	31

(with higher than average number of accidents) and planes registered outside of the EU (denoted by $NEASA$).

In R or Python, the analyst could write code to identify rows with values outside of the m -sigma range. She might notice the EU28 value and write code to remove rows where c_geo or c_regis are EU28. This is easy for a seasoned programmer, but our AI assistant allows a non-programmer to solve the problem with two simple interactions.

In Trifacta [9] the analyst can use the data quality bar and histogram (automatically displayed for each column) to locate unusual values in each column data. The “Column Details” window also offers a list of outlier values (identified based on proprietary chosen quantile in each column). Based on the outlier values, the analyst can construct a filter for removing rows, e.g., where the value for the 2017 column is between 15 and 25. However, Trifacta operates on individual columns and so it is not immediately obvious that the outliers represent aggregates with a special value in separate c_regis and c_geo columns.

APPENDIX D

SYSTEM OVERVIEW

AI assistants are available as an extension for the industry standard JupyterLab notebook system. Figure 1 shows the use of the `datadiff` AI assistant for solving the problem discussed in Section 5.1.1.

As discussed in Section 3.1, the fact that AI assistants use a unified interface means that a single extension provides access to a wide range of AI assistants available in a single data analysis environment. Our support for AI assistants utilizes the Wrattler extension [41] for JupyterLab. In this section, we discuss the system architecture and implementation of the abstract interface of AI assistants. The code for the Wrattler extension and several of the AI assistants is available at: <https://github.com/wrattler>.

System architecture. Our implementation leverages Wrattler [41], which extends JupyterLab with a new kind of polyglot notebook that can contain multiple kinds of cells. The Wrattler architecture, including the support for AI assistants, is illustrated in Figure 3. Wrattler separates the notebook (running in a web browser), from language runtimes and a data store (running on a server). Our extension implements a language plugin for Wrattler that defines a new “AI assistant” cell type and facilitates access to individual AI assistants. The new cell type uses a graphical user interface that allows users to choose the assistant they want to invoke, as well as select the input data. When the cell is evaluated, it invokes the AI assistant, previews the results and allows the user to select one of the options generated by the $choices_X$ operation of the AI assistant.

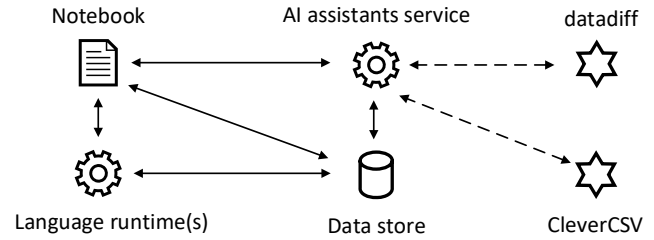


Fig. 3. AI assistants in Wrattler (partly adapted from [41]). Wrattler keeps all data in data store on the server. The notebook communicates with language plugins and data store via HTTP (solid lines). We add a new service that facilitates access to AI assistants, which communicates with individual assistants using standard input/output (dashed lines).

Common interface and integration. Our implementation aims to make it easy to create new AI assistants. For this reason, AI assistants use a shared and easy-to-implement communication interface—standard input and output—together with a simple protocol that implements the abstract definition.

Definition 3.1 defines an AI assistant formally in terms of operations f , $best_X$, and $choices_X$. In our implementation, AI assistants are command-line applications that read commands (corresponding to the operations) from standard input and respond via standard output. For example, our JupyterLab integration calls `datadiff` to get completions after the “Don’t transform LLU” constraint is selected by the analyst as follows (> marks standard input, < marks standard output):

```

1 > reference=/temp/bb15nice.csv,input=/temp/bb14.csv
2 > choices
3 > notransform(LLU)
4
5 < Don't transform 'Urban.rural'
6 < notransform(LLU)/notransform(Urban.rural)
7 < Don't match 'Nation' and 'Urban.rural'
8 < notransform(LLU)/nomatch(Nation,Urban.rural)
9 <

```

The first three lines invoke the $choices_X$ operation by specifying input data (line 1), operation name (line 2) and past human interactions H (line 3). The response generates possible human interactions followed by a blank line. The actual implementation returns multiple choices, but we list only the first two in the above example. Each choice consists of a name, followed by a new human interaction. Here, the human interactions are encoded as constraints, separated by a slash. The analyst previously selected the `notransform(LLU)` constraint, so the two offered human interactions include this and add one other constraint. The first one, named “Don’t transform Urban.rural” (line 5), is represented as two constraints (line 6), the existing `notransform(LLU)` constraint and a newly added `notransform(Urban.rural)` constraint. The second human interaction (lines 7-8) similarly represents two constraints, the existing `notransform(LLU)` constraint and a newly added `nomatch(Nation, Urban.rural)` constraint.

We chose standard input/output as our interface, because it makes it possible to implement AI assistants in any programming language. For example, the assistants presented in this paper have been implemented in R (`datadiff`), Python (`CleverCSV`, `pptype`), and F# (`Outlier`).

TABLE 4
An overview of datasets used throughout the paper and their sources.

Name	Description	Use	Source	Size
Broadband (2014)	UK home broadband performance	datadiff motivation	Ofcom [25]	32 cols, 1971 rows
Broadband (2015)	UK home broadband performance	datadiff motivation	Ofcom [25]	67 cols, 2802 rows
IMDB movies	Classification and rating of 100 movies	CleverCSV evaluation	Kaggle (*)	44 cols, 100 rows
Colors	File names and RGB color codes	CleverCSV scenario	GitHub (†)	11 cols, 300 rows
Cylinder Bands	Cylinder bands in rotogravure printing	pptype scenario	UCI [36]	40 cols, 512 rows
Corrupted UCI (1)	Corrupted (abalone, adult, bank, car, iris)	datadiff evaluation	UCI [36]	max 15 cols, 32,561 rows
Corrupted UCI (2)	Corrupted (abalone, adult, bank, car)	datadiff evaluation	UCI [36]	max 15 cols, 32,561 rows
CleverCSV failures	Subset of data from Gov.uk and GitHub	CleverCSV evaluation	CleverCSV [5]	255 files
pptype failures	Subset of data from Gov.uk and UCI	pptype evaluation	pptype [30]	43 columns
Aviation accidents	EU aviation accidents per year	outlier scenario	Eurostat (‡)	32 cols, 3469 rows

(*) <https://github.com/alan-turing-institute/CleverCSV/blob/master/example/imdb.csv>

(†) <https://github.com/victordiaz/color-art-bits->

(‡) <https://ec.europa.eu/eurostat/web/transport/data/main-tables>

TABLE 5
A glossary of symbols and special identifiers used throughout the paper.

Symbol	Scope	Explanation
e, e^*	AI assistants	Expressions (cleaning scripts) recommended by AI assistants; e^* denotes the best script
X, Y	AI assistants	Input dataset X and output dataset Y
H, H_0	AI assistants	Past human interactions with the AI assistant; H_0 denotes no prior interaction
$f(e, X)$	AI assistants	Operation that applies the expression e (cleaning script) to the input dataset X
$best_X(H)$	AI assistants	Operation that recommends the best expression for a given input, respecting past interactions
$choices_X(H)$	AI assistants	Operation that generates a sequence of options the analyst can choose from
$Q_H(X, e), Q$	Optimization	Objective function that assigns a score to an expression, w.r.t. past interaction (Q_H)
E_H, E	Optimization	Set of permitted expressions; E_H is restricted with respect to past human interaction
$p_H(X e)$	Probabilistic	Likelihood of the input data X given an expression e , w.r.t. past human interaction
$p_H(e)$	Probabilistic	Distribution representing prior beliefs about probabilities of expressions
P	datadiff	A single patch that can be applied to a column of the dataset
c	datadiff, CleverCSV	A single constraint that can be added to H in order to influence the inference
$valid_H$	datadiff, CleverCSV	A predicate that determines if a patch or type respects past interactions (constraints)
τ	pptype	Inferred primitive type such as Boolean, integer, floating-point number, date or string
σ	ColNet	Inferred semantic type from a knowledge graph such as <code>dbo:Company</code>
S	ColNet	Set of sample values, drawn from a column of the input dataset
p_S^σ	ColNet	Score of a sample S for a given semantic type σ in non-interactive mode
$q_{S,H}^\sigma$	ColNet	Score of a sample S for a given semantic type σ ; w.r.t past interactions

Part V

Publications: Data visualization

Chapter 12

Composable data visualisations

Tomas Petricek. 2021. Composable data visualizations. *J. Funct. Program.* 31 (2021), e13.
<https://doi.org/10.1017/S0956796821000046>

FUNCTIONAL PEARLS

Composable data visualizations

TOMAS PETRICEK 

School of Computing, University of Kent, Canterbury CT2 7NZ, UK
(e-mail: t.petricek@kent.ac.uk)

1 Introduction

Let's say we want to create the two charts in Figure 1. The chart on the left is a bar chart that shows two different values for each bar. The chart on the right consists of two line charts that share the x axis with parts of the timeline highlighted using two different colors.

Many libraries can draw bar charts and line charts, but extra features like multiple bars for each label, alignment of multiple charts, or custom color coding can only be used if the library author already thought about your exact scenario. Google Charts (Google, 2020) supports the left chart (it is called Dual-X Bar Chart), but there is no way to add a background or share an axis between charts. The alternative is to use a more low-level library. In D3 (Bostock *et al.*, 2011), you construct the chart piece by piece, but you have to tediously transform your values to coordinates in pixels yourself. For scientific plots, you could use ggplot2 (Wickham, 2016), based on the Grammar of Graphics (Wilkinson, 1999). A chart is a mapping from data to geometric objects (points, bars, and lines) and their visual properties (x and y coordinate, shape, and color). However, the range of charts that can be created using this systematic approach is still somewhat limited.

What would an elegant functional approach to data visualization look like? A functional programmer would want a domain-specific language that has a small number of primitives

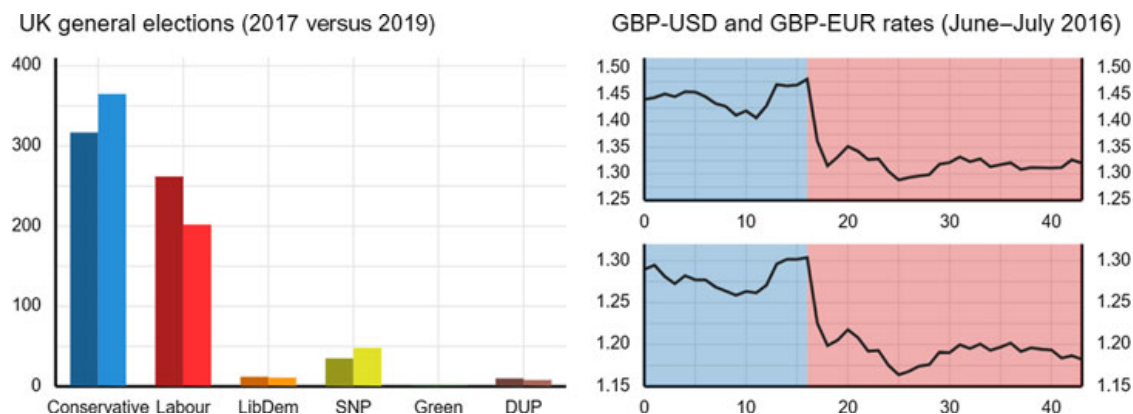


Fig. 1. Two charts about UK politics: comparison of election results from 2017 and 2019 (left) and GBP/USD exchange rate with highlighted areas before and after the 23 June 2016 Brexit vote.

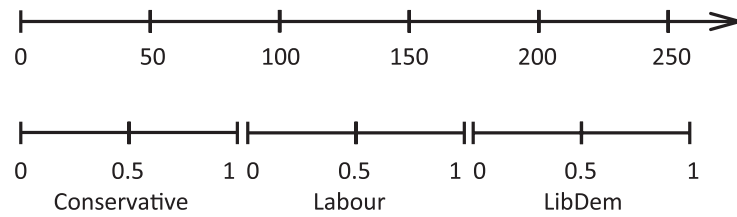


Fig. 2. On a continuous scale (above), an exact position is determined by a number. On a categorical scale (below), an exact position is determined by the category and a number between 0 and 1.

that allow us to define high-level abstractions such as a bar chart and that uses domain values such as the exchange rate, rather than pixels, in its basic building blocks.

As is often the case with domain-specific languages, finding the right primitives is more of an art than science. For this reason, we present our solution, a library named Compost, as a functional pearl. We hope to convince the reader that Compost is elegant and we illustrate this with a wide range of examples. Compost has a number of specific desirable properties:

- Concepts such as bar charts, line charts, or charts with aligned axes are all expressed in terms of more primitive building blocks using a small number of combinators.
- The primitives are specified in domain terms. When drawing a line, the value of a y coordinate is an exchange rate of 1.36 USD/GBP, not 67 pixels from the bottom.
- Common chart types such as bar charts or line charts can be easily captured as high-level abstractions, but many interesting custom charts can be created as well.
- The approach can easily be integrated with the Elm architecture (Czaplicki, 2012) to create web-based charts that involve animations or interaction with the user.

The presentation in this paper focuses on explaining the primitives and combinators of the domain-specific language. We outline the structure of an implementation but omit the details; filling those in merely requires careful thinking about geometry and projections.

Compost is available as open source at <http://compostjs.github.io>. It is implemented in F# but is available as a plain JavaScript library thanks to the Fable F# to JavaScript compiler. The core logic consists of 800 lines of code and depends on the virtual-DOM library (<http://npmjs.com/package/virtual-dom>) for the implementation of the interactive features, making it easily portable to other functional programming languages.

2 Basic charts: Overlaying chart primitives

We introduce individual features of the Compost library gradually. The first important aspect of Compost is that properties of shapes are defined in terms of domain-specific values. In this section, we explain what this means and then use domain-specific values to specify the core part of the UK election results bar chart.

2.1 Domain-specific values

In the election results chart in Figure 1 (left), the x axis shows categorical values representing the political parties such as **Conservative** or **Labour**. The y axis shows numerical values

<i>Category</i>	c		<i>Shape</i>	s	=	line	$\gamma, [u_{x1}, u_{y1}, \dots, u_{xn}, u_{yn}]$
<i>Ratio</i>	r					fill	$\gamma, [u_{x1}, u_{y1}, \dots, u_{xn}, u_{yn}]$
<i>Number</i>	n					text	γ, u_x, u_y, t
<i>Text</i>	t					bubble	$\gamma, u_x, u_y, n_w, n_h$
<i>Color</i>	γ					overlay	$[s_1, \dots, s_n]$
<i>Value</i>	v	=	cat	c, r		axis	$l/r/t/b$ s
			cont	n		padding	n_t, n_r, n_b, n_l, s

Fig. 3. Core primitives of the Compost domain-specific language. Values v are either categorical or continuous; a shape s is then defined as a simple recursive algebraic data type.

representing the number of seats won such as 365 MPs. When creating data visualizations, those are the values that the user needs to specify. This is akin to most high-level charting libraries such as Google Charts, but in contrast with more flexible libraries like D3.

Our design focuses on two-dimensional charts with x and y axes. Values mapped to those axes can be either categorical (e.g. political parties, and countries) or continuous (e.g. number of votes and exchange rates). The mapping from categorical and continuous values to positions on the chart is done automatically. We discuss this in Section 2.4.

For example, in the UK election results chart, the x axis is categorical. The library automatically divides the available space between the six categorical values (political parties). The value **Green** does not determine an exact position on the axis, but rather a range. To determine an exact position, we also need to attach a value between 0 and 1 to the categorical value. This identifies a relative position in the available range.

Figure 2 illustrates the two kinds of values using the axes from the UK election results chart. In Figure 3, we define a value v as either a continuous value **cont** n containing any number n or a categorical value **cat** c, r consisting of a categorical value c and a number r between 0 and 1. As discussed in Section 2.5, continuous and categorical values can also be annotated with units of measure to make the values more descriptive.

2.2 Basic primitives and combinators

Compost is an embedded domain-specific language, implemented as a set of functions. In the subsequent code samples, we will use color to distinguish primitives of the Compost language, such as **overlay** or **cat** from primitives of the host language such as **let** or **for**.

A chart element is represented by a shape s , as defined in Figure 3. A primitive shape can be a text label, a line connecting a list of points, a filled polygon defined by a list of points, or a bubble at a given point with a given width and height. The position of points is specified by x and y coordinates, which can be either categorical or continuous values. For text, line, polygon, and bubble, we also include a parameter γ that specifies the element color. The width and height of a bubble is given in pixels rather than in domain units.

Figure 3 also defines three combinators. The most important is **overlay**, which overlays given shapes. When doing this, Compost infers the range of values on the x and y axes and calculates suitable projections using a method discussed in the next section. The **padding** combinator adds padding around a specified shape and **axis** adds an axis showing

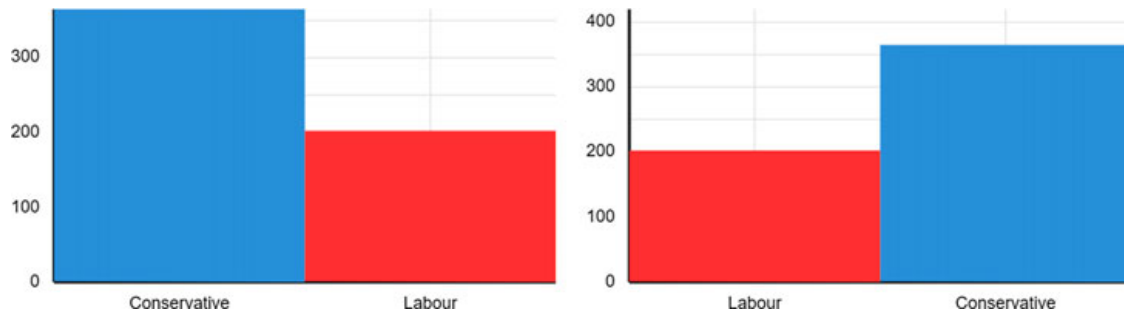


Fig. 4. Simple chart showing the UK election results; using automatically inferred scales (left) and using rounded Y scale and explicitly defined (reordered) X scale (right).

the inferred scale on the left, right, top, or bottom of a given shape. Using those primitives, we can construct the simple UK election results bar chart in Figure 4 (left):

```
let conservative, labour =
  fill #0000ff, [ (cat Conservative, 0), (cont 0), (cat Conservative, 0), (cont 365),
                 (cat Conservative, 1), (cont 365), (cat Conservative, 1), (cont 0) ],
  fill #ff0000, [ (cat Labour, 0), (cont 0), (cat Labour, 0), (cont 202),
                 (cat Labour, 1), (cont 202), (cat Labour, 1), (cont 0) ]

axisl (axisb (overlay [ conservative, labour ]))
```

We use the `let` construct of the host functional language to structure the code. The chart specification overlays two bars of different colors and then adds axes to the bottom and left of the chart. The two bars are filled rectangles defined using four corner points. The y coordinates are specified as continuous values, while the x coordinates are categorical. For the Conservative party, two of the points have the y coordinate set to `cont 0` (bottom of the bar) and two have the y coordinate set to `cont 365` (top of the bar). The two x coordinates are the start and the end of the range allocated for the `Conservative` category, that is, `cat Conservative, 0` on the left and `cat Conservative, 1` on the right.

Extending the snippet to generate a grouped bar chart that shows two results for each party as in Figure 1 is not much harder. Given a party p , we need to generate two rectangles, one with x coordinates `cat p, 0` and `cat p, 0.5` and the other with x coordinates `cat p, 0.5` and `cat p, 1`. In the following snippet, we use a `for` comprehension to generate the list. All remaining constructs are primitives of the Compost domain-specific language. Assuming `elections` is a list of election results containing a five-element tuple consisting of a party name, colors for 2017 and 2019, and results for 2017 and 2019, we create the chart using:

```
axisl (axisb (overlay [
  for party, clr17, clr19, mp17, mp19 in elections →
    padding 0, 10, 0, 10, overlay [
      fill clr17, [ (cat party, 0), (cont 0), (cat party, 0), (cont mp17),
                  (cat party, 0.5), (cont mp17), (cat party, 0.5), (cont 0) ],
      fill clr19, [ (cat party, 0.5), (cont 0), (cat party, 0.5), (cont mp19),
                  (cat party, 1), (cont mp19), (cat party, 1), (cont 0) ] ] ]))
```


Aside from iterating over all available parties and splitting the bar, the example also adds padding around the bars, which is specified in pixels. A similar result could be achieved by drawing a bar using a range from 0.05 to 0.5, but specifying padding precisely in pixels is sometimes preferable. The chart is still missing a title, which we add in Section 4.

2.3 Choosing the level of language abstraction

Perhaps the most important aspect of the design of any domain-specific language is the level of abstraction it uses. The bar chart example discussed in the previous section illustrates the choice made in Compost. On the one hand, Compost gives us flexibility by letting us compose charts from shapes. On the other hand, Compost limits what we can do using two-dimensional space with positions determined by categorical or continuous values. In other words, the Compost design lies in the middle of a broader spectrum.

An example of a more general domain-specific language is the Pan language (Elliott, 2003) for producing images. Pan represents images as functions from a 2D point to a color. This makes it possible to create powerful combinators, for example, polar image transformation, but it makes it harder to express logic important for charting, for example, automatic alignment of shapes defined in terms of categorical values. Compost makes it easy to put two bars side by side in a bar chart, but harder to define generic combinators for aligning images.

An example of a less general domain-specific language is the Haskell Chart library (Docker, 2020). Haskell Chart provides a wide range of plots (such as lines, candles, areas, points, error bars, pies, etc.), but those can only be composed in limited ways by overlaying them or arranging them in a grid. The language is closer to the domain of the most common applications, but it places more restrictions on what can be expressed.

The design of the Compost domain-specific language aims to capture the key principles shared by most charts but avoid using a long list of different plot types. Different types of charts are all produced by composing shapes, but the ways in which shapes can be composed and transformed are limited to those that are needed for typical charts. We discuss the limitations of this approach in more detail in Section 7.

2.4 Inferring scales and projections

We follow the terminology of Vega (Satyanarayan *et al.*, 2015) and use the term *scale* to refer to the mapping of values to positions on a screen; a *coordinate* is a value representing a position on a scale and the term *axis* is used to refer to the visual representation of a scale.

Scales are an important concept in Compost. When composing shapes using the *overlay* primitive, the user does not need to specify how to position the child elements relative to each other. The Compost library positions the elements automatically. This is done in two steps. During pre-processing, Compost infers the scales for x and y axes. A scale represents the range of values that needs to fit in the space available for the chart. When rendering a shape, Compost projects domain-specific values to the available screen space based on the inferred scale. A scale l is defined in Figure 5. A continuous scale is defined by a minimal and maximal value that need to be mapped to the available chart space. A categorical scale is defined by a list of individual categorical values. Note that we do not need minimal and

$$\text{Scale } l = \text{continuous } n_{\min}, n_{\max} \mid \text{categorical } [c_1, \dots, c_k]$$

Fig. 5. A scale l can be continuous, defined by a range, or categorical, defined by a list of values.

maximal ratios of the used categorical values as Compost will use equal space for each category, regardless of where in this space a shape needs to appear.

Scale inference is done by a simple recursive function that walks over the given shape and constructs two scales for the x and y axis, using the x and y coordinates that appear in the shape. Most of the work is done by a simple helper function that takes two scales, l_1 and l_2 , and produces a new scale that represents the union of the two:

$$\begin{aligned} \text{union } (\text{continuous } n_l, n_h) (\text{continuous } n'_l, n'_h) &= \\ &\text{continuous } \min(n_l, n'_l), \max(n_h, n'_h) \\ \text{union } (\text{categorical } [c_1, \dots, c_p]) (\text{categorical } [c'_1, \dots, c'_q]) &= \\ &\text{categorical } [c_1, \dots, c_p] @ [c'_i \mid \forall i \in 1 \dots q, \nexists j. c_j = c'_i] \end{aligned}$$

When unioning two continuous scales, the minimum and maximum of the resulting scale is the smallest and largest of the two minimums and maximums, respectively. When unioning two categorical scales, we take all values of the first scale and append all values of the second scale that do not appear in the first one. Note that this means that the order of categorical values in a scale depends on the order in which they appear in the shape. (A possible improvement to Compost would be to support ordinal values, which are categorical values with a well-defined ordering.) It is also worth noting that a categorical scale cannot be combined with a continuous scale. In other words, mixing categorical and continuous values in a single scale results in an error.

The scales inferred during pre-processing are later used when rendering a shape. We discuss the implementation in Section 6. The key operation is projection which, given a coordinate, a scale, and an area on the screen, produces a position on the screen. For a continuous scale, the projection is a linear transformation. For categorical scale with k values, we split the available chart space into k equally sized regions and then map a categorical value $\text{cat } c, r$ to the region corresponding to c according to the ratio r .

2.5 Types and units of measure

We introduce the Compost domain-specific language as untyped, but there are some obvious ways in which types can make composing charts in Compost safer. First, a type representing a shape could specify whether the x and y axes represent categorical or continuous values. This would rule out mixing of different values on a single scale and guarantee that the union operation, sketched in the previous section, is never called in a way leading to an undefined result. Second, the type of values mapped to an axis could be further annotated with units of measure (Kennedy, 2009). Using the F# notation where $n\langle u \rangle$ is a number n with unit u , an axis containing a value $\text{cont } 317\langle \text{mp} \rangle$ would then be incompatible with an axis containing a value $\text{cont } 1.32\langle \text{gbp/usd} \rangle$.

We only outline the type system here. There are two kinds of types: σ is a type of values and τ is a type of shapes. Assuming u denotes a unit of measure, the types are defined as:

$$\sigma = \text{Cat } u \mid \text{Cont } u \qquad \tau = \text{Shape } \sigma_x, \sigma_y$$

$$\begin{array}{l} \text{Shape } s = \text{roundScale}_{x/y} s \quad | \quad \text{nest}_{x/y} v_{\min}, v_{\max}, s \\ \quad | \quad \text{explicitScale}_{x/y} l, s \quad | \quad (\dots) \end{array}$$

Fig. 6. Additional combinators for controlling and nesting scales, extending earlier definition of s .

Correspondingly, there are two kinds of typing judgments; $v \vdash \sigma$ indicates the type of a value, while $s \vdash \tau$ indicates the type of a shape. The typing rules for two of the basic chart primitives, **line** and **overlay** look as follows:

$$\frac{v_{xi} \vdash \sigma_x v_{yi} \vdash \sigma_y}{\text{line } \gamma, [v_{x1}, v_{y1}, \dots, v_{xn}, v_{yn}] \vdash \text{Shape } \sigma_x, \sigma_y} \quad \frac{s_i \vdash \text{Shape } \sigma_x, \sigma_y}{\text{overlay } [s_1, \dots, s_n] \vdash \text{Shape } \sigma_x, \sigma_y}$$

The rule for **line** ensures that all X and Y values have the same types, σ_x and σ_y , respectively, and infers **Shape** σ_x, σ_y as the type of the shape. The rule for **overlay** ensures that all composed shapes have the same type, including the type of x and y scales.

3 Advanced charts: Controlling scale composition

Most charts have one x and one y scale that are determined by the values the chart shows, but there are interesting exceptions. The chart in Figure 1 (right) has two different y axes, one for GBP/USD and one for GBP/EUR. In the next two sections, we look at three combinators that control the scale inference process and what flexibility this enables.

3.1 Defining nice scale ranges

The automatic scale inference often results in scales where the maximum is a non-round number. This leads to charts that fully utilize the available space but may not be easy to read. The first two primitives, shown in Figure 6 (left), allow the chart designer to adjust the automatically inferred range of scales. The operations can be applied to either the x scale or the y scale, which is indicated by the x/y subscript. The **roundScale** primitive takes the inferred x or y scale of the shape s and, if it is a continuous scale, rounds its minimal and maximal values to a “nice” number. For example, if a continuous scale has minimum 0 and maximum 365, the resulting scale would have a maximum 400. For categorical scale, the operation does not have any effect. The **explicitScale** operation replaces the inferred scale with an explicitly provided scale (the type of the inferred scale has to match with the type of the explicitly given scale). For example, the chart in Figure 4 (right) is constructed using the following code (reusing the conservative and labour variables defined earlier):

```
axis_l (axis_b (roundScale_y (explicitScale_x (categorical [ Labour, Conservative ]),
    overlay [ conservative, labour ] )))
```

Reading the code from the inside out, the snippet first overlays the two colored bars defined earlier; it then replaces the X axis with an explicitly given one that changes the order of the values. As a result, the bar for **Labour** will appear on the left, even though the value comes later in the list of overlaid chart elements.

The code next uses **roundScale** to automatically round the minimum and maximum of the continuous Y scale (showing the total number of seats). Finally, we add axes around the

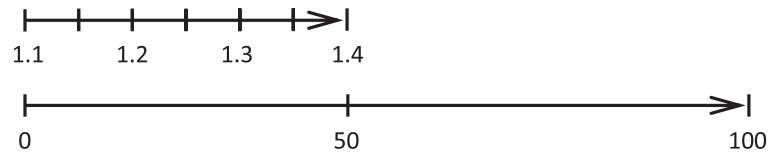


Fig. 7. A continuous scale with values from 0 to 6, nested in another scale.

shape, producing a usual labeled chart. It is worth noting that `axis` and `roundScale` could be implemented as derived operations; `roundScale` would need to infer the scale of the nested shape and then insert `explicitScale` with a rounded number; `axis` would also need to infer the scales and then generates labels and lines in suitable locations.

3.2 Nested scales

The most interesting primitive for controlling scale composition is `nestx/y`. As with other primitives like `padding`, the primitive takes a shape with some additional parameters and defines a new shape. Its behavior is similar to that of the SVG viewport (Dahlström *et al.*, 2011). The `nest` primitive takes two values, v_{\min} and v_{\max} , and a shape s as arguments and nests the scale of the shape s inside the region defined by v_{\min} , v_{\max} . When inferring scales of shapes, the scale of `nestx/y v_{\min} , v_{\max} , s` will be a categorical or continuous scale constructed from the values v_{\min} and v_{\max} , regardless of the values that are used inside the shape s . The chart space between v_{\max} and v_{\min} will then be used to render the nested shape s using its inferred scale. In other words, the operation defines a virtual coordinate system that exists only inside the newly created shape but is invisible to anything outside of the shape. An example of nesting is shown in Figure 7. Here, a chart with a continuous scale from 1.1 to 1.4 (e.g. GBP/EUR exchange rates) is nested in the left half of another chart, which has a continuous scale from 0 to 100.

The nesting of scales can be used in a variety of ways. For example, to nest a scatter plot showing individual data points inside a bar of a histogram, we would use `cat ABC, 0` and `cat ABC, 1` as the points that define the start and the end of the region. A simpler use case for the combinator is showing multiple charts in a single view. For example, the motivating example in Figure 1 (right) compares aligned line charts of exchange rates for two different currencies. Assuming `gbpusd` and `gbpeur` are lists containing days as x values and exchange rates as y values, we can construct a simple chart with two line charts, as shown in Figure 8 (left), using:

```
overlay [ nesty (cont 0), (cont 50), (axisl (axisr (axisb (line #202020 gbpusd))))
          nesty (cont 50), (cont 100), (axisl (axisr (axisb (line #202020 gbpeur)))) ]
```

In this example, the x scale shows the days of the year. This scale is shared by both of the charts. Indeed, if data were only available for the second half of the month for one of the charts, we would want the line to start in the middle of the chart. However, the y scale needs to be separate for each of the charts. To achieve this, we use `nesty`. The scale of the inner shapes is continuous, from the minimal to the maximal exchange rate for a given period. The outer scale is determined by the explicitly defined points. For the upper chart, these are `cont 0` and `cont 50`; for the lower chart, these are `cont 50` and `cont 100`. The continuous values define a scale that only contain two shapes – one in the upper half

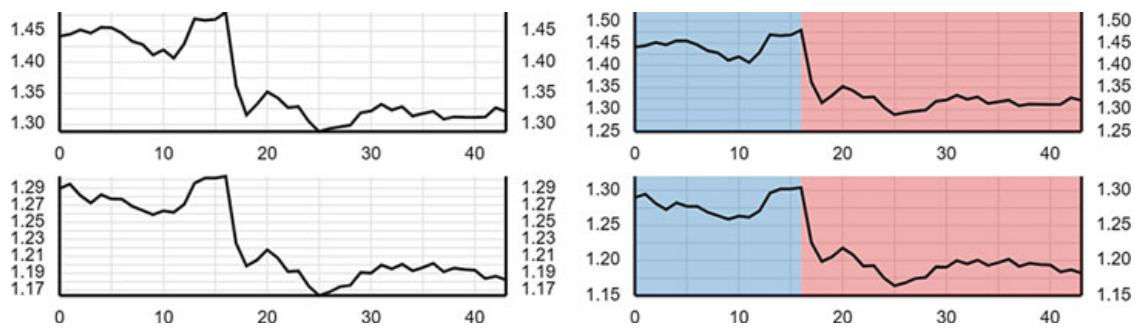


Fig. 8. Two charts showing currency exchange rates with a shared X scale and separate Y scales.

and one in the lower half – and so the three numbers could have equally been, for example, 0, 1, 2. The outer scale used here is synthetic and it is not aligned with other chart elements. A chart that does not have synthetic outer scale is pairplot, discussed in the next section.

For completeness, the following code snippet shows how to construct the full currency exchange rate chart shown in Figure 8 (right), including the blue and red background:

```
let xrate (lo, hi) rates = overlay [
  fill #1F77B460, [ cont 0, cont lo, cont 16, cont lo, cont 16, cont hi, cont 0, cont hi ],
  fill #D6272860, [ cont 16, cont lo, cont 44, cont lo, cont 44, cont hi, cont 16, cont hi ],
  line #202020 rates ]

overlay [ nest_y (cont 0), (cont 50), (axis_l (axis_r (axis_b (xrate (1.25, 1.50) gbpusd))))
  nest_y (cont 50), (cont 100), (axis_l (axis_r (axis_b (xrate (1.15, 1.30) gbpeur)))) ]
```

Here, we use the `let` binding of the host language to define a function that takes the data rates together with the minimum and maximum. This is used for drawing two filled rectangles, covering the first 16 days of the view in blue and the rest in red. The shapes combined using `overlay` are rendered in the order in which they appear and so the line shape is last, so that it appears above the background.

4 Standard charts: Defining new abstractions

The functional domain-specific language design makes it easy to define high-level chart features and chart types, known from standard charting libraries, using the low-level primitives of the core language. To illustrate this, we give two examples.

First, one last remaining feature of the two charts in Figure 1 is a chart title. This can be added to any chart using the following derived combinator:

```
let title t s = overlay [
  nest_x (cont 0), (cont 100), (nest_y (cont 0), (cont 15),
    explicitScale_x (continuous 0, 100), (explicitScale_y (continuous 0, 100),
      text #000000, (cont 50), (cont 50), t )
  nest_x (cont 0), (cont 100), (nest_y (cont 15), (cont 100), s) ]
```

We use `let` in the host language to define `title` as a function taking a title t and a shape s . It overlays two shapes. To position the title above the chart, the first shape has an outer y scale `continuous 0, 15`, while the second has an outer y scale `continuous 15, 100`. Similarly, the outer x scale of both is `continuous 0, 100`. These are defined using `nestx/y`.

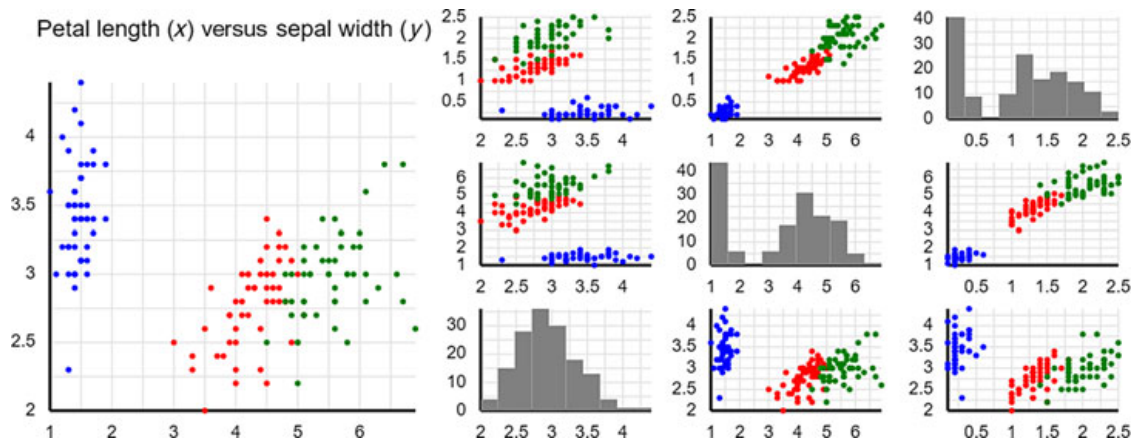


Fig. 9. Sample charts built using derived abstractions; a scatter plot visualizing the Iris dataset with a title (left) and a pairplot comparing two Iris features (right).

The second shape simply wraps the specified chart s to which we are attaching the title. The first positions the text title in the middle of the available space. To do so, we explicitly set the x and y scales inside the upper shape to continuous scales from 0 to 100 and then position the text label in the middle, at a point $(\text{cont } 50), (\text{cont } 50)$. We assume that the `text` primitive centers the text, although the actual implementation also allows the user to specify horizontal and vertical alignment. Figure 9 (left) shows a sample scatter plot chart with a title created using the title combinator.

A more complex chart that can be composed using the Compost primitives is pairplot from the seaborn library (Waskom *et al.*, 2014). Pairplot visualizes pairwise relationships between features of a dataset. An example using three features (sepal width, petal width, and petal length) from the Iris dataset is shown in Figure 9 (right). A pairplot draws a grid of charts, each visualizing the relationship between two numerical features. For distinct features, pairplot shows a scatter plot using one feature for x values and the other for y values. When the features are the same (the diagonal), it draws a histogram of the feature values. A categorical feature can be used to determine the color of dots in the scatter plots.

To generate a pairplot, we use `nest` to overlay and align a grid of plots. Each of those overlays a number of bubbles or filled shapes and adds left and bottom axis. As before, we use `let` to define a function and list comprehensions to generate individual chart elements. We assume that data is a list of rows, `attrs` is a list of available attributes, and `get a r` obtains the attribute a of a row r . We also assume the dataset contains the “color” attribute:

```
let pairplot attrs data = overlay [
  for x in attrs → for y in attrs →
    nest_x (cat x, 0), (cat x, 1), (nest_y (cat y, 0), (cat y, 1), axis_l (axis_b
      (if x ≠ y then overlay [ for r in data →
        bubble (get “color” r), (get x r), (get y r), 1, 1 ]
      else overlay [ for x1, x2, y in bins x data →
        fill #808080 [x1, y, x2, y, x2, 0, x1, 0 ] ])))]
```

As before, `nest` is essential for composing individual charts. Here, the points that determine the locations of individual charts are categorical values defined by the attributes of the

$$s = \begin{array}{l|l} \text{mouseUp } (\lambda x y \rightarrow e), s & \text{mouseDown } (\lambda x y \rightarrow e), s \\ \text{mouseMove } (\lambda x y \rightarrow e), s & (\dots) \end{array}$$

Fig. 10. Additional combinators for mouse-based interaction, extending earlier definition of s .

dataset. The choice between two possible nested charts is made using the host language `if` construct. Scatter plots are generated by overlaying bubbles with x and y coordinates obtained using `get x r` and `get y r`. Histograms are composed from filled shapes. To obtain their locations, we use a helper function `bins x data`, which returns a list of bins specified by a triple consisting of a lower and an upper range x_1, x_2 and the count y .

The example shows that Compost is simple yet expressive. With just a few lines of code, we are able to construct charts that, in other systems, require dedicated libraries. The essential aspect of the language making this possible is the automatic inference of scales and their mapping to the available space as well as the `nest` operation.

5 Interactive charts: Domain-specific event handling

Many data visualizations published on the web feature interactivity. Standard forms of interactivity include animations, hover labels, or zooming. More interesting custom visualizations include “You Draw It” introduced by the New York Times (Aisch *et al.*, 2015). The chart shows only the first half of the data, such as a timeline, and the reader has to guess the second half before clicking a button and seeing the actual data. Standard forms of interactivity are often supported by high-level libraries; Google Charts supports panning using drag & drop, zooming to a selected chart range and animations. Custom interactivity is typically implemented using low-level libraries such as D3, but doing so requires directly handling JavaScript events and modifying the browser DOM.

Compost uses the Elm architecture (Czaplicki, 2016) to support interactive data visualizations. In this model, an interactive visualization is described using a pair of user-defined types and a pair of user-defined functions. The *state* type represents the current state of what is displayed (e.g. animation step or selection) and the *event* type represents actions that the user can perform (e.g. start an animation or draw a selection range). The two functions use the state and event types. The *view* function creates a chart based on the current state and the *update* function specifies how the state changes when an event occurs.

5.1 Domain-specific events

To support handling of mouse-based events, Compost adds three additional primitives to the definition of `shape`, as shown in Figure 10. The three new primitives make it possible to handle three common mouse events using custom functions $\lambda x y \rightarrow e$, specified in the host language. The most interesting aspect is that the functions are given x and y coordinates of the event specified in the domain units of the chart. This means that if the user clicks on the bar representing the Conservative party in a bar chart, the values might be, for example, `cat Conservative`, `0.75` for x and `cont 120.5` for y .

5.2 You Draw It data visualizations

To illustrate building interactive data visualizations using Compost, we look at one aspect of “You Draw It.” We want to create a bar chart where the user can use drag & drop to move individual bars. Figure 11 shows the interactive chart before and after an interaction. The first step is to define types representing the state and events that can occur:

```
type State = bool * (string * int) list
type Event = Update of (string * int) | Moving of bool
```

The state is a pair of a boolean, indicating whether the user is currently dragging, and a list of key/value pairs, storing the number of seats for each political party. Two types of events can occur in the visualization. First, the user may start or stop dragging, which is indicated using `Moving(true)` and `Moving(false)`, respectively. Second, the user may change a value for a party, which is represented by the `Update` event.

The next part of the implementation is the update function which takes an old state together with an event and produces a new state:

```
let update (_, s) (Moving(m)) = m, s
    update (true, s) (Update(p, v)) = true, map (λ(k, o) → k, if k = p then v else o) s
    update (m, s) (Update(_, _)) = m, s
```

The first case handles the `Moving` event, which replaces the first component of the state tuple, that is, a flag indicating whether a mouse button is down. The next two cases handle the `Update` event. The event carries two values, p and v , which represent the party (which bar the user is dragging) and the new value (new number of seats). If the user is currently dragging, we replace the value associated with the party p in the list s using the `map` function. If the user is not currently dragging, the event is ignored.

Finally, the view function takes the current state and builds the data visualization using the Compost domain-specific language. In addition, it also takes a parameter `trigger`, which is an effectful function of type `Event → unit` that can be used to trigger events in handlers, registered using primitives such as `mouseMove`. The trigger function is provided by the Compost runtime. When it is invoked from an event handler, it takes the current state, transforms it using the `update` function, sets the new state as the current state, and invokes the `view` function to display the new state.

To build the bar chart in Figure 11, we use the same approach as in Section 2.2. The only addition are the event handlers registered using `mouseMove`, `mouseUp`, and `mouseDown`:

```
let view trigger (_, state) =
  axisl (axisb (explicitScaley (continuous 0, 400),
    ( mouseMove (λ (cat p, _) (cont v) → trigger(Update(p, v))),
    ( mouseUp (λ _ _ → trigger(Moving(true))),
    ( mouseDown (λ _ _ → trigger(Moving(false))), overlay [
      for party, mps in state → padding 0, 10, 0, 10, (fill (color party),
        [(cat party, 0), (cont 0), (cat party, 0), (cont mp),
        (cat party, 0.5), (cont mps), (cat party, 0.5), (cont 0) ] ]))))))
```

When the user interacts with the visualization created using Compost, the library translates the coordinates associated with events from pixels to domain-specific values. In case

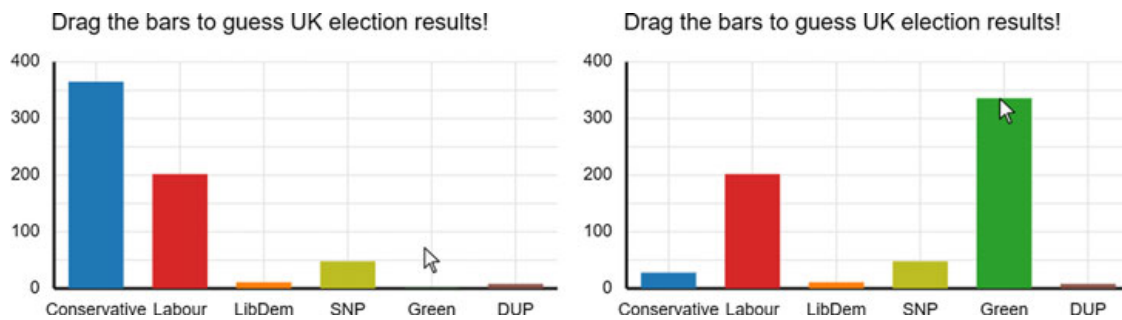


Fig. 11. Interactive “You Draw it” data visualization. The user moves cursor to a bar (left), pushes a mouse button, and drags the bar to the position that they think is the correct one (right).

of the above bar chart, when the user moves a mouse, the function registered using `mousemove` is given a categorical value `cat p, r` as the x coordinate, and a continuous value `cont v` as the y coordinate. It then takes p , which is the name of the party corresponding to the bar and the value v corresponding to the number of seats and triggers the `Update(p, v)` event to update the state. The handlers for `mouseup` and `mousedown` do not use the coordinates. They simply switch the flag indicating whether the user is currently dragging or not.

The primitives for specifying mouse event handlers can be nested or appear in multiple subshapes of the composed shape. This makes it possible to attach different event handlers to different parts of a chart and get event coordinates in local units. In case of nesting, the nested handler will capture events that occur in the space occupied by the shape it wraps, but it will ignore events occurring outside of this area.

The pair of functions, `update` and `view`, together with an initial state is all that is needed to create an interactive data visualization. `Compost` calls `view` each time the state changes and uses `virtual-dom` to update the chart displayed in the browser. Although creating an interactive visualization is more work than creating a static one, the domain-specific nature of `Compost` is invaluable. We can simply take the values p and v produced by a mouse event, use those to update the state and then, again, render an updated chart.

6 Implementation structure: Scale inference and rendering

`Compost` is an open-source library, implemented in the functional language F#. The full source code can be found at <http://github.com/compostjs>. As is often the case with functional domain-specific languages, the implementation is not difficult once we find the right collection of basic primitives and the right structure for the implementation. This largely applies to the `Compost` library and so we will not go into the implementation details. It is, however, worth giving an outline of the implementation structure.

As mentioned in Section 2.4, the rendering of shapes proceeds in two stages. First, the library infers the scales of a shape. When doing so, it also annotates some shapes with additional information that is needed later for rendering. Second, the library projects the shape onto an available space and produces the chart, represented as an SVG object.

6.1 Inferring the scales of a shape

In order to render a shape, we need to know the range of values that should appear on the x and y axes. This is done by inferring a scale for each of the axes from the individual x

and y coordinates that specify shape locations. As discussed earlier, a scale can be either categorical (displaying only categorical values) or continuous (displaying only continuous values). When inferring scales, we use two helper operations: union, discussed earlier, combines two scales and singleton creates a scale from a single coordinate.

The operation that infers the scales of a shape is `calculateScales`. It takes a shape and produces a pair of x and y scales, together with a transformed shape:

$$\text{calculateScales} : \text{Shape} \rightarrow (\text{Scale} * \text{Scale}) * \text{Shape}$$

The operation does not need to transform the shape in most cases. The exception is the shape `nestx/y vmin, vmax, s`. In this case, the returned scale is based solely on the values of v_{\min} and v_{\max} . For rendering, we need to keep the inferred scales of the nested shape s . To do so, the operation replaces the `nestx/y` shape with an auxiliary shape `scaledNestx/y`:

$$s = \text{scaledNest}_{x/y} \ v_{\min}, v_{\max}, s_{x/y}, s \quad | \quad (\dots)$$

There are two kinds of cases handled by `calculateScales`. For primitives, it constructs a pair of scales from individual coordinates using union and singleton. For shapes containing a subshape, the operation calculates the scales of a subshape recursively and then adapts those somehow. To illustrate, we consider two interesting cases:

```
calculateScales (nestx vmin, vmax, s) =
  let (sx, sy), s' = calculateScales s
  (union (singleton vmin) (singleton vmax), sy), scaledNestx vmin, vmax, sx, s'
```

```
calculateScales (overlay l) =
  let scales, l' = unzip (map calculateScales l)
  let sx, sy = unzip scales
  (reduce union sx, reduce union sy), overlay l'
```

When calculating the scales of the `nestx`, the function first calculates scales of the subshape s recursively. The resulting y scale s_y is returned as the result, while the x scale is obtained from the two coordinates v_{\min} and v_{\max} . This is also the case where the shape is transformed and the returned `scaledNestx` shape stores the inferred x scale s_x of the subshape s . The second example is the `overlay` case which recursively computes scales of all subshapes and combines those using the list folding function `reduce` with union as an argument.

6.2 Projecting coordinates and drawing

The key operation that needs to be performed when drawing a shape is projecting coordinates from domain-specific values to the screen coordinates. As we draw a shape, we keep the x and y scale and the space in pixels that it should be drawn on. Initially, the x and y scales are those inferred for the entire shape and the space in pixels is `0 . . . width` and `0 . . . height` where `width × height` is the size of the target SVG element.

The key calculation is done by the `project` function, which takes the space in pixels (as a pair of floating point numbers representing the range), the current scale, and a domain-specific value and produces a coordinate in pixels:

$$\text{project} : \text{float} * \text{float} \rightarrow \text{Scale} \rightarrow \text{Value} \rightarrow \text{float}$$

The function is only defined if the value and scale are compatible. As discussed in Section 2.5, this could be guaranteed using a simple type system. If both are continuous, the function performs a simple linear transformation. If both are categorical, the available pixel space is divided into a equally sized bins, one for each categorical value on the scale, and the value is then projected into the appropriate bin.

The drawing of shapes is done by a function that takes the available area as a quadruple $(x_1, y_1), (x_2, y_2)$ together with the x and y scale mapped onto the area and a shape to be drawn. The result is a data structure representing an SVG document:

$$\text{drawShape} : (\text{float} * \text{float}) * (\text{float} * \text{float}) \rightarrow \text{Scale} * \text{Scale} \rightarrow \text{Shape} \rightarrow \text{Svg}$$

For primitive shapes, the operation projects the coordinates using `project` and constructs a corresponding SVG document. For shapes with subshapes, it calls itself recursively, possibly with an adjusted scale or area. The two cases discussed earlier illustrate this:

```
drawShape a s (overlay l) =
  concat (map (drawShape a s) l)

drawShape ((x1, y1), (x2, y2)) (sx, sy) (scaledNest_x v_min, v_max, ns_x, shape) =
  let x'_1 = project (x1, x2) sx v_min
      x'_2 = project (x1, x2) sx v_max
  in drawShape ((x'_1, y1), (x'_2, y2)) (ns_x, sy) shape
```

When drawing `overlay`, the function draws all subshapes onto the same area using the same scales and then concatenates the returned SVG components using the `concat` helper. The `scaledNest_x` case is more illuminating. Here, we first use `project` to find the range x'_1, x'_2 corresponding to the domain values v_{\min} and v_{\max} . This defines the area corresponding to the nested scale ns_x , onto which the x coordinates in the subshape `shape` should be projected. To do this, we recursively call `drawShape` but use x'_1 and x'_2 as the x coordinates of the target area and ns_x as the x scale. The y area and scales are propagated unchanged.

7 Limitations and future work

As discussed in Section 2.3, the Compost library chooses a level of abstraction that makes it possible to express a wide range of charts but does not allow arbitrary image manipulation. The examples discussed so far provide a good review of what can be expressed using Compost. It is also worth considering what cannot currently be expressed. For many of the current limitations, we also consider what additional primitive would address the problem.

7.1 Radial charts and image transformations

Compost cannot currently produce pie charts and other radial charts. This could be supported by defining a primitive `polar` that renders a shape s specified as a parameter using a polar coordinate system instead of the default Cartesian system. Like the `nest` primitive, this would create a new shape that occupies a newly defined chart region. The `polar` primitive would make it possible to create pie charts, but also more elaborate Circos charts used to visualize genomic data (Krzywinski *et al.*, 2009).

Radial charts provide a clear motivation for supporting polar geometries, but we do not currently expect the need for more general image transformations such as those supported by Pan (Elliott, 2003). Those are useful for producing visually appealing images but may not be necessary for data visualization. Arguably, we also do not expect the need for more general layout combinators such as above or besides (Yorgey, 2012). Those can be expressed elegantly using image transformations. In Compost, we can achieve similar effect using `nest` and `explicitScale`, as shown when defining title in Section 4.

7.2 *Combinations and transformations of scales*

Another area in which Compost could be extended is to allow more flexible handling of scales. Currently, categorical scales are mapped to bins of equal size and continuous scales are mapped using a linear transformation. The current design does not make it possible to use logarithmic scale or, for example, contracted axis where a subrange of values in the middle is omitted. Both of these could be supported if Compost allowed the user to specify a custom value transformation function.

Another interesting challenge is to allow overlaying of charts with multiple scales. This can currently be done using `overlay` together with `nest`. However, a more principled approach would be to allow the user to specify multiple, possibly named, scales for each shape. The `calculateScales` operation discussed in Section 6.1 would then need to return a list of scales rather than just a pair.

7.3 *Controlling visual elements of a chart*

There is also a number of occasions where the user might require more control over various visual elements of the chart such as fonts, text alignment, or visual aspects of the automatically generated axes. The current implementation of Compost already allows control over fonts, font sizes, and text alignment, but we omit the details for brevity.

Controlling the visual aspects of axes is a more interesting problem. In fact, the `axis` primitive described in this paper is not a primitive operation, but rather a derived one. It is implemented by calculating the scales of the shape specified as an argument and overlaying it with lines (for axes and grid), text elements (for labels), and adding a padding. The current implementation does not allow much customization, but the user can look at the implementation and easily create their own version, much like they can create their own version of the title operation described in Section 4.

8 Conclusions

This paper presents a functional take on the problem of designing easy to use, but flexible abstractions for composing data visualizations. We hope to find a sweet spot between high level, but inflexible approaches, and low level, but hard to use approaches.

Most work in this space is based on Grammar of Graphics (Wilkinson, 1999), designing more or less complex and powerful variants (Stolte *et al.*, 2002; Wickham, 2010; Satyanarayan *et al.*, 2015, 2016). In Grammar of Graphics, a chart is a mapping from

data to chart elements and their visual attributes. In contrast, in Compost, the mapping is specified in the host programming language and a chart is merely a resulting data type describing the visual elements using domain-specific primitives.

Our approach is very flexible as it lets the user compose primitive visual elements in any way they want; it lets them define their own high-level abstractions and it also integrates well with reactive programming architectures to support interactive data visualizations.

In this paper, we focus on presenting the core ideas behind Compost. However, much remains to be explored, both in terms of finding the best set of primitives and in terms of their language integration. First, we only support categorical and continuous values, but there are also ordinal values (which cannot be compared, but can be sorted). Second, some of our primitives, namely `axis` and `roundScale`, could be implemented as derived operations, but we treat those as built-in for simplicity. Third, we only treat x and y as scales, but we could similarly treat other visual features (colors of bars and size of bubbles) as scales, which would allow a more high-level specification of certain charts.

Acknowledgements

The Compost library is the result of my prolonged effort to create an elegant charting API for F#, which was supported, at various stages, by Don Syme at Microsoft Research and Howard Mansell at BlueMountain Capital. The idea of Compost first came together in discussion with Mathias Brandewinder and was (much much later) implemented thanks to the support of Google Digital News Initiative and The Alan Turing Institute. The final motivation for this paper was an invitation to talk at the Lambda Days conference in Kraków and the positive comments from the attendees. Finally, the anonymous referees provided valuable feedback that made this a better paper.

Supplementary materials

For supplementary material for this article, please visit <http://doi.org/10.1017/S0956796821000046>

Conflicts of Interest

None.

References

- Aisch, G., Cox, A. & Quealy, K. (2015) *You Draw it: How Family Income Predicts Children's College Chances*. New York Times. Accessed May 24, 2020. Available at: <https://www.nytimes.com/interactive/2015/05/28/upshot/you-draw-it-how-family-income-affects-childrens-college-chances.html>.
- Bostock, M., Ogievetsky, V. & Heer, J. (2011) D³ data-driven documents. *IEEE Trans. Visualization Comput. Graphics* **17**(12), 2301–2309.

- Czaplicki, E. (2012) *Elm: Concurrent FRP for Functional GUIs*. Senior Thesis, Harvard University. Available at <https://elm-lang.org/assets/papers/concurrent-frp.pdf>.
- Czaplicki, E. (2016) *A Farewell to FRP: Making Signals Unnecessary with The Elm Architecture*. Accessed May 24, 2020. Available at: <https://elm-lang.org/news/farewell-to-frp>.
- Dahlstr m, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D. & Watt, J. (2011) *Scalable Vector Graphics (svg) 1.1*, 2nd ed. W3C Recommendation. Accessed May 24, 2020. Available at: <http://www.w3.org/TR/2011/REC-SVG11-20110816/>.
- Docker, T. (2020) *Chart: A Library for Generating 2D Charts and Plots*. Haskell Hackage. Accessed December 9, 2020. Available at: <https://hackage.haskell.org/package/Chart>.
- Elliott, C. (2003) Functional images. In *The Fun of Programming*, Chapter 7, Gibbons, J. & de Moor, O. (eds). Palgrave.
- Google. (2020) *Google Charts: Interactive Charts for Browsers and Mobile Devices*. Google. Accessed May 24, 2020. Available at: <https://developers.google.com/chart>.
- Kennedy, A. (2009) Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*. Springer, pp. 268–305.
- Krzywinski, M., Schein, J., Birol, I., Connors, J., Gascoyne, R., Horsman, D., Jones, S. J. & Marra, M. A. (2009) Circos: An information aesthetic for comparative genomics. *Genome Res.* **19**(9), 1639–1645.
- Satyanarayan, A., Moritz, D., Wongsuphasawat, K. & Heer, J. (2016) Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization Comput. Graphics* **23**(1), 341–350.
- Satyanarayan, A., Russell, R., Hoffswell, J. & Heer, J. (2015) Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Visualization Comput. Graph.* **22**(1), 659–668.
- Stolte, C., Tang, D. & Hanrahan, P. (2002) Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Visualization Comput. Graphics* **8**(1), 52–65.
- Waskom, M., Botvinnik, O., Hobson, P., Warmenhoven, J., Cole, J. B., Halchenko, Y., Vanderplas, J., Hoyer, S., Villalba, S. & Quintero, E. (2014) *Seaborn: Statistical Data Visualization*. Accessed May 24, 2020. Available at: <https://seaborn.pydata.org/>.
- Wickham, H. (2010) A layered grammar of graphics. *J. Comput. Graphical Stat.* **19**(1), 3–28.
- Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer.
- Wilkinson, L. (1999) *The Grammar of Graphics*. New York: Springer-Verlag.
- Yorgey, B. A. (2012) Monoids: Theme and variations (functional pearl). In Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012, Voigtl nder, J. (ed). ACM, pp. 105–116.

Chapter 13

Linked visualizations via Galois dependencies

Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked visualisations via Galois dependencies. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. <https://doi.org/10.1145/3498668>

Linked Visualisations via Galois Dependencies

ROLY PERERA^{*}, The Alan Turing Institute, UK

MINH NGUYEN, University of Bristol, UK

TOMAS PETRICEK[†], University of Kent, UK

MENG WANG, University of Bristol, UK

We present new language-based dynamic analysis techniques for linking visualisations and other structured outputs to data in a fine-grained way, allowing users to explore how data attributes and visual or other output elements are related by selecting (focusing on) substructures of interest. Our approach builds on bidirectional program slicing techniques based on Galois connections, which provide desirable round-tripping properties. Unlike the prior work, our approach allows selections to be negated, equipping the bidirectional analysis with a De Morgan dual which can be used to link different outputs generated from the same input. This offers a principled language-based foundation for a popular view coordination feature called *brushing and linking* where selections in one chart automatically select corresponding elements in another related chart.

CCS Concepts: • **Theory of computation** → **Program semantics**.

Additional Key Words and Phrases: Galois connections; data provenance

ACM Reference Format:

Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked Visualisations via Galois Dependencies. *Proc. ACM Program. Lang.* 6, POPL, Article 7 (January 2022), 29 pages. <https://doi.org/10.1145/3498668>

1 INTRODUCTION

Techniques for dynamic dependency analysis have been fruitful, with applications ranging from information-flow security [Sabelfeld and Myers 2003] and optimisation [Kildall 1973] to debugging and program comprehension [De Lucia et al. 1996; Weiser 1981]. There are, however, few methods suitable for fine-grained analysis of richly structured outputs, such as data visualisations and multidimensional arrays. Dataflow analyses [Reps et al. 1995] tend to focus on analysing variables rather than parts of structured values. Where-provenance [Buneman et al. 2001] and related data provenance techniques are fine-grained, but are specific to relational query languages. Taint tracking [Newsome and Song 2005] is also fine-grained, but works forwards from input to output. For many applications, it would be useful to be able to focus on a particular part of a structured output, and have an analysis isolate the input data pertinent only to that substructure.

This is a need that increasingly arises outside of traditional programming. Journalists and data scientists use programs to compute charts and other visual summaries from data, charts which must be interpreted by colleagues, policy makers and lay readers alike. Interpreting a chart correctly

^{*}Also with University of Bristol.

[†]Also with The Alan Turing Institute.

Authors' addresses: Roly Perera, The Alan Turing Institute, London, UK, rperera@turing.ac.uk; Minh Nguyen, min.nguyen@bristol.ac.uk, University of Bristol, Bristol, UK; Tomas Petricek, University of Kent, Canterbury, UK, tpetricek@kent.ac.uk; Meng Wang, meng.wang@bristol.ac.uk, University of Bristol, Bristol, UK.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2022 Copyright held by the owner/author(s).

2475-1421/2022/1-ART7

<https://doi.org/10.1145/3498668>

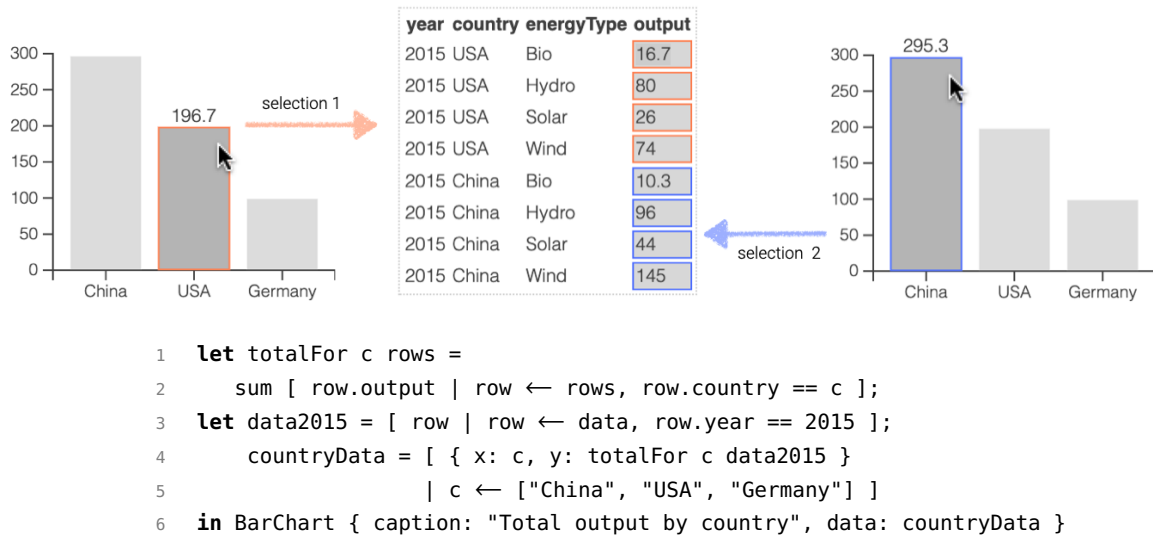


Fig. 1. Fine-grained linking of outputs to inputs, focusing on data for USA (left) and China (right).

means understanding what the components of the visualisation actually *represent*, i.e. the mapping between data and visual elements. But this is a hard task, requiring time and expertise, even with access to the data and source code used to create the visualisation. It is easy for innocent (but devastating) mistakes such as transposing two columns of data to go unnoticed [Miller 2006]. Since visualisations are simply cases of programs that transform structured inputs (data tables) into structured outputs (charts and other graphics), general-purpose language-based techniques for fine-grained dependency tracking should be able to help with this, by making it possible to reveal these relationships automatically to an interested user.

1.1 Linking Structured Outputs to Structured Inputs

First, interpreting a chart would be much easier if the user were able to explore the relationship between the various parts of the chart and the underlying data interactively, discovering the relevant relationships on a need-to-know basis. For example, selecting a particular bar in a bar chart could highlight the relevant data in a table, perhaps showing only the relevant rows, as illustrated in Figure 1. We could certainly do more and say something about the nature of the relationship (summation, in this case), but even just revealing the relevant data puts a reader in a much better position to fact-check or confirm their own understanding of what they are looking at. (The figure shows how selecting the bar for the USA should highlight different data than selecting the bar for China.) Indeed, this is useful enough that visualisation designers sometimes create “data-linked” artefacts like these by hand, such as Nadieh Bremer’s award-winning visualisation of population density growth in Asian cities [Bremer and Ranzijn 2015], at the cost of significant programming effort. Libraries such as Altair [VanderPlas et al. 2018] alleviate some of this work, but require data transformations to be specified using a limited set of combinators provided (and understood) by the library.

What we would like to do is allow data scientists to author analyses and visualisations using an expressive functional language like the one shown in Figure 1, and obtain data linking automatically for the generated artefact, as a baked-in transparency feature. At the core of this is a program analysis problem: we want to be able to focus on a particular visual attribute — say the value of y in the record $\{x: \text{"USA"}, y: 196.7\}$ passed to `BarChart` in the example above — and perform some kind of backwards analysis to determine the relevant inputs, in this case the value of output in four

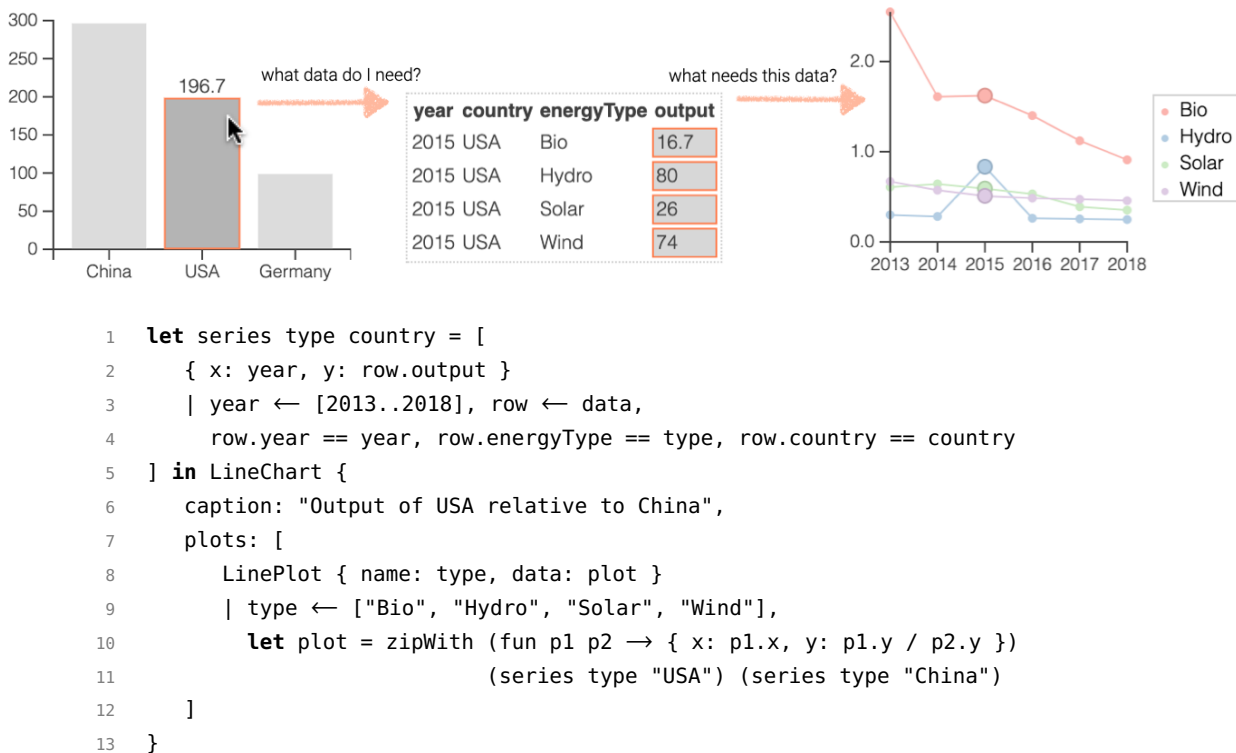


Fig. 2. Linking visualisations via common data dependencies

of the records that appear in the data source. Framing this as a program analysis problem not only provides a path to automation, but also invites interesting questions that a hand-crafted solution is unlikely to properly address. For example, does the union of two output selections depend on the union of their respective dependencies? Do dependencies “round-trip”, in that they identify sufficient resources to reconstruct the selected output? Are they minimal? These questions are important to establishing trust, and a language-based approach offers a chance to address them.

1.2 Linking Structured Outputs to Other Structured Outputs

Second, authors often present distinct but related aspects of data in separate charts. In this situation a reader should be able to focus on (select) a visual element in one chart or other structured output and automatically see elements of a different chart which were computed using related inputs. For example in Figure 2 below, selecting the bar on the left should automatically highlight all the related visual elements on the right. This is a well-recognised use case called *brushing and linking* [Becker and Cleveland 1987], which is supported by geospatial applications like GeoDa [Anselin et al. 2006] and charting libraries like Plotly, but tends to be baked into specific views, or require programmer effort and therefore anticipation in advance by the chart designer. Moreover these applications and libraries provide no direct access to the common data which explains why elements are related.

Again, we would like to enable a more automated (and ubiquitous) version of brushing and linking, without imposing a burden on the programmer. They should be able to express visualisations and other data transformations using standard functional programming features such as those shown in Figure 2, and have brushing and linking enabled automatically between computed artefacts which depend on common data. At the core of this requirement is a variant of our original program analysis problem: we want to select part of the output and perform a backwards analysis to identify the required inputs, as before, but then also perform a forwards analysis to identify the dependent

parts of the other output. In Figure 2, these consist of the value of y for the record passed to `LinePlot` where x has the value 2015, for each `LinePlot` in the list passed to `LineChart`. Moreover, we would also like the brushing and linking feature to be able to provide a concise view of the data that explain why the two selections are linked. Note, however, that the intuition behind the forwards analysis here is not the same as the one we appealed to in the context of round-tripping: there the (hypothetical) question was whether the selected data was *sufficient* to reconstruct the selected output, whereas to identify related items in another view, we must determine those parts for which the selected data is *necessary*. As before, a language-based approach offers the prospect of addressing these sorts of question in a robust way.

1.3 Contributions

To make progress towards these challenges, we present a bidirectional analysis which tracks fine-grained data dependencies between input and output selections, with round-tripping properties characterised by Galois connections. Selections have a complement, which we use to adapt the analysis to compute fine-grained dependencies between two outputs which depend on common inputs. Recent program slicing techniques [Perera et al. 2012, 2016; Ricciotti et al. 2017] allow the user to focus on the output by “erasing” parts deemed to be irrelevant; the erased parts, called *holes*, are propagated backwards by a backwards analysis which identifies parts of the program and input which are no longer needed. Although these approaches also enjoy useful round-tripping properties characterised by Galois connections, they only allow focusing on *prefixes* (the portion of the output or program that remains after the irrelevant parts have been erased), a notion which is not closed under complement. Our specific contributions are as follows:

- a new bidirectional dynamic dependency analysis which operates on selections of arbitrary parts of data values, for a core calculus with lists, records and mutual recursion, and a proof that the analysis is a Galois connection (§ 3);
- a second bidirectional dependency analysis, derived from the first by De Morgan duality, which is also a Galois connection and which can be composed with the first analysis to link outputs to outputs, with an extended example based on matrix convolution (§ 4);
- a richer surface language called Fluid¹, implemented in PureScript, with familiar functional programming features such as piecewise definitions and list comprehensions, and a further Galois connection linking selections between the core and surface languages (§ 5).

Proofs and other supplementary materials can be found at <https://arxiv.org/abs/2109.00445>.

2 CORE LANGUAGE

The core calculus which provides the setting for the rest of the paper is a mostly standard call-by-value functional language with datatypes and records. The main unusual feature is the use of *eliminators*, a trie-like construct that provides a uniform syntax and semantics for pattern-matching; this allows us to assume that incomplete or overlapping patterns and other syntactic considerations have been dealt in the surface language. (In § 5 we show how familiar pattern-matching features like case expressions and piecewise function definitions easily desugar into eliminators.) We give a big-step environment-based semantics, which is easier for the backward and forward dependency analyses in § 3, and introduce a compact (term-like) representation of derivation trees in the operational semantics, called *traces*, which we will use to define the analyses over a fixed execution. Mutual recursion requires some care for the backwards analysis, so we also treat that as a core language feature.

¹See <https://github.com/explorables-viz/fluid/>. To generate the figures in this paper, check out tag v0.4.2 and follow the instructions in [artifact-evaluation.md](#).

	Type			Continuation type	
$A, B ::=$	Bool	Booleans	$K ::=$	A	term
	Int	integers		$A \multimap K$	eliminator
	$\text{Rec } (\vec{x}: \vec{A})$	records			
	List A	lists			
	$A \rightarrow B$	functions			
$\Gamma, \Delta ::=$	$\vec{x}: \vec{A}$	typing context			
	Term			Continuation	
$e ::=$	true false	Boolean	$\kappa ::=$	e	term
	n	integer		σ	eliminator
	x	variable			
	$\phi(\vec{e})$	primitive application			
	$e e'$	application			
	$[] u : v$	list			
	$(\vec{x}: \vec{e})$	record			
	$e.x$	record projection			
	$\lambda\sigma$	anonymous function			
	let h in e	recursive let			
$h ::=$	$\vec{x}: \vec{\sigma}$	recursive functions			
				Eliminator	
			$\sigma, \tau ::=$	$x: \kappa$	variable
				$\{\text{true}: \kappa, \text{false}: \kappa'\}$	Boolean
				$\{(\vec{x}): \kappa\}$	record
				$\{[]: \kappa, (:): \sigma\}$	list
				Value	
			$u, v ::=$	true false	Boolean
				n	integer
				$[] u : v$	list
				$(\vec{x}: \vec{v})$	record
				$\text{cl}(\rho, h, \sigma)$	closure
			$\rho ::=$	$\vec{x}: \vec{v}$	environment

Fig. 3. Syntax of core language

2.1 Syntax and Typing

Although our implementation is untyped, types help describe the structure of the core language. Figure 3 introduces the types A, B which include Bool, Int and function types $A \rightarrow B$, but also lists List A and records $\text{Rec } (\vec{x}: \vec{A})$ which exemplify the two kinds of structured data which are of interest: recursive datatypes with varying structure, and tabular data with a fixed shape. As usual the notation $x: A$ denotes the binding of x to A (understood formally as a pair); $\vec{x}: \vec{A}$ denotes the sequence of bindings that results from zipping same-length sequences \vec{x} and \vec{A} . In a record type $\text{Rec } (\vec{x}: \vec{A})$ the field names in \vec{x} are required to be unique.

The terms e of the language are defined in Figure 3. These include Boolean constants true and false, integers n , variables x , and applications $e e'$. Primitives are not first-class; the expression $\phi(\vec{e})$ is the fully saturated application of ϕ to a sequence of arguments. (First-class and infix primitives are provided by desugarings in § 5). We also provide list constructors nil $[]$ and cons $e : e'$, record construction $(\vec{x}: \vec{e})$ and record projection $e.x$. The final two term forms, anonymous functions $\lambda\sigma$ and recursive let-bindings let h in e where h is of the form $\vec{x}: \vec{\sigma}$, are explained below after we introduce the pattern-matching construct σ (eliminator). The typing rules for terms are given in Figure 4, and are intended only to help a reader understand the language; therefore the rules are simple and do not include features such as polymorphism. The main typing rules of interest are the ones which involve eliminators.

2.2 Eliminators

Eliminators σ, τ are also defined in Figure 3, and are essentially generalised tries [Connelly and Morris 1995; Hinze 2000] extended with variable binding. An eliminator specifies how to match an initial part of a value and select a continuation κ for further execution; κ may be a term e , or another eliminator σ . The Boolean eliminator $\{\text{true}: \kappa, \text{false}: \kappa'\}$ selects either κ or κ' depending on whether a Boolean value is true or false. The record eliminator $\{(\vec{x}): \kappa\}$ matches a record with

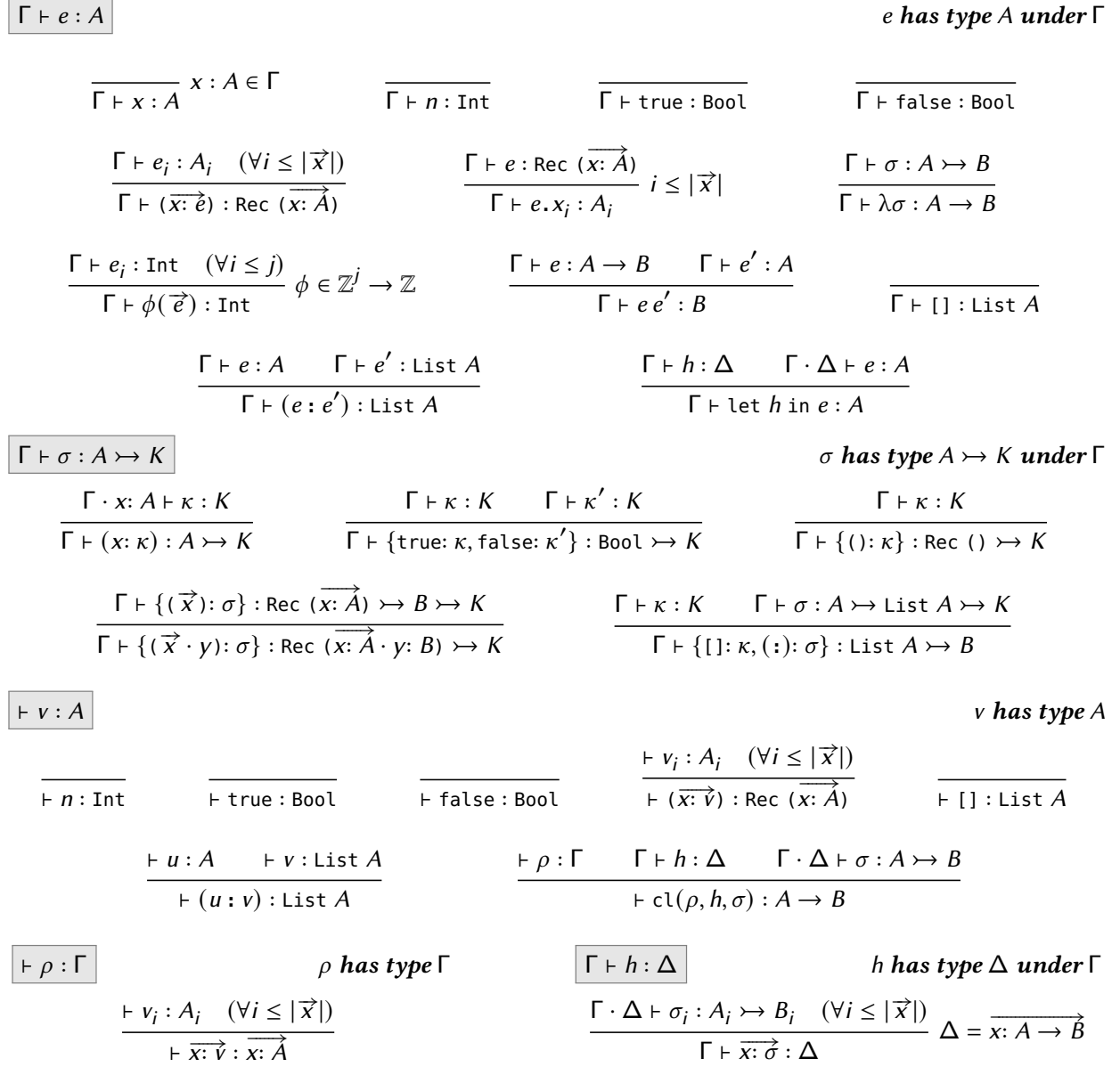


Fig. 4. Typing rules for core language

fields \vec{x} and then selects κ with the variables \vec{x} bound to the components of the record. The list eliminator $\{[] : \kappa, (:) : \sigma\}$ selects κ if the list is empty and otherwise defers to another eliminator σ which specifies how the head and tail of the list are to be matched. Finally, the variable eliminator $x : \kappa$ extends the usual notion of trie, matching any value, and selecting κ with x bound to that value. Eliminators resemble the “case trees” commonly used as an intermediate form when compiling languages with pattern-matching [Graf et al. 2020], and can serve as an elaboration target for more user-oriented features such as the piecewise definitions described in § 5.

The use of nested eliminators to match sub-values will become clearer if we consider the typing judgement $\Gamma \vdash \sigma : A \multimap K$ given in Figure 4. Eliminators always have a function-like type; the judgement form should be read as a four-place relation, with \multimap being part of the notation. (The definition delegates to an auxiliary judgement $\Gamma \vdash \kappa : K$ which we define to be the union of the $\Gamma \vdash e : A$ and $\Gamma \vdash \sigma : A \multimap K$ relations.) The typing rule for variable eliminators reveals

the connection between eliminators and functions: it converts a continuation κ which can be assigned type K under the assumption that x is of type A into an eliminator of type $A \multimap K$. The typing rule for Boolean eliminators says that to make an eliminator of type $\text{Bool} \multimap K$, we simply need continuations κ and κ' of type K . The rule for the empty record states that to make an eliminator of type $\text{Rec } () \multimap K$, we simply need a continuation κ of type K . The rule for non-empty records allows us to transform a “curried” eliminator of type $\text{Rec } (x: \vec{A}) \multimap B \multimap K$ into one of type $\text{Rec } (x: \vec{A} \cdot y: B) \multimap K$, analogous to the isomorphism between $A \rightarrow B \rightarrow C$ and $A \times B \rightarrow C$ [Hinze 2000]. (Formalising eliminators precisely requires nested datatypes [Bird and Meertens 1998] and polymorphic recursion, but these details need not concern us here.)

The typing rule for list eliminators $\{[]: \kappa, (:): \sigma\}$ combines some of the flavour of record and Boolean eliminators. To make an eliminator of type $\text{List } A \multimap K$, we need a continuation of type K for the empty case, and for the non-empty case, an eliminator of type $A \multimap \text{List } A \multimap K$ which will be used to process the head and tail.

2.2.1 Functions as Eliminators. We can now revisit the term forms $\lambda\sigma$ and $\text{let } h \text{ in } e$. If σ is an eliminator of type $A \multimap B$, then $\lambda\sigma$ is an anonymous function of type $A \rightarrow B$. If h is of the form $\vec{x}: \vec{\sigma}$, then $\text{let } h \text{ in } e$ introduces a sequence of mutually recursive functions which are in scope in e . The typing rule for $\text{let } h \text{ in } e$ uses an auxiliary typing judgement $\Gamma \vdash h : \Delta$ which assigns to every x in Δ the function type $A \rightarrow B$ if the σ to which x is bound in h has the eliminator type $A \multimap B$.

2.2.2 Values. Values v , u , and environments ρ are also defined in Figure 3, and are standard for call-by-value. (Environments are more convenient than substitution for tracking variable usage.) To support mutual recursion, the closure form $\text{cl}(\rho, h, \sigma)$ captures the (possibly empty) sequence h of functions with which the function was mutually defined, in addition to the ambient environment ρ . For the typing judgements $\vdash \rho : \Gamma$ and $\vdash v : A$ for environments and values (Figure 4), only the closure case is worth noting, which delegates to the typing rules for recursive definitions and eliminators.

2.2.3 Evaluation. Figure 6 gives the operational semantics of the core language. In § 3 we will define forward and backward analyses over a single execution; in anticipation of that use case, we treat the operational semantics as an inductive data type, following the “proved transitions” approach adopted by Boudol and Castellani [1989] for reversible CCS. The inhabitants of this data type are derivation trees explaining how a result was computed, and the analyses will be defined by structural recursion over these trees. Expressed in terms of inference rules, these trees can become quite cumbersome, so we introduce an equivalent but more term-like syntax for them, called a *trace* (Figure 5), similar to the approach taken by Perera et al. [2016] for π -calculus.

Trace				
$T, U ::=$	x	variable	$\phi(\vec{U}_n)$	primitive application
	$\text{true} \mid \text{false}$	Boolean	$\text{let } h \text{ in } T$	recursive let
	n	integer		
	$(x: \vec{T})$	record	Match	
	$T_{\vec{x}: \vec{v}} \cdot Y$	record projection	$w ::=$	x
	$[] \mid T : U$	list		$\text{true} \mid \text{false}$
	$\lambda\sigma$	anonymous function		$(\vec{x}: \vec{w})$
	$T U \blacktriangleright w: T'$	application		$[] \mid w: w'$
				list

Fig. 5. Syntax of traces and matches

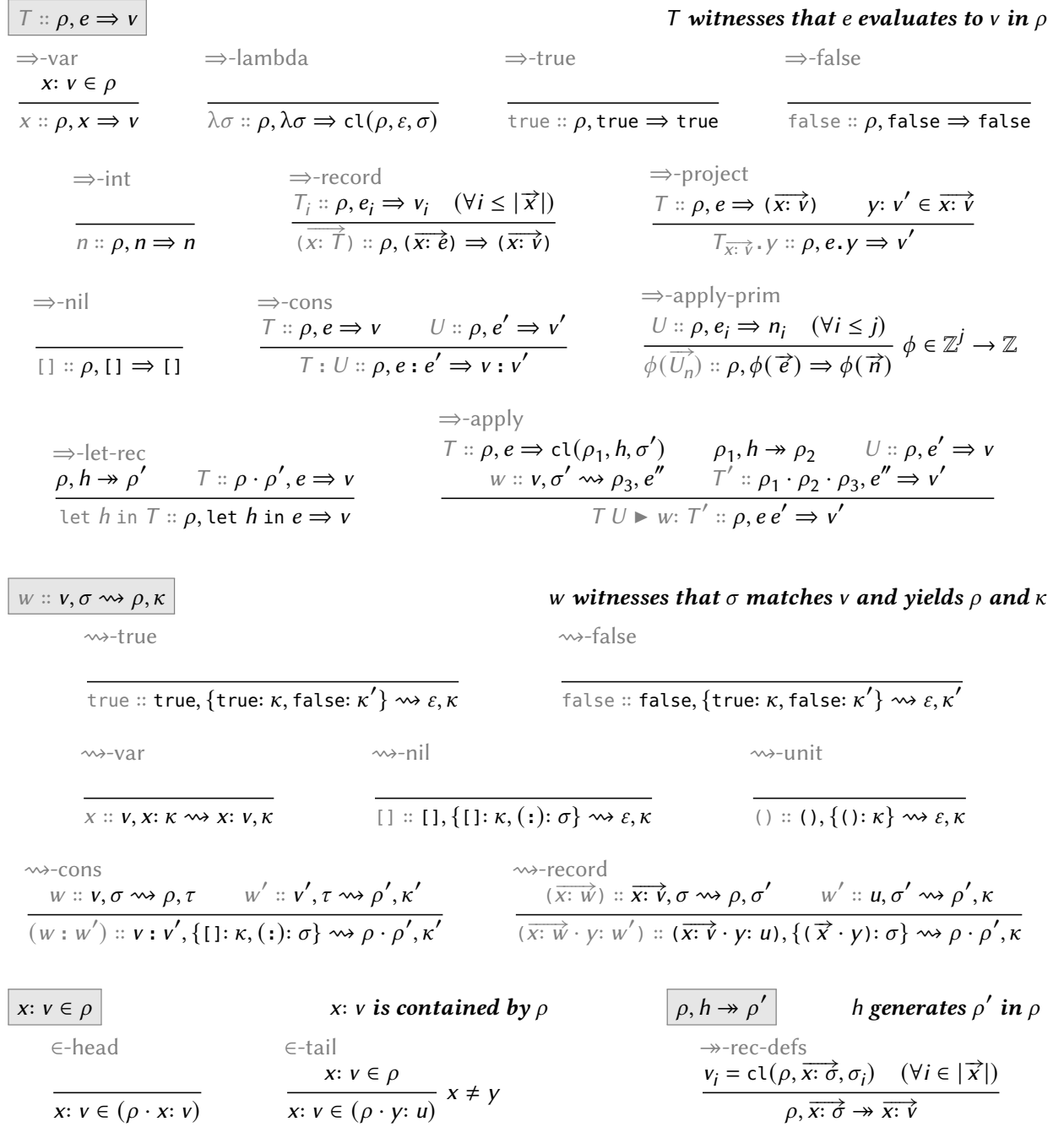


Fig. 6. Operational semantics

The judgement $T :: \rho, e \Rightarrow v$ defined at the top of Figure 6 states that term e under environment ρ evaluates to value v , and that T is a proof term that witness that fact. (In the figure, the traces appear in grey, to reinforce the idea that they are not part of the definition of \Rightarrow but rather a notation for its inhabitants.) The rules for Booleans, integers and lists are standard and have unsurprising trace forms. For variables, we give an explicit inductive definition of the environment lookup relation \in at the bottom of the figure, again so that later we can perform analysis over a proof that an environment contains a binding. The lambda rule is standard except that we specify ε for the sequence of definitions being simultaneously defined, since a lambda is not recursive. For record

construction, the trace form contains a subtrace T_i for each field, and for record projection, which also uses the lookup relation \in , the trace form $T_{\overline{x:\vec{v}}.y}$ records both the record $\overline{x:\vec{v}}$ and the field name y that was selected.

The rule for (mutually) recursive functions let h in e , where h is a sequence $\overline{x:\vec{\sigma}}$ of function definitions, makes use of the auxiliary relation $\rho, h \rightarrow \rho'$ at the bottom of Figure 6 which turns h into an environment ρ' binding each function name x_i to a closure $\text{cl}(\rho, h, \sigma_i)$ capturing ρ and a copy of h . For primitive applications, the trace records the values of the arguments which were passed to the operation ϕ . The rule for application $e e'$ is slightly non-standard, because it must deal with both mutual recursion and pattern-matching. First we unpack the recursive definitions h from the closure $\text{cl}(\rho_1, h, \sigma)$ computed by e , and again use the auxiliary relation \rightarrow to promote this into an environment ρ_2 of closures. We then use the relation \rightsquigarrow explained below to match v against the eliminator σ , obtaining the branch e'' of the function to be executed and parameter bindings ρ_3 . In addition to subtraces T and U for the function and argument, the application trace $T U \blacktriangleright w: T'$ also has subtraces w for the pattern-match and T' for the selected branch.

2.2.4 Pattern Matching. The judgement $w :: v, \sigma \rightsquigarrow \rho, \kappa$ also defined in Figure 6 states that eliminator σ can match v and produce environment ρ and continuation κ , with ρ containing the variable bindings that arose during the match. *Matches* w are a compact notation for proof terms for the \rightsquigarrow relation, analogous to traces for the \Rightarrow relation, and again appear in grey in the figure.

Variable eliminators $x: \kappa$ match any value, returning the singleton environment $x: v$ and continuation κ . Boolean eliminators match any Boolean value, returning the appropriate branch and empty environment ε . List eliminators $\{[]: \kappa, (:): \sigma\}$ match any list. The nil case is analogous to the handling of Booleans; the cons case depends on the fact that the nested eliminator σ for the cons branch has the curried type $A \mapsto \text{List } A \rightarrow K$. First, we recursively match the head v of type A using σ , obtaining bindings ρ and eliminator $\tau: \text{List } A \mapsto K$ as the continuation. Then the tail v' is matched using τ to yield additional bindings ρ' and final continuation κ' of type K . As a simple example, which omits the proof terms w , consider the following pattern-match:

$$\rightsquigarrow\text{-cons} \frac{\rightsquigarrow\text{-var} \frac{}{5, x: xs: e_2 \rightsquigarrow x: 5, xs: e_2} \quad \rightsquigarrow\text{-var} \frac{}{6: [], xs: e_2 \rightsquigarrow xs: (6: []), e_2}}{5: 6: [], \{[]: e_1, (:): x: xs: e_2\} \rightsquigarrow (x: 5) \cdot (xs: 6), e_2}$$

Here the eliminator $\{[]: e_1, (:): x: xs: e_2\}$ is used to match $5: 6: []$. The $[]$ case is disregarded; the $(:)$ case is used to retrieve a variable eliminator $x: xs: e_2$, which is used to match the head 5. This produces the binding $x: 5$ and a further variable eliminator $xs: e_2$ as the continuation, which is used to match the tail. This produces the additional binding $xs: (6: [])$ and the expression e_2 as the continuation. To see how this might generalise to a nested pattern, consider how one could replace the inner variable eliminator $xs: e_2$ by another list eliminator.

Record matching is similar: the empty record case resembles the nil case, and the non-empty case relies on the nested eliminator having curried type $\text{Rec } (\overline{x:\vec{A}}) \mapsto B \mapsto K$. The initial part $\overline{x:\vec{v}}$ of the record is matched using σ , returning another eliminator σ' of type $B \mapsto K$. Then the last field $y: u$ is matched using σ' to yield final continuation κ of type K .

3 A BIDIRECTIONAL DYNAMIC DEPENDENCY ANALYSIS

We now extend the core language from § 2 with a bidirectional mechanism for tracking data dependencies. § 3.1 establishes a way of selecting (parts of) values, such as the height of a bar in a bar chart. § 3.2 defines a forward analysis function \mathcal{A}_T which specifies how selections on programs and environments (collectively: *input selections*) become selections on outputs; selections represent *availability*, with the computed output selection indicating the data available to the downstream

computation. § 3.3 defines a backward dependency function \bowtie_T specifying how output selections are mapped back to inputs; then selections represent demands, with the computed input selection identifying the data needed from the upstream computation. Both functions are monotonic. This will become important in § 3.4, where we show that \bowtie_T and \nearrow_T form a *Galois connection*, establishing the round-tripping properties alluded to in § 1.1.

3.1 Lattices of Selections

Our approach to representing selections is shown in Figure 7. The basic idea is to parameterise the type Val of values by a (bounded) lattice \mathcal{A} of *selection states* α . We add selection states to Booleans, integers, records and lists; while it would present no complications to equip closures with selection states too, for present purposes we are only interested in dependencies between first-order data, so closures are not (directly) selectable. Closures do however have selectable parts, and moreover capture the current *argument availability*, explained in § 3.2.2 below, which is also a selection state α . We parameterise the type Term of terms similarly, allowing us to trace data dependencies back to expressions that appear in the source code, but only add selection states to the term constructors corresponding to selectable values. We return to this in § 5.

$e \in \text{Term } \mathcal{A} ::=$	Terms selections	$u, v \in \text{Val } \mathcal{A} ::=$	Value selections
...	...	\square	hole
\square	hole	$\text{true}_\alpha \mid \text{false}_\alpha$	Boolean
$\text{true}_\alpha \mid \text{false}_\alpha$	Boolean	n_α	integer
n_α	integer	$(\vec{x} : \vec{v})_\alpha$	record
$(\vec{x} : \vec{e})_\alpha$	record	$[\]_\alpha \mid u :_\alpha v$	list
$[\]_\alpha \mid e :_\alpha e'$	list	$\text{cl}(\rho, h, \alpha, \sigma)$	closure
$\alpha, \beta \in \mathcal{A}$	selection state		

Fig. 7. Selection states, term selections and value selections

The top and bottom elements \top and \perp of \mathcal{A} represent fully selected and fully unselected; the meet and join operations \sqcap and \sqcup , which have \top and \perp as their respective units, are used to combine selection information. In Figure 1, the data field of `BarChart` expects a list of records with fields `x` and `y`, mapping strings representing categorical data to floats determining the height of the corresponding bar; the record computed for China is $(x: \text{"China"} \cdot y: 295.3)$. The two-point lattice $2 \stackrel{\text{def}}{=} \{\text{tt}, \text{ff}\}, \text{tt}, \text{ff}, \wedge, \vee$ can be used to represent the selection of the field `y` within this record as $(x: \text{"China"}_{\text{ff}} \cdot y: 295.3_{\text{tt}})_{\text{ff}}$, indicating that the number 295.3 is selected, but that neither the string "China", nor the record itself, is selected. Because lattices are closed under component-wise products, we sometimes write $(\alpha, \beta) \sqsubseteq (\alpha', \beta')$ to mean that $\alpha \sqsubseteq \alpha'$ and $\beta \sqsubseteq \beta'$. This also suggests more interesting lattices of selections, such as vectors of Booleans to represent multiple selections simultaneously, which might be visualised using different colours (as in Figure 1).

3.1.1 Selections of a Value. The analyses which follow will be defined with respect to a fixed computation, and so we will often need to talk about the selections of a given value. To make this notion precise, consider that the raw (selection-free) syntax described in § 2 can be recovered from a term selection via an erasure operation $[\cdot] : \text{Val } \mathcal{A} \rightarrow \text{Val } 1$ which forgets the selection information, where 1 is the trivial one-point lattice. We refer to $[\nu]$ as the *shape* of ν . Allowing \mathbf{u}, \mathbf{v} from now on to range over raw values, and reserving u, v for value selections, we can then define:

Definition 3.1 (Selections of \mathbf{v}). Define $\text{Sel}_{\mathbf{v}} \mathcal{A}$ to be the set of all values $\nu \in \text{Val } \mathcal{A}$ with shape \mathbf{v} , i.e. that erase to \mathbf{v} .

$$\boxed{v \sqsubseteq v'}$$

$$\begin{array}{c}
\frac{}{\square \sqsubseteq v} \quad \frac{\alpha \sqsubseteq \alpha'}{n_\alpha \sqsubseteq n_{\alpha'}} \quad \frac{}{n_\perp \sqsubseteq \square} \quad \frac{\alpha \sqsubseteq \alpha'}{\text{true}_\alpha \sqsubseteq \text{true}_{\alpha'}} \quad \frac{}{\text{true}_\perp \sqsubseteq \square} \quad \frac{\alpha \sqsubseteq \alpha'}{\text{false}_\alpha \sqsubseteq \text{false}_{\alpha'}} \\
\frac{}{\text{false}_\perp \sqsubseteq \square} \quad \frac{\alpha \sqsubseteq \alpha' \quad v_i \sqsubseteq u_i \quad (\forall i \in |\vec{x}|)}{(\vec{x}:\vec{v})_\alpha \sqsubseteq (\vec{x}:\vec{u})_{\alpha'}} \quad \frac{v_i \sqsubseteq \square \quad (\forall i \in |\vec{x}|)}{(\vec{x}:\vec{v})_\perp \sqsubseteq \square} \quad \frac{\alpha \sqsubseteq \alpha'}{[]_\alpha \sqsubseteq []_{\alpha'}} \quad \frac{}{[]_\perp \sqsubseteq \square} \\
\frac{(\alpha, v, v') \sqsubseteq (\alpha', v, v')}{v :_\alpha v' \sqsubseteq u :_{\alpha'} u'} \quad \frac{(v, v') \sqsubseteq (\square, \square)}{v :_\perp v' \sqsubseteq \square} \quad \frac{(\rho, h, \alpha, \sigma) \sqsubseteq (\rho', h', \alpha', \sigma')}{\text{cl}(\rho, h, \alpha, \sigma) \sqsubseteq \text{cl}(\rho', h', \alpha', \sigma')} \quad \frac{(\rho, h, \sigma) \sqsubseteq (\square_\rho, \square, \square)}{\text{cl}(\rho, h, \perp, \sigma) \sqsubseteq \square}
\end{array}$$

Fig. 8. Preorder on value selections

Since its elements have a fixed shape, the pointwise comparison of any $v, v' \in \text{Sel}_v \mathcal{A}$ using the partial order \sqsubseteq of \mathcal{A} is well defined, as is the pointwise application (zip) of a binary operation [Gibbons 2017]. It should therefore be clear that if \mathcal{A} is a lattice, then $\text{Sel}_v \mathcal{A}$ is also a lattice, with \top_v , \perp_v , \sqcap_v , and \sqcup_v defined pointwise. For example, if u and u' have the same shape and v and v' have the same shape, the join of the lists $(u :_\alpha v)$ and $(u' :_{\alpha'} v')$ is defined and equal to $(u \sqcup u') :_{\alpha \sqcup \alpha'} (v \sqcup v')$. Similarly, the top element of $\text{Sel}_v \mathcal{A}$ is the selection of \mathbf{v} which has \top at every selection position. (We omit the \mathbf{v} indices from these lattice operations if it is clear which lattice is being referred to.) The notion of the “selections” of \mathbf{v} extends to the other syntactic forms.

3.1.2 Environment Selections and Hole Equivalence. The notion of the “selections” of \mathbf{v} also extends (pointwise) to environments, so that $\text{Sel}_\rho \mathcal{A}$ means the set of environment selections ρ' of shape ρ , where the variables in ρ' are bound to selections of the corresponding variables in ρ . One challenge arises from the pointwise use of \sqcup to combine environment selections. Since environments contain other environments recursively, via closures, a naive implementation of environment join is a very expensive operation. One solution is to employ an efficient representation of values which are fully unselected, which is often the case during the backward analysis.

We therefore augment the set of value selections $\text{Val} \mathcal{A}$ with a distinguished element *hole*, written \square , which is an alternative notation for \perp_v for any \mathbf{v} , i.e. the selection of shape \mathbf{v} which has \perp at every selection position, and generalise this idea to terms and eliminators. The equivalence of \square to any such bottom element is established explicitly by the preorder order defined (for values) in Figure 8: the first rule always allows \square on the left-hand side of \sqsubseteq , and other rules allow \square on the right-hand side of \sqsubseteq as long as all the selections that appear on the left-hand side are \perp . (The rules for terms e and eliminators σ are analogous and are omitted.) If we write \doteq for the equivalence relation induced by \sqsubseteq on values selections, which we call *hole-equivalence*, it should be clear that $\square \sqcup v \doteq v$ and $\square \sqcap v \doteq \square$. This means the join of two selections v, v' of \mathbf{v} can be implemented efficiently, whenever one selection is \square , by simply discarding \square and returning the other selection without further processing.

Definition 3.2 (Hole equivalence). Define \doteq as the intersection of \sqsubseteq and \supseteq .

Because \square is equivalent to \perp_v for any \mathbf{v} , all such bottom elements are hole-equivalent. For example, the value selection $\square :_\top \square$ is hole-equivalent to $5_\perp :_\top \square$, but also to $6_\perp :_\top []_\perp$, and so the last two selections, even though they have different shapes, are hole-equivalent by transitivity. In practice we only use the hole ordering to compare selections with the same shape.

3.2 Forward Data Dependency

We now define the core bidirectional data dependency analyses for a fixed computation $T :: \rho, \mathbf{e} \Rightarrow \mathbf{v}$, where T is a trace. In practice one would obtain T by first evaluating \mathbf{e} in ρ , and then run multiple forward or backward analyses over T with appropriate lattices. We start with the forward dependency function \mathcal{A}_T which “replays” evaluation, turning input availability into output availability, with T guiding the analysis whenever holes in the input selection would mean the analysis would otherwise get stuck. \mathcal{A}_T uses the auxiliary function $\mathcal{A}_w^\triangleright$ for forward-analysing a pattern-match; we explain this first, as it introduces the key idea of a selection as identifying the data available to a downstream computation.

3.2.1 Forward Match. Figure 9 defines a family of *forward-match* functions $\mathcal{A}_w^\triangleright$ of type $\text{Sel}_{\mathbf{v}, \sigma} \mathcal{A} \rightarrow (\text{Sel}_{\rho, \kappa} \mathcal{A}) \times \mathcal{A}$ for any $w :: \mathbf{v}, \sigma \rightsquigarrow \rho, \kappa$. (The definition is presented in a relational style for readability, but should be understood as a total function defined by structural recursion on w , which appears in grey to emphasise the connection to Figure 6.) Forward match replays the match witnessed by w , turning availability $(\mathbf{v}, \sigma) \in \text{Sel}_{\mathbf{v}, \sigma} \mathcal{A}$ on the matched value and eliminator into availability $(\rho, \kappa) \in \text{Sel}_{\rho, \kappa} \mathcal{A}$ on the variable bindings and continuation yielded by the match.

$\mathcal{A}_w^\triangleright$ also returns the *meet* of the selection states associated with the part of \mathbf{v} which was matched by σ . We call this the *argument availability*, since it represents the availability of the matched part of a function argument. In the variable case, the empty part of \mathbf{v} was matched and so the argument availability in this context is simply \top , the unit for \sqcap . In the Boolean case, the argument availability is simply the α on true_α or false_α ; the empty list and empty record cases are similar. In the cons case, we return the meet of the α on the cons node itself with the availabilities β and β' computed for v and v' . Non-empty records are similar, but to process the initial part of the record, we supply the neutral selection state \top on the subrecord in order to use the definition recursively. (Note that these subrecords exist only as intermediate artefacts of the interpreter.)

One might hope to be able to dispense with the match witness w and simply define $\mathcal{A}^\triangleright$ by case analysis on v and σ . However, it is then unclear how to proceed in the event that either v or σ is a hole. In particular, it is not clear how to obtain the ρ associated with the original pattern-match in order to produce an environment selection $\rho' \in \text{Sel}_\rho \mathcal{A}$. If $\mathcal{A}^\triangleright$ is defined with respect to a known w , this can be achieved via additional rules $\mathcal{A}^\triangleright\text{-hole-}v$ and $\mathcal{A}^\triangleright\text{-hole-}\sigma$ that define the behaviour at hole to be the same as the behaviour at any \doteq -equivalent value in $\text{Sel}_v \mathcal{A}$ or $\text{Sel}_\sigma \mathcal{A}$.

Operationally, these hole rules can be interpreted as “expanding” the holes in v or σ , in a shape-preserving way, until another rule of the definition applies. Recall the pattern-matching example from § 2.2.4. This pattern-match has the witness $x : xs$, recording that the list $5 : 6 : []$ was matched to the depth of a single cons. Suppose we wish to forward-analyse over the pattern-match using \sqcap to represent the selection on the matched list. The information in the match witness allows us to expand \sqcap to $\sqcap : \perp \sqcap$ and then use the $\mathcal{A}^\triangleright\text{-cons}$ rule to derive the following forward-match:

$$\mathcal{A}^\triangleright\text{-hole-}v \frac{\mathcal{A}^\triangleright\text{-cons} \frac{\mathcal{A}^\triangleright\text{-var} \frac{}{\sqcap, x : xs : e_2 \quad \mathcal{A}_x^\triangleright \quad x : \sqcap, xs : e_2, \top} \quad \mathcal{A}^\triangleright\text{-var} \frac{}{\sqcap, xs : e_2 \quad \mathcal{A}_{xs}^\triangleright \quad xs : \sqcap, e_2, \top}}{\sqcap : \perp \sqcap, \{[] : e_1, (:): x : xs : e_2\} \quad \mathcal{A}_{x : xs}^\triangleright \quad (x : \sqcap) \cdot (xs : \sqcap), e_2, \perp}}{\sqcap, \{[] : e_1, (:): x : xs : e_2\} \quad \mathcal{A}_{x : xs}^\triangleright \quad (x : \sqcap) \cdot (xs : \sqcap), e_2, \perp}}$$

Lemma 3.3 below implies that an implementation is free to replace any term by a hole-equivalent one of the same shape, with the result of $\mathcal{A}_w^\triangleright$ being unique up to \doteq . This justifies the strategy of expanding holes just enough for a non-hole rule to apply; there will be exactly one such rule, corresponding to the execution path originally taken, and although there may be multiple possible

$v, \sigma \vDash_w \rho, \kappa, \alpha$			v and σ forward-match along w to ρ and κ , with argument availability α		
\vDash -hole- v	\vDash -hole- σ	\vDash -var			
$\frac{\square \doteq v \quad v, \sigma \vDash_w \rho, \kappa, \alpha}{\square, \sigma \vDash_w \rho, \kappa, \alpha}$	$\frac{\square \doteq \sigma \quad v, \sigma \vDash_w \rho, \kappa, \alpha}{v, \square \vDash_w \rho, \kappa, \alpha}$	$\frac{}{v, x: \kappa \vDash_x x: v, \kappa, \top}$			
\vDash -true	\vDash -false				
$\frac{}{\text{true}_\alpha, \{\text{true}: \kappa, \text{false}: \kappa'\} \vDash_{\text{true}} \varepsilon, \kappa, \alpha}$	$\frac{}{\text{false}_\alpha, \{\text{true}: \kappa, \text{false}: \kappa'\} \vDash_{\text{false}} \varepsilon, \kappa', \alpha}$				
\vDash -unit	\vDash -record				
$\frac{}{()_\alpha, \{(): \kappa\} \vDash_{()} \varepsilon, \kappa, \alpha}$	$\frac{(\vec{x}: \vec{v})_\top, \{(\vec{x}): \sigma\} \vDash_{(\vec{x}: \vec{w})} \rho, \sigma', \beta \quad u, \sigma' \vDash_w \rho', \kappa, \beta'}{(\vec{x}: \vec{v} \cdot y: u)_\alpha, \{(\vec{x} \cdot y): \sigma\} \vDash_{(\vec{x}: \vec{w} \cdot y: w')} \rho \cdot \rho', \kappa, \alpha \sqcap \beta \sqcap \beta'}$				
\vDash -nil	\vDash -cons				
$\frac{}{[]_\alpha, \{[]: \kappa, (:): \sigma'\} \vDash_{[]} \varepsilon, \kappa, \alpha}$	$\frac{v, \sigma \vDash_w \rho, \tau, \beta \quad v', \tau \vDash_w \rho', \kappa', \beta'}{v: \alpha v', \{[]: \kappa, (:): \sigma\} \vDash_{w: w'} \rho \cdot \rho', \kappa', \alpha \sqcap \beta \sqcap \beta'}$				

Fig. 9. Forward match

expansions, they will produce hole-equivalent results. This also explains why it is reasonable to think of \vDash_w not just as a relation, but as a function.

LEMMA 3.3 (MONOTONICITY OF \vDash_w). *Suppose $w :: \mathbf{v}, \sigma \rightsquigarrow \rho, \kappa$, with $v, \sigma \vDash_w \rho, \kappa, \alpha$ and $v', \sigma' \vDash_w \rho', \kappa', \alpha'$. If $(v, \sigma) \sqsubseteq (v', \sigma')$ then $(\rho, \kappa, \alpha) \sqsubseteq (\rho', \kappa', \alpha')$.*

The forward-match function \vDash_w is a key component of the forward evaluation function \nearrow_T defined in § 3.2.2 below. When forward-analysing a function call, the argument is forward-matched using \vDash_w , and the resulting argument availability α used to upper-bound the availability of any partial values constructed by that function, establishing a forward link from resources consumed and resources produced. Since the dynamic context of a function call extends over multiple evaluation steps, \nearrow_T is threaded with an additional input α which tracks the active argument availability; at the outermost level, before there are any active function calls, this has the value \top .

3.2.2 Forward Evaluation. Figure 10 defines a family of *forward-evaluation* functions \nearrow_T of type $(\text{Sel}_{\rho, e} \mathcal{A}) \times \mathcal{A} \rightarrow \text{Sel}_v \mathcal{A}$ for any $T :: \rho, \mathbf{e} \Rightarrow \mathbf{v}$. (Like forward match, forward evaluation is presented in a relational style, but should be read as a total function defined by structural recursion on T .) Forward evaluation replays T , taking a selection $(\rho, e) \in \text{Sel}_{\rho, e} \mathcal{A}$ identifying the available parts of the environment and program, and an $\alpha \in \mathcal{A}$ representing the argument availability for the dynamically innermost function call, and returning a selection $v \in \text{Sel}_v \mathcal{A}$ identifying the outputs that can be produced using only the available resources. The rules resemble those for the evaluation relation \Rightarrow . The general pattern is that each rule takes the active argument availability α , combines it (using \sqcap) with any availability supplied on the expression form consumed at that step, and uses the result as the availability of any partial values constructed at that step. The argument availability α is passed down unchanged to any subcomputations, except in the case of function application.

Function application. In the application case, the rule must determine a new argument availability for the function body, because the function context is changing. First, we unpacks the β stored in the closure, representing the argument availability which was active when the closure was constructed. Then we determine an additional selection state β' , representing the availability of

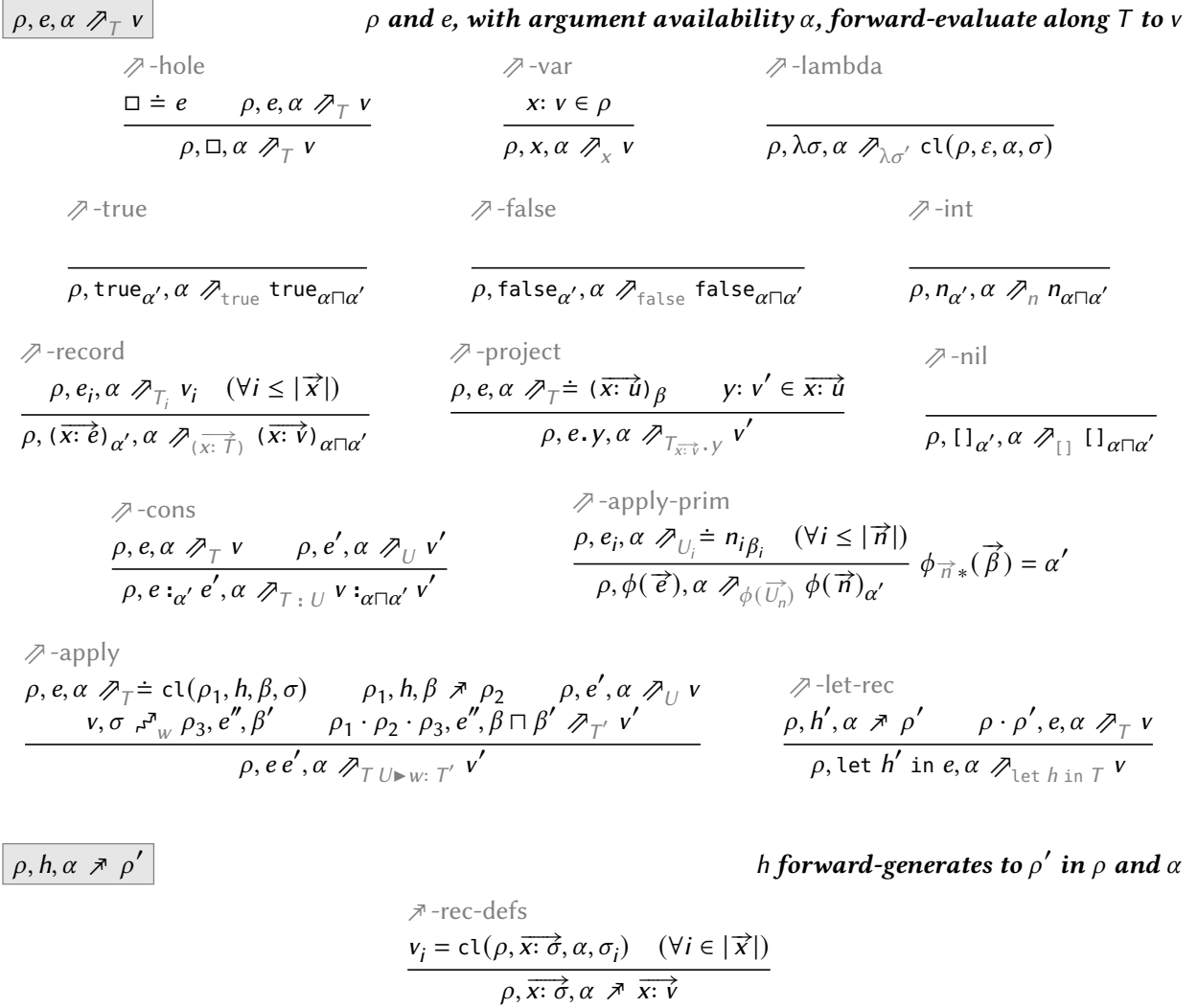


Fig. 10. Forward evaluation

the matched part of the current argument, by forward-matching v with the eliminator σ from the closure. These are combined using \cap to represent the conjoined availability of all arguments that were pattern-matched in order to execute the function body, and the result $\beta \cap \beta'$ used to forward-evaluate the function body. The auxiliary function $\not\Rightarrow_{\rho, h}: (\text{Sel}_{\rho, h} \mathcal{A}) \times \mathcal{A} \rightarrow \text{Sel}_{\rho'} \mathcal{A}$ for any $\rho, h \rightarrow \rho'$ is given at the bottom of Figure 10 and resembles \rightarrow , but captures the active argument availability into each closure.

Primitive application. Since primitive operations are opaque, their input-output dependencies cannot be derived from their execution, but must be supplied by the primitive operation itself. More specifically, every primitive $\phi \in \text{Int}^i \rightarrow \text{Int}$ is required to provide a forward-dependency function $\phi_{\vec{n}^*}: \mathcal{A}^i \rightarrow \mathcal{A}$ for every $\vec{n} \in \text{Int}^i$ which specifies how to turn an input selection $\vec{\alpha} \in \mathcal{A}^i$ for \vec{n} into an output selection α' on $\phi(\vec{n})$. There is one such function per possible input \vec{n} so that the dynamic dependencies for that specific call can depend on the values passed to the operation. For example, in our implementation, the dependency function for multiplication establishes (for non-zero n) that both $n * 0$ and $0 * n$ depend only on 0. However, primitives are free to implement forward-dependency however they choose, with the caveat that § 3.3.2 will also require ϕ to provide

a backward-dependency function for any input \vec{n} , and § 3.4 will require these to be related in a certain way for the consistency of the whole system to be guaranteed.

Other rules. The remaining rules follow the general pattern. Variable lookup disregards α , simply preserving the selection on the value extracted from the environment. The lambda rule captures α in the closure along with the environment; the letrec rule passes α on to \nearrow so it can be captured by recursive closures as well. Record projection is more interesting, disregarding not only the argument availability α but also the availability β of the record itself. This is because containers are considered to be independent of the values they contain: here, v_i has its own internal availability which is preserved by projection, but there is no implied dependency of the field on the record from which it was projected. Record construction also reflects this principle, preserving the field selections unchanged into the resulting record selection. But since this rule also constructs a partial value – the record itself – it must specify an availability on that output. The availability is set to $\alpha \sqcap \alpha'$, reflecting the dependency of the constructed container on both the constructing expression and the active argument match. The rules for nil, cons, integers and Booleans are similar, since they also construct values.

Hole cases. Environments have no special \square form. However, a hole rule is needed to allow forward evaluation to continue in the event that e is \square ; this is essential because subsequent steps may result in non- \square outputs (for example by extracting non- \square values from ρ). The rule is similar to the hole rules for \nearrow_w and again can be understood operationally as using the information in T to expand \square sufficiently for another rule to apply, with a result which is unique up to \doteq . In addition, application and record projection must accommodate the case where the selection on the closure or record being eliminated is represented by \square . In these rules $\nearrow_T \doteq$ is used to denote the relational composition of \nearrow_T and \doteq .

LEMMA 3.4 (MONOTONICITY OF \nearrow_T). *Suppose $T :: \rho, e \Rightarrow v$ with $\rho, e, \alpha \nearrow_T v$ and $\rho', e', \alpha' \nearrow_T v'$. If $(\rho, e, \alpha) \sqsubseteq (\rho', e', \alpha')$ then $v \sqsubseteq v'$.*

3.3 Backward Data Dependency

The backward dependency function \searrow_T “rewinds” evaluation, turning output demand into input demand, with T guiding the analysis backward. We start with the auxiliary function \searrow_w which is used for backward-analysing a pattern-match.

3.3.1 Backward Match. Figure 11 defines a family of *backward-match* functions \searrow_w of type $(\text{Sel}_{\rho, \kappa} \mathcal{A}) \times \mathcal{A} \rightarrow \text{Sel}_{v, \sigma} \mathcal{A}$ for any $w :: v, \sigma \rightsquigarrow \rho, \kappa$. Backward-match rewinds the match witnessed by w , turning demand on the environment and continuation into demand on the value and eliminator that were originally matched. The additional input α represents the downstream demand placed on any resources that were constructed in the context of this match; \searrow_w transfers this to the matched portion of v , establishing a backwards link from resources produced to resources consumed in a given function context. We call α the *argument demand* since it represents the demand to be pushed backwards onto the matched part of a function argument.

In the variable case, the empty part of v was matched, so α is disregarded. The rule need only ensure that the demand v in the singleton environment $x: v$ is propagated backward. If a Boolean constant was matched, α becomes the demand on that constant, and κ , capturing the demand on the continuation, is used to construct the demand on the original eliminator, with \square used to represent the absence of demand on the non-taken branch. (This use of \square explains why matches w need only retain information about taken branches.) The nil case is similar.

For a cons match $w : w'$, we split the environment into ρ and ρ' , using the fact that there is a unique well-typed decomposition. We then backward-match w and w' recursively to obtain v and v' , representing the demand on the head and tail of the list. These are combined into the demand on

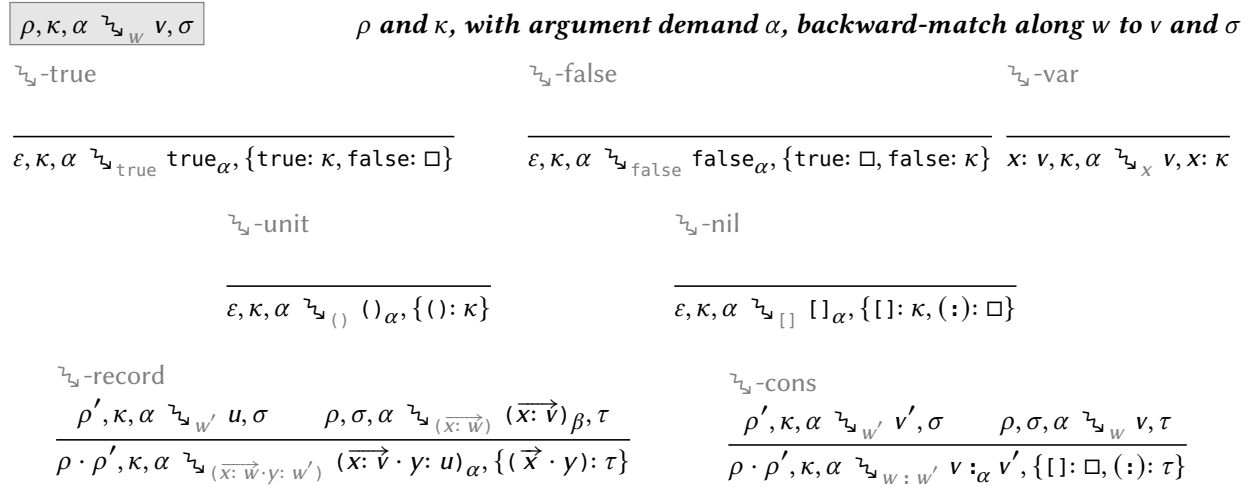


Fig. 11. Backward match

the entire list, using α as the demand on the root cons node. The eliminator selection σ represents the demand on the interim eliminator used to match the tail, and τ the demand on the eliminator used to match the head; these are then combined into a demand on the eliminator used to match the whole list, with \square again used to represent the absence of demand on the nil branch. Records are similar, except that there is only a single branch. The selection state β computed for the initial part of the record is an artefact of processing records recursively, and is disregarded.

LEMMA 3.5 (MONOTONICITY OF \rightsquigarrow_w). *Suppose $w :: v, \sigma \rightsquigarrow \rho, \kappa$, with $\rho, \kappa, \alpha \rightsquigarrow_w v, \sigma$ and $\rho', \kappa', \alpha' \rightsquigarrow_w v', \sigma'$. If $(\rho, \kappa, \alpha) \sqsubseteq (\rho', \kappa', \alpha')$ then $(v, \sigma) \sqsubseteq (v', \sigma')$.*

3.3.2 Backward Evaluation. Figure 12 defines a family of *backward-evaluation* functions \rightsquigarrow_T of type $\text{Sel}_v \mathcal{A} \rightarrow (\text{Sel}_{\rho, e} \mathcal{A}) \times \mathcal{A}$ for any $T :: \rho, e \Rightarrow v$. Backward evaluation rewinds T , using the output selection $v \in \text{Sel}_v \mathcal{A}$ to determine an input selection $(\rho, e) \in \text{Sel}_{\rho, e} \mathcal{A}$ and an argument demand $\alpha \in \mathcal{A}$ which will eventually be pushed back onto the argument of the dynamically innermost function call. (At the outermost level, where there are no active function calls, the argument demand is discarded.) The rules resemble those of the evaluation relation \Rightarrow with inputs and outputs flipped. The general pattern is that each backward rule takes the join of the demand attached to any partial values constructed at that step, and the argument demand associated with any subcomputations, and passes it upwards as the new argument demand. The output environment is constructed similarly, by joining the demand flowing back through the environment copies used to evaluate subcomputations. Demand is also attached to the source expression when it is the expression form responsible for the construction of a demanded value.

Function application. The application rule is where the argument demand is used and the function context changes, so we start here. The rule essentially runs the forward evaluation rule in reverse, using the trace T' to backward-evaluate the function body. The argument demand β associated with T' is the join of the demand on any resources constructed directly by that function invocation, and is transferred to the matched part of the function argument by the backward-match function \rightsquigarrow_w . The argument demand passed upwards into the enclosing function context is $\alpha \sqcup \alpha'$, representing the resources needed along T and U . The auxiliary function $\rightsquigarrow_{\rho, h} : \text{Sel}_{\rho'} \mathcal{A} \rightarrow (\text{Sel}_{\rho, h} \mathcal{A}) \times \mathcal{A}$ for any $\rho, h \rightarrow \rho'$ defined at the bottom of Figure 12 is used to turn ρ_2 , capturing the demand flowing back through any recursive uses of the function and any others with which it was mutually defined,

$v \Downarrow_T \rho, e, \alpha$	v <i>backward-evaluates along T to ρ and e, with argument demand α</i>		
\Downarrow -hole	\Downarrow -var	\Downarrow -lambda	\Downarrow -int
$\frac{\square \doteq v \quad v \Downarrow_T \rho, e, \alpha}{\square \Downarrow_T \rho, e, \alpha}$	$\frac{\rho' \ni_{\rho} x: v}{v \Downarrow_x \rho', x, \perp}$	$\frac{}{cl(\rho, \varepsilon, \alpha, \sigma) \Downarrow_{\lambda\sigma'} \rho, \lambda\sigma, \alpha}$	$\frac{}{n_{\alpha} \Downarrow_n \square_{\rho}, n_{\alpha}, \alpha}$
\Downarrow -true	\Downarrow -false	\Downarrow -record	
$\frac{}{true_{\alpha} \Downarrow_{true} \square_{\rho}, true_{\alpha}, \alpha}$	$\frac{}{false_{\alpha} \Downarrow_{false} \square_{\rho}, false_{\alpha}, \alpha}$	$\frac{v_i \Downarrow_{T_i} \rho_i, e_i, \alpha'_i \quad (\forall i \leq \vec{x}')}{(x: \vec{v})_{\alpha} \Downarrow_{(x: \vec{T})} \sqcup \vec{\rho}, (x: \vec{e})_{\alpha}, \alpha \sqcup \sqcup \vec{\alpha}'}$	
\Downarrow -project		\Downarrow -nil	
$\frac{\vec{x}: \vec{u} \ni_{\vec{x}: \vec{v}} y: v' \quad (\vec{x}: \vec{u})_{\perp} \Downarrow_T \rho, e, \alpha}{v' \Downarrow_{T_{\vec{x}: \vec{v}} \cdot y} \rho, e, y, \alpha}$		$\frac{}{[]_{\alpha} \Downarrow_{[]} \square_{\rho}, []_{\alpha}, \alpha}$	
\Downarrow -cons	\Downarrow -let-rec		
$\frac{v \Downarrow_T \rho, e, \alpha \quad v' \Downarrow_U \rho', e', \alpha'}{v: \beta v' \Downarrow_{T: U} \rho \sqcup \rho', e: \beta e', \beta \sqcup \alpha \sqcup \alpha'}$	$\frac{v \Downarrow_T \rho \cdot \rho_1, e, \alpha \quad \rho_1 \Downarrow \rho', h', \alpha'}{v \Downarrow_{let\ h\ in\ T} \rho \sqcup \rho', let\ h' \ in\ e, \alpha \sqcup \alpha'}$		
\Downarrow -apply-prim			
$\frac{n_{i\alpha_i} \Downarrow_{U_i} \rho_i, e_i, \beta_i \quad (\forall i \in \vec{n})}{m_{\alpha'} \Downarrow_{\phi(\vec{U}_n)} \sqcup \vec{\rho}, \phi(\vec{e}), \sqcup \vec{\beta}} \phi_{\vec{n}}^*(\alpha') = \vec{\alpha}$			
\Downarrow -apply			
$\frac{v \Downarrow_{T'} \rho_1 \cdot \rho_2 \cdot \rho_3, e, \beta \quad \rho_3, e, \beta \Downarrow_w v', \sigma \quad v' \Downarrow_U \rho, e_2, \alpha}{\rho_2 \Downarrow \rho'_1, h, \beta' \quad cl(\rho_1 \sqcup \rho'_1, h, \beta \sqcup \beta', \sigma) \Downarrow_T \rho', e_1, \alpha'}{v \Downarrow_{T U \triangleright w: T'} \rho \sqcup \rho', e_1 e_2, \alpha \sqcup \alpha'}$			
$\rho' \ni_{\rho} x: v$	ρ' <i>contains</i> $x: v$	$\rho \Downarrow \rho', h, \alpha$	ρ <i>backward-generates to</i> ρ', h, α
\ni -head	\ni -tail	\Downarrow -rec-defs	
$\frac{}{(\square_{\rho} \cdot x: u) \ni_{\rho \cdot x: v} x: u}$	$\frac{\rho' \ni_{\rho} x: u \quad x \neq y}{(\rho' \cdot y: \square) \ni_{\rho \cdot y: v} x: u}$	$\frac{v_i = cl(\rho_i, h_i, \alpha_i, \sigma_i) \quad (\forall i \in \vec{x}')}{\vec{x}: \vec{v} \Downarrow \sqcup \vec{\rho}, \vec{x}: \vec{\sigma} \sqcup \sqcup \vec{h}, \sqcup \vec{\alpha}}$	

Fig. 12. Backward evaluation

into information that can be merged back into the demand on the closure. The function $\Downarrow_{\rho, h}$ is also used in the letrec rule, which otherwise follows the general pattern described above.

Primitive application. Each primitive operation $\phi: \text{Int}^i \rightarrow \text{Int}$ must provide a backward-dependency function $\phi_{\vec{n}}^*: \mathcal{A} \rightarrow \mathcal{A}^i$ for every $\vec{n} \in \text{Int}^i$ which specifies how to turn the output selection α' on $\phi(\vec{n})$ into an input selection $\vec{\alpha} \in \mathcal{A}^i$ on \vec{n} . The rule for primitive application uses this information to pair each argument n_i with its demand α_i and then backwards-evaluate the argument. The argument demand passed upward is the join of those arising from these subcomputations, and is unrelated to the execution of the primitive itself, similar to a function application. Here $\sqcup \vec{\beta}$ means the fold of \sqcup (with unit \perp) over the sequence of selection states $\beta_1 \cdot \dots \cdot \beta_{|\vec{x}'|}$. Environment demands $\vec{\rho} = \rho_1 \cdot \dots \cdot \rho_{|\vec{n}'|}$ are joined (pointwise) in a similar fashion.

Other rules. In the variable case, no partial values were constructed during evaluation and there are no subcomputations, so the argument demand is \perp , the unit for \sqcup . The returned environment selection demands v for the variable x and \square for all other variables, using the family of *backwards lookup* functions $- \ni_{\rho} x: -$ of type $\text{Sel}_v \mathcal{A} \rightarrow \text{Sel}_{\rho} \mathcal{A}$ for any $x: v \in \rho$ also defined in Figure 12. (The output of the function is on the left in the relational notation.) For atomic values such as integers, Booleans and `nil`, the argument demand is simply the demand α associated with the constructed value, which is also attached to the corresponding expression, and the environment demand has \square for every variable in the original environment ρ , written \square_{ρ} .

For closures, the argument demand is unpacked along with the other components, preserving any internal selections on ρ and σ . Composite values such as records and cons cells follow the general pattern; thus for records, the argument demands α'_i from the subcomputations are joined with the α on the record itself to produce the argument demand passed upward. Record projection never demands the record constructor itself, but simply promotes the field demand into a record demand, using $\ni_{x:\vec{v}}$ to demand fields other than y with \square .

Hole rule. The hole rule, as elsewhere, ensures that the function is defined when v is \square , and it is easy to show that \ni_T preserves \sqsubseteq , and thus \doteq .

LEMMA 3.6 (MONOTONICITY OF \ni_T). *Suppose $T :: \rho, e \Rightarrow v$ with $v \ni_T \rho, e, \alpha$ and $v' \ni_T \rho', e', \alpha'$. If $v \sqsubseteq v'$ then $(\rho, e, \alpha) \sqsubseteq (\rho', e', \alpha')$.*

3.4 Round-Tripping Properties of \ni_T and \ni_T

We now establish more formally the round-tripping properties, alluded at the beginning of the section, that relate \ni_T to \ni_T . For the analyses to be coherent, we expect $\ni_T(\ni_T(v))$ to produce a value selection $v' \sqsupseteq v$, and $\ni_T(\ni_T(\rho, e))$ to produce an input selection $(\rho', e') \sqsubseteq (\rho, e)$. Pairs of (monotonic) functions $f: X \rightarrow Y$ and $g: Y \rightarrow X$ that are related in this way are called *Galois connections*. Galois connections generalise isomorphisms: f and g are not quite mutual inverses, but are the nearest to an inverse each can get to the other. We will present a visual example of some of these round-tripping properties in § 4.2; here we establish the relevant theorems.

Definition 3.7 (Galois connection). Suppose X and Y are sets equipped with partial orders \leq_X and \leq_Y . Then monotonic functions $f: X \rightarrow Y$ and $g: Y \rightarrow X$ form a *Galois connection* $(f, g): X \rightarrow Y$ iff $g(f(x)) \geq_X x$ and $f(g(y)) \leq_Y y$.

Galois connections are also adjoint functors between poset categories, with left and right adjoints f and g usually called the *lower* and *upper* adjoints, because f approximates an inverse of g from below, and g an inverse of f from above. Galois connections compose component-wise, so it is useful to think of them as having a type $X \rightarrow Y$, with the direction (by convention) given by the lower adjoint. If $\gamma: X \rightarrow Y$ is a Galois connection, we will write γ^* and γ_* for the lower and upper adjoints respectively; an important property we will return to is that γ^* preserves joins and γ_* preserves meets. We now show that, for any \mathcal{A} , \ni_T and \ni_T form a Galois connection (Theorem 3.11), by first establishing that the relevant auxiliary functions also form Galois connections.

THEOREM 3.8 (GALOIS CONNECTION FOR PATTERN-MATCHING). *Suppose $w :: v, \sigma \rightsquigarrow \rho, \kappa$. Then $(\ni_w, \ni_w): (\text{Sel}_{\rho, \kappa} \mathcal{A}) \times \mathcal{A} \rightarrow \text{Sel}_{v, \sigma} \mathcal{A}$ is a Galois connection.*

PROOF. Included with [supplementary materials](#). □

LEMMA 3.9 (GALOIS CONNECTION FOR ENVIRONMENT LOOKUP). *Suppose $x: v \in \rho$. Then $(- \ni_{\rho} x, \in_{\rho} x: -): \text{Sel}_v \mathcal{A} \rightarrow \text{Sel}_{\rho} \mathcal{A}$ is a Galois connection.*

PROOF. Included with [supplementary materials](#). □

THEOREM 3.10 (GALOIS CONNECTION FOR RECURSIVE BINDINGS). *Suppose $\rho, \mathbf{h} \rightarrow \rho'$. Then $(\Downarrow_{\rho, \mathbf{h}}, \Uparrow_{\rho, \mathbf{h}}) : \text{Sel}_{\rho'} \mathcal{A} \rightarrow (\text{Sel}_{\rho, \mathbf{h}} \mathcal{A}) \times \mathcal{A}$ is a Galois connection.*

PROOF. Included with [supplementary materials](#). □

We assume (rather than prove) that the backward and forward dependency functions $\phi_{\vec{n}}^*$ and $\phi_{\vec{n}}$ provided for every primitive operation $\phi : \text{Int}^i \rightarrow \text{Int}$ and every \vec{n} of length i form a Galois connection of type $\mathcal{A} \rightarrow \mathcal{A}^i$. Under this assumption the following holds.

THEOREM 3.11 (GALOIS CONNECTION FOR EVALUATION). *Suppose $T :: \rho, \mathbf{e} \Rightarrow \mathbf{v}$. Then $(\Downarrow_T, \Uparrow_T) : \text{Sel}_{\mathbf{v}} \mathcal{A} \rightarrow (\text{Sel}_{\rho, \mathbf{e}} \mathcal{A}) \times \mathcal{A}$ is a Galois connection.*

PROOF. Included with [supplementary materials](#). □

Establishing that $(\Downarrow_T, \Uparrow_T)$ is an adjoint pair might seem rather weak as a correctness property: it merely ensures that the two analyses are related in a sensible way, not that they actually capture any useful information. This is a familiar problem from other approximate analyses like type systems and model checking, where properties like soundness or completeness are essential but do not by themselves guarantee utility. One could certainly define versions of \Downarrow_T and \Uparrow_T that are too coarse grained to be useful, yet still satisfy Theorem 3.11. However Galois connections do at least require that every tightening or tweak to the forward analysis is paired with a corresponding adjustment to the backward analysis, and vice-versa. In § 6 we consider how other ideas from provenance and program slicing might be adapted to provide additional correctness criteria.

4 DE MORGAN DEPENDENCIES FOR BRUSHING AND LINKING

§ 3 addresses the first kind of question we motivated in the introduction (§ 1.1). In particular \Downarrow_T can answer questions like: “what data is needed to compute this bar in a bar chart?”, and indeed we were able to use our implementation to generate Figure 1. The second problem we set ourselves was how to link selections between *cognate* outputs, i.e. outputs computed from the same data (§ 1.2). This is called “brushing and linking” in data visualisation [Becker and Cleveland 1987], and has been extensively studied as an interaction paradigm, but with little emphasis on techniques for automation. Intuitively, the problem has a bidirectional flavour: one must consider how dependencies flow backward from a selection in one output to a selection v in the common data, and then forward from the selected data v to a corresponding selection in the other output. A natural question then is whether the analysis established in § 3 can supply the information required to support an automated solution.

An immediate problem is that the flavour of the forward dependency required here differs from that provided by the forward analysis \Uparrow_T defined in § 3.2. That was able to answer the question: what can we compute given only the data selected in v ? But to identify the related data in another output, we must determine not what the input selection v is sufficient for, but what it is necessary for: those parts of the other output that depend on v . In fact the question can be formulated as a kind of dual: what would we *not* be able to compute if the data selected in v were *unavailable*?

4.1 De Morgan Duality

Why \Uparrow_T is unsuitable as a forward dependency relation for linking cognate outputs can also be understood in terms of compositionality. Suppose \mathcal{V}_1 and \mathcal{V}_2 are the lattices of selections for two views computed from a shared input source, and \mathcal{D} is the lattice of selections for the shared input. Using the procedure given in § 3, we can obtain two Galois connections $\gamma : \mathcal{V}_1 \rightarrow \mathcal{D}$ and $\delta : \mathcal{V}_2 \rightarrow \mathcal{D}$ as shown in Figure 13a. (The reader can ignore Figure 13b for the moment.)

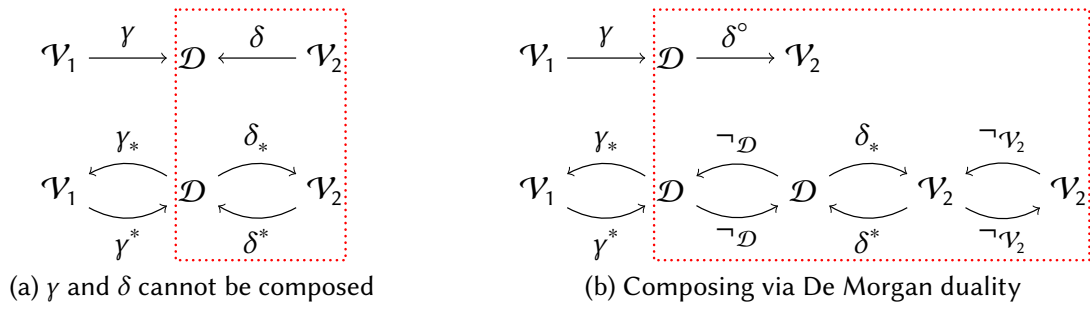


Fig. 13. Dualising $\delta : \mathcal{V}_2 \rightarrow \mathcal{D}$ for composition with $\gamma : \mathcal{V}_1 \rightarrow \mathcal{D}$

Unfortunately, γ and δ are not composable, as their types makes clear. While the upper adjoint $\delta_* : \mathcal{D} \rightarrow \mathcal{V}_2$ has the correct type to compose with the lower adjoint $\gamma^* : \mathcal{V}_1 \rightarrow \mathcal{D}$, the result is not a Galois connection: δ_* preserves meets, whereas γ^* preserves joins. However, it turns out that if selections are closed under complement, we can derive an analysis of what is *necessary* for a given input selection from an analysis of what it is *sufficient* for. The effect is to invert δ , yielding a Galois connection δ° with a type that allows it to compose with γ . Then the composite $\delta^\circ \circ \gamma$ is a Galois connection linking \mathcal{V}_1 to \mathcal{V}_2 via \mathcal{D} , as shown in Figure 13b, offering a general mechanism for brushing and linking, with nice round-tripping properties. We now unpack this in more detail.

First we shift settings from the lattices used in § 3 to Boolean lattices (or Boolean algebras) $\mathcal{A} = \langle \mathcal{A}, \top, \perp, \sqcap, \sqcup, \neg \rangle$, which are lattices equipped with an involution $\neg : \mathcal{A} \rightarrow \mathcal{A}$ called *complement*. Boolean algebras satisfy complementation laws $x \sqcap \neg x = \perp$ and $x \sqcup \neg x = \top$ and De Morgan laws $\neg x \sqcap \neg y = \neg(x \sqcup y)$ and $\neg x \sqcup \neg y = \neg(x \sqcap y)$. If \mathcal{A} is a Boolean algebra, then $\text{Sel}_v \mathcal{A}$ is also a Boolean algebra, with the Boolean operations, and in particular $\neg_v : \text{Sel}_v \mathcal{A} \rightarrow \text{Sel}_v \mathcal{A}$, defined pointwise. An additional distinguished value selection \blacksquare serves as the negation of \square . The two-point lattice 2 we used to illustrate § 3 is also a Boolean algebra $\langle \{\text{tt}, \text{ff}\}, \text{tt}, \text{ff}, \wedge, \vee, \neg \rangle$ with \neg corresponding to logical negation.

It is an easy consequence of the complementation and De Morgan laws that any meet-preserving operation $g : \mathcal{A} \rightarrow \mathcal{B}$ on Boolean algebras has a join-preserving De Morgan dual $g^\circ : \mathcal{A} \rightarrow \mathcal{B}$ given by $\neg_{\mathcal{B}} \circ g \circ \neg_{\mathcal{A}}$, and any join-preserving operation h has a meet-preserving De Morgan dual h° defined similarly. Moreover if h is the lower adjoint of g , then g° is the lower adjoint of h° . Thus Galois connections on Boolean algebras also admit a (contravariant) notion of De Morgan duality, defined component-wise.

Definition 4.1 (De Morgan dual of a Galois connection). Suppose \mathcal{A} and \mathcal{B} are Boolean algebras and $\gamma : \mathcal{A} \rightarrow \mathcal{B}$ is a Galois connection (γ^*, γ_*) . Define the *De Morgan dual* γ° of γ to be the Galois connection $(\gamma_*^\circ, \gamma^{*\circ}) : \mathcal{B} \rightarrow \mathcal{A}$.

Dualising a Galois connection flips the direction of the arrow by swapping the roles of the upper and lower adjoints. So while $\gamma : \mathcal{A} \rightarrow \mathcal{B}$ and $\delta : \mathcal{C} \rightarrow \mathcal{B}$ are not composable, γ and $\delta^\circ : \mathcal{B} \rightarrow \mathcal{C}$ are, and the composition is achieved by transforming δ_* from something which determines what we can compute with v into something which determines what we cannot compute without v . This offers a principled basis for an automated brushing and linking feature between cognate computations T and U . When the user selects part of the output of T , we can use \searrow_T to compute the needed data v , and then use \nearrow_U° to compute the parts of the output of U that depend on v . This is the approach implemented in Fluid, and we used this to generate Figure 2 in § 1.2.

```

1  let zero n = const n;
2  wrap n n_max = ((n - 1) `mod` n_max) + 1;
3  extend n = min (max n 1);
4  nth2 i j xss = nth (j - 1) (nth (i - 1) xss);
5
6  let convolve image kernel method =
7    let ((m, n), (i, j)) = (dims image, dims kernel);
8      (half_i, half_j) = (i `quot` 2, j `quot` 2);
9      area = i * j
10   in < let weightedSum = sum [
11       image!(x, y) * kernel!(i' + 1, j' + 1)
12       | (i', j') ← range (0, 0) (i - 1, j - 1),
13       let x = method (m' + i' - half_i) m,
14         let y = method (n' + j' - half_j) n,
15         x ≥ 1, x ≤ m, y ≥ 1, y ≤ n
16     ] in weightedSum `quot` area
17     | (m', n') in (m, n) >;
1  let emboss = [[-2, -1, 0],
2               [-1, 1, 1],
3               [ 0, 1, 2]];
4  filter = < nth2 i j emboss
5             | (i, j) in (3, 3) >;
6  image' = [[15, 13, 6, 9, 16],
7            [12, 5, 15, 4, 13],
8            [14, 9, 20, 8, 1],
9            [ 4, 10, 3, 7, 19],
10           [ 3, 11, 15, 2, 9]];
11  image = < nth2 i j image'
12          | (i, j) in (5, 5) >
13  in convolve image filter zero

```

Fig. 14. Matrix convolution example, with methods zero, wrap and extend for dealing with boundaries

4.2 Example: Matrix Convolution

We now illustrate the $(\mathcal{A}_T^\circ, \mathcal{S}_T^\circ)$ Galois connection, contrasting it with $(\mathcal{S}_T, \mathcal{A}_T)$, using an example which computes the convolution of a 5×5 matrix with a 3×3 kernel. Convolution has an intuitive dependency structure and the values involved have an easy visual presentation, making it useful for conveying the flavour of the four distinct (but connected) dependency relations that arise in the framework. The source code for the example is given in Figure 14, and shows the convolve function, plus zero, wrap and extend which provide different methods for handling the boundaries of the input matrix. The angle-bracket notation is used to construct matrices, which were omitted from § 2. (The formal treatment is similar to records.)

Fluid was used to generate the diagrams in Figure 15, which show the four dependency relations and two of their four possible round-trips. Figure 15a shows the $(\mathcal{S}_T, \mathcal{A}_T)$ Galois connection defined in § 3.4. In the upper figure, the user selects (in green) the output cell at position (2, 2) (counting rows downwards from 1). This induces a demand (via the lower adjoint \mathcal{S}_T) on the input matrix `image` and the kernel `filter`, revealing (in blue) that the entire kernel was needed to compute the value 1, but only some of the input matrix. In particular the elements at (1, 3) and (3, 1) in `image` were not needed, because of zeros present in `filter`. If we then “round-trip” that input selection, computing the corresponding availability \mathcal{A}_T on the output using the upper adjoint \mathcal{A}_T , the green selection grows: it turns out that the data needed to make (2, 2) available are sufficient to make (1, 1) available as well.

Figure 15b shows the De Morgan dual $(\mathcal{A}_T^\circ, \mathcal{S}_T^\circ)$. In the upper part of the figure, the user selects (green) kernel cell (1, 2) to see the output cells that depend on it. This is computed using the De Morgan dual \mathcal{A}_T° . First we negate the input selection, marking (1, 2) as unavailable, and all other inputs as available. Then we forward-analyse with \mathcal{A}_T to determine that with this data selection, we can only compute the top row of the output. (If it seems odd that we can compute even the top row, notice that the example uses the method zero for dealing with boundaries; wrap or extend would give a different behaviour.) Then we negate that top row selection to produce the (blue) output selection shown in the figure. These are exactly the output cells which depend on kernel cell (1, 2) in the sense that they cannot be computed if that input is unavailable.



Fig. 15. Upper and lower pairs are dual; left and right pairs are adjoint

We can then round-trip this output selection using the De Morgan dual \bowleftarrow_T° . We first negate the blue output selection (selecting the top row of the output again), and then use \bowtie_T to determine the needed inputs, which turn out to be the top two rows of image, and the top row of filter. Negating again produces the green output selection shown in the lower figure. Thus the backwards De Morgan dual computes the inputs that would *not* be needed if the selected outputs were not needed: more economically, the inputs that are *only* needed for the selected output. Here the round-trip reveals that if kernel cell (1, 2) is unavailable, then the entire top row of the kernel might as well have been unavailable too, and similarly for the bottom 3 rows of the input.

4.3 Relationship to Galois Slicing

The De Morgan dual puts us in a better position to consider the relationship between the present system and earlier work on *Galois slicing*, a program slicing technique that has been explored for pure functional programs [Perera 2013; Perera et al. 2012], functional programs with effects [Ricciotti et al. 2017], and π -calculus [Perera et al. 2016]. We consider other related work in § 6.1.

Galois slicing operates on lattices of *slices*, which are programs (or values) where parts deemed irrelevant are replaced by a hole \square . (If we think of the notion of selection defined in § 3.1.1 as picking out a subset of the paths in a term, then slices resembles selections which are prefix-closed, meaning that if a given path in a term is selected, then so are all of its prefixes.) For a fixed computation, a meet-preserving *forward-slicing* function is defined which takes input slices to output slices, discarding parts which cannot be computed because the needed input is not present, plus a join-preserving *backward-slicing* function taking output slices to input slices, retaining the parts needed for the output slice. For example Figure 16a shows a computation with output $(\theta.4, \theta.6)$, and Figure 16b gives the backward slice for output slice $(\theta.4, \square)$. Forward and backward slicing, for a given computation, form a Galois connection, giving the analyses the nice round-tripping properties we motivated in § 3.4.

Unfortunately, the notion of slice does not lend itself to computing dependencies where the needed input or output is a proper part of a value, such as a component of a tuple. *Differential slicing* [Perera et al. 2012] improves on this by using Galois slicing to compute a pair of input slices (e, e') for a pair of output slices (v, v') where $v \sqsubseteq v'$. By monotonicity, $e \sqsubseteq e'$. This can be used to

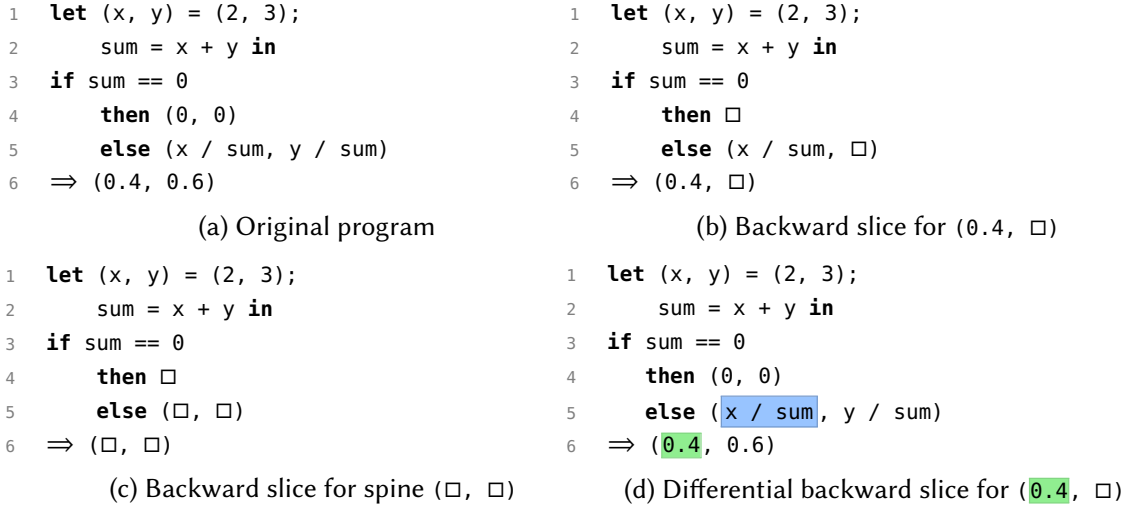


Fig. 16. Differential Galois slicing selects input (blue) needed *only* for selected output (green)

compute a (differential) slice for an arbitrary subtree, by setting v to be the “spine” of the original output up to the location of the subtree, and v' to be v with the subtree of interest plugged back in. Here we could focus on the value 0.4 in the output by computing the backward slice for (\square, \square) (Figure 16c) and then comparing it with the backward slice for $(0.4, \square)$, generating a differential slice where the parts that are different are highlighted (Figure 16d). But although it supports a notion of selection which is closer to what we need, the differential slice highlights only the program parts that are needed *exclusively* by the selected output, and as such underapproximates the dependency information needed for data linking. (In fact differential slicing is similar to the De Morgan dual \bowtie_T° .) Because in this example 2 and 3 are needed to compute the spine as well (in order to decide which conditional branch to execute), they are excluded from the differential slice, whereas our backward analysis \bowtie_T is able to directly determine that both 2 and 3 are needed to compute 0.4.

5 GALOIS CONNECTIONS FOR DESUGARING

Elaborating a richer surface language into a simpler core is a common pattern with well known benefits. It can, however, make it harder to express certain information to the programmer in terms of the surface language. We face this problem with the analysis in § 3, which links outputs not only to inputs, but also to expressions responsible for introducing data. We could use this information in an IDE to link structured outputs to relevant code fragments, but only if we are able to map term selections back to the surface program. We now sketch a bidirectional desugaring procedure which addresses this, and which composes with the Galois dependency analysis defined in § 3.

5.1 Surface Language Syntax

The surface language, Fluid, extends the core syntax with list notation $[s, \dots, s']$, Haskell 98-style list comprehensions [Jones 2003], list enumerations, first-class primitives and piecewise function definitions and pattern-matching, as shown in Figure 17. Typing rules are included with the supplementary materials. We attach selection states α to surface terms s, t that desugar to core terms with selections, and let \mathbf{s}, \mathbf{t} range over “raw” surface terms, which are isomorphic to the term selections where the type of selection states is the unit lattice 1.

Figure 18 shows how the end-to-end mapping would appear to a user. (For illustrative purposes the library function `map` and some raw data are included in the same source file.) On the left, the user

<p>Identifier</p> <p>$x, y ::= \dots$</p> <p>\oplus operator name</p> <p>Surface term</p> <p>$s, t ::= \dots$</p> <p>(\oplus) first-class operator</p> <p>$s \oplus s'$ binary application</p> <p>$\text{let } \vec{g} \text{ in } s$ recursive functions</p> <p>$\text{if } s \text{ then } s \text{ else } s$ if</p> <p>$\text{match } s \text{ as } \vec{p} = \vec{s}$ match</p> <p>$\text{let } p = s \text{ in } s$ structured let</p> <p>$[\alpha \ s \ r]$ non-empty list</p> <p>$[s \ . \ . \ s]$ list enum</p> <p>$[s \ \ \vec{q}]_{\alpha}$ list comprehension</p> <p>List rest term</p> <p>$r ::=]_{\alpha}$ end</p> <p>$, \alpha \ s \ r$ cons</p>	<p>Recursive function</p> <p>$g ::= x: \vec{c}$</p> <p>Clause</p> <p>$c ::= \vec{p} = s$</p> <p>Pattern</p> <p>$p ::= x$ variable</p> <p>$(x: \vec{p})$ record</p> <p>$[]$ nil</p> <p>$p : p$ cons</p> <p>$[p \ o]$ non-empty list</p> <p>List rest pattern</p> <p>$o ::=]$ end</p> <p>$, p \ o$ cons</p> <p>Qualifier</p> <p>$q ::= s$ guard</p> <p>$\text{let } p = s$ declaration</p> <p>$p \leftarrow s$ generator</p>
--	--

Fig. 17. Syntax for surface language, with selection states

selects a cons cell (green) in the output; by backwards evaluating and then backwards desugaring, we are able to highlight the list comprehension, the cons in the second clause of map, and both occurrences of the constant "Hydro". These last two are highlighted because the selected cons cell was constructed by eliminating a Boolean that was in turn constructed by the primitive == operator, which consumed the two strings. The user might then conjecture that the two occurrences of "Geo" were somehow responsible for the inclusion of the third cons cell in the output; they can confirm this by making the green selection on the right. (Highlighting == too would clearly be helpful here; we discuss this possibility in § 6.1.) The grey selection is included to contrast the cons highlighting with the data demanded by the list elements themselves, which is quite different.

<pre> 1 let map f [] = []; 2 map f (x : xs) = f x : map f xs; 3 let data = [4 { energyType: "Bio", output: 6.2 }, 5 { energyType: "Hydro", output: 260 }, 6 { energyType: "Solar", output: 19.9 }, 7 { energyType: "Wind", output: 91 }, 8 { energyType: "Geo", output: 14.4 } 9]; 10 let xs = [row.output 11 type ← ["Hydro", "Solar", "Geo"], 12 row ← data, row.energyType == type 13] in 14 map (fun x → floor (x / sum xs * 100)) xs 15 ⇒ (88 : (6 : (4 : []))) </pre>	<pre> 1 let map f [] = []; 2 map f (x : xs) = f x : map f xs; 3 let data = [4 { energyType: "Bio", output: 6.2 }, 5 { energyType: "Hydro", output: 260 }, 6 { energyType: "Solar", output: 19.9 }, 7 { energyType: "Wind", output: 91 }, 8 { energyType: "Geo", output: 14.4 } 9]; 10 let xs = [row.output 11 type ← ["Hydro", "Solar", "Geo"], 12 row ← data, row.energyType == type 13] in 14 map (fun x → floor (x / sum xs * 100)) xs 15 ⇒ (88 : (6 : (4 : []))) </pre>
--	--

Fig. 18. Source selections (blue) resulting from selecting different list cells (green)

$s \nearrow e$					s <i>forward-desugars to</i> e
\nearrow -nil	\nearrow -cons	\nearrow -non-empty-list	\nearrow -list-comp-done		
$\frac{}{[\]_\alpha \nearrow [\]_\alpha}$	$\frac{s \nearrow e \quad s' \nearrow e'}{s :_\alpha s' \nearrow e :_\alpha e'}$	$\frac{s \nearrow e \quad r \nearrow e'}{[\]_\alpha s r \nearrow e :_\alpha e'}$	$\frac{s \nearrow e}{[s \mid \varepsilon]_\alpha \nearrow e :_\alpha [\]_\alpha}$		
	\nearrow -list-comp-gen				
	$\frac{[s \mid \vec{q}]_\alpha \nearrow e \quad p, e \nearrow \sigma \quad \sigma, \alpha \nearrow_p \sigma' \quad s' \nearrow e'}{[s \mid p \leftarrow s' \cdot \vec{q}]_\alpha \nearrow \text{concatMap } \lambda \sigma' e'}$				
\nearrow -list-comp-guard	\nearrow -list-comp-decl				
$\frac{[s \mid \vec{q}]_\alpha \nearrow e \quad s' \nearrow e'}{[s \mid s' \cdot \vec{q}]_\alpha \nearrow \lambda\{\text{true}: e, \text{false}: [\]_\alpha\} e'}$	$\frac{[s \mid \vec{q}]_\alpha \nearrow e \quad p, e \nearrow \sigma \quad s' \nearrow e}{[s \mid \text{let } p = s' \cdot \vec{q}]_\alpha \nearrow \lambda \sigma e}$				
$e \Downarrow_t s$					e <i>backward-desugars along t to</i> s
\Downarrow -nil	\Downarrow -cons	\Downarrow -non-empty-list	\Downarrow -list-comp-done		
$\frac{}{[\]_\alpha \Downarrow [\]_\alpha}$	$\frac{e \Downarrow_t s \quad e' \Downarrow_{t'} s'}{e :_\alpha e' \Downarrow_{t:t'} s :_\alpha s'}$	$\frac{e \Downarrow_t s \quad e' \Downarrow_r r'}{e :_\alpha e' \Downarrow_{[t]r} [\]_\alpha s r'}$	$\frac{e \Downarrow_t s}{e :_\alpha [\]_\alpha \Downarrow_{[t] \varepsilon} [s \mid \varepsilon]_\alpha \Downarrow_\alpha}$		
	\Downarrow -list-comp-gen				
	$\frac{e \Downarrow_t s \quad \sigma \searrow_p \sigma', \beta \quad \sigma' \searrow_p e' \quad e' \Downarrow_{[t'] \vec{q}} [s' \mid \vec{q}']_\beta}{\text{concatMap } \lambda \sigma e \Downarrow_{[t'] p \leftarrow t \cdot \vec{q}} [s' \mid p \leftarrow s \cdot \vec{q}']_\beta \Downarrow_\beta}$				
\Downarrow -list-comp-guard	\Downarrow -list-comp-decl				
$\frac{e' \Downarrow_{t'} s' \quad e \Downarrow_{[t] \vec{q}} [s \mid \vec{q}']_\beta}{\lambda\{\text{true}: e, \text{false}: [\]_\alpha\} e' \Downarrow_{[t] t' \cdot \vec{q}} [s \mid s' \cdot \vec{q}']_\alpha \Downarrow_\beta}$	$\frac{\sigma \searrow_p e' \quad e' \Downarrow_{[t'] \vec{q}} [s' \mid \vec{q}']_\beta \quad e \Downarrow_t s}{\lambda \sigma e \Downarrow_{[t'] \text{let } p = t \cdot \vec{q}} [s' \mid \text{let } p = s \cdot \vec{q}']_\beta}$				
$r \nearrow e$	r <i>forward-desugars to</i> e	$e \Downarrow_r r'$			e <i>backward-desugars along r to</i> r'
\nearrow -list-rest-end	\nearrow -list-rest-cons	\Downarrow -list-rest-end	\Downarrow -list-rest-cons		
$\frac{}{[\]_\alpha \nearrow [\]_\alpha}$	$\frac{s \nearrow e \quad r \nearrow e'}{(\]_\alpha s r) \nearrow e :_\alpha e'}$	$\frac{}{[\]_\alpha \Downarrow [\]_\alpha}$	$\frac{e \Downarrow_t s \quad e' \Downarrow_r r'}{e :_\alpha e' \Downarrow_{(, t r)} (\]_\alpha s r')}$		

Fig. 19. Forwards and backwards desugaring (selected rules only)

5.2 Forward Desugaring

To define the forward evaluation function \nearrow_T in § 3.2, we performed a regular evaluation using \Rightarrow to obtain a trace T , and then defined \nearrow_T by recursion over T , with T guiding the analysis in the presence of \square . There are no holes in the surface language, so we can take a simpler approach, defining a single *forward desugaring* relation \nearrow , and then showing that for every raw surface term $\mathbf{t} \nearrow \mathbf{e}$, there is a monotonic function $\nearrow_t: \text{Sel}_t \mathcal{A} \rightarrow \text{Sel}_e \mathcal{A}$, which is simply \nearrow domain-restricted to $\text{Sel}_t \mathcal{A}$. The full definition of \nearrow is included with the supplementary materials; Figure 19 gives a representative selection of the rules.

The definition follows a similar pattern to \nearrow_T . At each step, we take the meet of the availability on any parts of s being consumed at that step, and use that as the availability of any parts of e being generated at that step. Thus the rules for list notation simply transfer the selection state α on the opening and closing brackets $[\]_\alpha$ and $[\]_\alpha$ to the corresponding cons and nil of the resulting list,

and those on intervening delimiters $,_\alpha$ to the corresponding cons. List comprehensions $[s \mid \vec{q}]_\alpha$ have a rule for each kind of qualifier q at the head of \vec{q} , plus a rule for when \vec{q} is ε . The general pattern is to push the α on the comprehension itself through recursively, so it ends up on all core terms generated during its elaboration: in particular the `false` branch when q is a guard, and the singleton list when \vec{q} is empty. Auxiliary relations \nearrow and \nearrow_p (included with the supplementary materials) transfer availability on guards and generators onto the eliminators they elaborate into.

5.3 Backward Desugaring

The backwards analysis is then defined as a family of *backward desugaring* functions $\searrow_t: \text{Sel}_e \mathcal{A} \rightarrow \text{Sel}_t \mathcal{A}$ for any $t \nearrow e$, with the raw surface term t guiding the analysis backwards. (The role of t in disambiguating the backwards rules should be clear if you consider that e typically matches multiple rules but only one for a given t .) Figure 19 gives some representative rules; the full definition is included with the supplementary materials. To reverse a desugaring step, we take the join of the demand on any parts of e which were constructed at this step, and use that as the demand on the parts of s which were consumed at this step, turning demand on the core term into (minimal) demand on the surface term. Thus the effect of the list comprehension rules and auxiliary judgements is to set the demand on the comprehension itself to be the join of the demand of all the syntax generated during the elaboration of the comprehension, using auxiliary judgments \searrow_p and \searrow_p to transfer demand from eliminators back onto the guards and generators.

5.4 Round-Tripping Properties and Compositionality

It is easy to verify that \nearrow_t and \searrow_t are monotonic. Moreover they form an adjoint pair.

THEOREM 5.1 (GALOIS CONNECTION FOR DESUGARING). *Suppose $t \nearrow e$. Then $(\searrow_t, \nearrow_t) : \text{Sel}_e \mathcal{A} \rightarrow \text{Sel}_t \mathcal{A}$ is a Galois connection.*

PROOF. Included with [supplementary materials](#). □

The (\searrow_t, \nearrow_t) Galois connection readily composes with (\searrow_t, \nearrow_t) to produce surface-language selections like the ones shown in Figure 18. This is useful, although somewhat monolithic. In future work we will investigate techniques for backwards desugaring at arbitrary steps in the computation, perhaps by interleaving desugaring with execution in the style of [Pombrio and Krishnamurthi \[2014\]](#), as well as presenting selections on intermediate values (such as lists) in the surface language, even though they were not obtained via desugaring.

6 CONCLUSION

Our research was motivated by the goal of making computational outputs which are automatically able to reveal how they relate to data in a fine-grained way. A casual reader who wants to understand or fact-check a chart, or a scientist evaluating another's work, should be able to do so by interacting directly with an output. Recent work by [Walny et al. \[2019\]](#) suggests that developers would also benefit from such a feature while implementing visualisations, for example to check whether a quantity is represented by diameter or area in a bubble chart.

Galois connections provide an appealing setting for this problem because of their elegant round-tripping properties. However, existing dynamic analysis techniques based on Galois connections do not lend themselves to richly structured outputs like visualisations and matrices. We presented an approach that allows focusing on arbitrary substructures, which also means data selections can be inverted. This enables linking not just of outputs to data, but of outputs to other outputs, providing a mathematical basis for a widely used (but so far ad hoc) feature in data visualisation. We implemented our approach in [Fluid](#), a realistic high-level functional programming language.

6.1 Other Related Work and Future Directions

We close by considering some limitations and opportunities in the context of other related work. Galois slicing [Perera et al. 2012, 2016; Ricciotti et al. 2017] was considered in § 4.3.

Executable slicing. Executable slices [Hall 1995] are programs with some parts removed, but which are still executable. Our approach computes data selections, not executable slices, but such a notion has obvious relevance in data science: “explaining” part of a result should (arguably) entail being able to recompute it. *Expression provenance* [Acar et al. 2012] explains how primitive values are computed using only primitive operations; however, this still omits crucial information, and does not obviously generalise to structured outputs. Work on executable slicing in term rewriting [Field and Tip 1998] could perhaps be adapted to structured data and combined with dependency tracking for higher-order data (§ 3.1).

Dynamic program analysis. Dynamic analysis techniques like dataflow analysis [Chen and Poole 1988] and taint tracking [Reps et al. 1995] tend to focus on variables, rather than parts of structured values, and lack round-tripping properties; Galois dependencies have a clear advantage here. A limitation of dynamic techniques which is shared by our approach is that they can usually only reveal *that* certain dependencies arise, not *why*, which requires analysing path conditions [Hammer et al. 2006]. In a data science setting this would clearly be valuable too, and it would be interesting to see if the benefits of the Galois framework can be extended to techniques for computing dynamic path conditions.

Brushing and linking. Brushing and linking has been extensively studied in the data visualisation community [Becker and Cleveland 1987; McDonald 1982], but although Roberts and Wright [2006] argued it should be ubiquitous, no automated method of implementation has been proposed to date. Geospatial applications like GeoDa [Anselin et al. 2006] hard-code view coordination features into specific views, and libraries like d3.js and Plotly support ad hoc linking mechanisms, with varying degrees of programmer effort required. No existing approach provides automation or round-tripping guarantees, or is able to provide data selections explaining why visual selections are linked.

Data provenance in data visualisation. A recent vision paper by Psallidas and Wu [2018] is the only work we are aware of that proposes that brushing and linking, and related view coordination features like cross-filtering, can be understood in terms of data provenance. In a relational (query processing) setting, where the relevant notion of provenance is lineage, they propose backward-analysing to data, and then forward-analysing to another view, although again without the round-tripping features of Galois connections. Moreover theirs is primarily a concept paper, proposing a research programme, rather than solving a specific problem.

7 ACKNOWLEDGEMENTS

Perera and Petricek were supported by The UKRI Strategic Priorities Fund under EPSRC Grant EP/T001569/1, particularly the *Tools, Practices and Systems* theme within that grant, and by The Alan Turing Institute under EPSRC grant EP/N510129/1. Wang was supported by *Expressive High-Level Languages for Bidirectional Transformations*, EPSRC Grant EP/T008911/1.

REFERENCES

- Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. A Core Calculus for Provenance. In *Proceedings of the First International Conference on Principles of Security and Trust* (Tallinn, Estonia) (POST '12). Springer-Verlag, Berlin, Heidelberg, 410–429. https://doi.org/10.1007/978-3-642-28641-4_22
- Luc Anselin, Ibnu Syabri, and Youngihn Kho. 2006. GeoDa: An Introduction to Spatial Data Analysis. *Geographical Analysis* 38, 1 (2006), 5–22. <https://doi.org/10.1111/j.0016-7363.2005.00671.x>

- Richard A. Becker and William S. Cleveland. 1987. Brushing Scatterplots. *Technometrics* 29, 2 (May 1987), 127–142. <https://doi.org/10.1080/00401706.1987.10488204>
- Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction*, Johan Jeuring (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–67.
- G erard Boudol and Ilaria Castellani. 1989. Permutation of transitions: An event structure semantics for CCS and SCCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J.W. Bakker, W.-P. Roever, and G. Rozenberg (Eds.). Lecture Notes in Computer Science, Vol. 354. Springer, 411–427. <https://doi.org/10.1007/BFb0013028>
- Nadieh Bremer and Marlieke Ranzijn. 2015. Urbanization in East Asia between 2000 and 2010. <http://nbremer.github.io/urbanization/>.
- Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)*. Springer-Verlag, London, UK, 316–330.
- TY Chen and PC Poole. 1988. Dynamic dataflow analysis. *Information and Software Technology* 30, 8 (1988), 497–505. [https://doi.org/10.1016/0950-5849\(88\)90146-2](https://doi.org/10.1016/0950-5849(88)90146-2)
- Richard H. Connelly and F. Lockwood Morris. 1995. A generalization of the trie data structure. *Mathematical Structures in Computer Science* 5, 3 (1995), 381–418. <https://doi.org/10.1017/S096012950000803>
- A. De Lucia, A.R. Fasolino, and M. Munro. 1996. Understanding function behaviors through program slicing. In *WPC '96. 4th Workshop on Program Comprehension*. 9–18. <https://doi.org/10.1109/WPC.1996.501116>
- John Field and Frank Tip. 1998. Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. *Information and Software Technology* 40, 11–12 (November/December 1998), 609–636.
- Jeremy Gibbons. 2017. APlicative Programming with Naperian Functors. In *European Symposium on Programming (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). 568–583. https://doi.org/10.1007/978-3-662-54434-1_21
- Sebastian Graf, Simon Peyton Jones, and Ryan G Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–30.
- Robert J. Hall. 1995. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering* 2 (1995), 33–53. <https://doi.org/10.1007/BF00873408>
- Christian Hammer, Martin Grimme, and Jens Krinke. 2006. Dynamic path conditions in dependence graphs. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 58–67. <https://doi.org/10.1145/1111542.1111552>
- Ralf Hinze. 2000. Generalizing generalized tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. <https://doi.org/10.1017/S0956796800003713>
- Simon L. Peyton Jones. 2003. Haskell 98: Introduction. *Journal of Functional Programming* 13, 1 (2003), 0–6.
- Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) (POPL '73). Association for Computing Machinery, New York, NY, USA, 194–206. <https://doi.org/10.1145/512927.512945>
- John Alan McDonald. 1982. *Interactive graphics for data analysis*. Ph.D. Dissertation.
- Greg Miller. 2006. A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science* 314, 5807 (2006), 1856–1857. <https://doi.org/10.1126/science.314.5807.1856>
- James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Security Symposium*.
- Roland Perera. 2013. *Interactive Functional Programming*. Ph.D. Dissertation. University of Birmingham, Birmingham, UK. <http://etheses.bham.ac.uk/4209/>.
- Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional Programs That Explain Their Work. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) (ICFP '12). ACM, New York, NY, USA, 365–376. <https://doi.org/10.1145/2364527.2364579>
- Roly Perera, Deepak Garg, and James Cheney. 2016. Causally Consistent Dynamic Slicing. In *Concurrency Theory, 27th International Conference, CONCUR '16 (Leibniz International Proceedings in Informatics (LIPIcs))*, Jos e Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum f ur Informatik, Dagstuhl, Germany. <https://doi.org/10.4230/LIPIcs.CONCUR.2016.18>
- Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences through Syntactic Sugar. *SIGPLAN Notices* 49, 6 (Jun 2014), 361–371. <https://doi.org/10.1145/2666356.2594319>
- Fotis Psallidas and Eugene Wu. 2018. Provenance for Interactive Visualizations. In *Workshop on Human-In-the-Loop Data Analytics (HILDA 2018)*. ACM.
- Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) (POPL '95). Association for Computing Machinery, New York, NY, USA, 49–61. <https://doi.org/10.1145/199448.199462>

- Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 14 (2017), 28 pages. <https://doi.org/10.1145/3110258>
- J. C. Roberts and M. A. E. Wright. 2006. Towards Ubiquitous Brushing for Information Visualization. In *Tenth International Conference on Information Visualisation (IV'06)*. 151–156. <https://doi.org/10.1109/IV.2006.113>
- A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan 2003), 5–19. <https://doi.org/10.1109/JSAC.2002.806121>
- Jacob VanderPlas, Brian E. Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *The Journal of Open Source Software* 3, 32 (2018). <https://doi.org/10.21105/joss.01057>
- Jagoda Walny, Christian Frisson, Mieka West, Doris Kosminsky, Søren Knudsen, Sheelagh Carpendale, and Wesley Willett. 2019. Data Changes Everything: Challenges and Opportunities in Data Visualization Design Handoff. *IEEE Transactions on Visualization and Computer Graphics* PP (08 2019), 1–1. <https://doi.org/10.1109/TVCG.2019.2934538>
- Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) (ICSE '81). IEEE Press, Piscataway, NJ, USA, 439–449. <https://doi.org/10.5555/800078.802557>

Part VI

Conclusions

Chapter 14

Contributions and outlook

The work presented in this thesis is the result of my long-term effort to rethink data science tooling from the perspective of programming languages research. The research has been undertaken at multiple institutions (Microsoft Research, The Alan Turing Institute, University of Kent) and involved collaboration with a number of co-authors.

The work presented in this thesis is not merely theoretical. An important contribution of this thesis is a practical implementation of the presented programming systems, languages and libraries. The resulting software packages have been made available as open-source. Some were developed by a larger team of collaborators, while others later received valuable contributions from the broader community. In some cases, industry adopters further developed the project and became maintainers of the tool.

This chapter briefly reviews my own contributions to the papers included in this thesis, as well as my role in the resulting open-source projects. I will then reflect on the new perspective on programming and data science tooling that emerges from the work, as well as future research directions inspired by the presented work.

14.1 Contributions to included publications

The publications selected for this thesis focus on an independent research direction that I have been pursuing after completing my PhD. I often developed the initial idea or a prototype, but many papers were the result of a broader collaboration or an attempt to integrate the idea with the work of my colleagues and collaborators.

- *Chapter 6 (Petricek et al., 2016)* – I developed the initial version of the library, developed the formal model, and wrote most of the paper. Gustavo Guerra significantly improved the initial implementation. Don Syme first developed the CSV type provider, assisted with the formalization, and provided the problem framing.
- *Chapter 7 (Petricek, 2017)* – I am the sole author of the paper, but some aspects of the work have benefited from discussion with Don Syme.
- *Chapter 8 (Petricek, 2020)* – I am the sole author of the paper, but the work has benefited from discussions with Dominic Orchard, Stephen Kell, Roly Perera and Jonathan Edwards and detailed feedback from Dominic Orchard.
- *Chapter 9 (Petricek et al., 2018)* – I developed the prototype implementation of the presented system and wrote most of the paper. Charles Sutton provided inspiration for work on provenance and James Geddes shaped the system design.

- *Chapter 10 (Petricek, 2022)* – I am the sole author, but valuable implementation work, adjacent to the paper, has been done by May Yong and Nour Boulahcen.
- *Chapter 11 (Petricek et al., 2023)* – The work on individual AI assistants has been done by the first five authors. I proposed the initial formal model and system architecture and led paper writing jointly with Christopher Williams.
- *Chapter 12 (Petricek, 2021)* I am the sole author of the paper, but the earliest form of the idea was born in discussion with Mathias Brandewinder.
- *Chapter 13 (Perera et al., 2022)* – The work was led by Roly Perera, Minh Nguyen contributed to implementation and formalization and Meng Wang to paper writing. I was involved in the original conceptual development with Roly Perera and writing.

14.2 Open-source software contributions

The ideas discussed in the earlier parts of the thesis have been implemented in several open-source software packages that are available under the permissive Apache 2.0 (F# Data) and MIT (all other projects) licenses.

- *F# Data* (<https://github.com/fsprojects/FSharp.Data>) has become a widely-used F# library for data access. It implements utilities, parsers, and type providers for working with structured data in multiple formats (XML, JSON, HTML, and CSV). I developed the initial version of the library and later described it in the paper included as Chapter 6. The library has since attracted over 100 industry contributors and further development has been led by industry maintainers including Gustavo Guerra, Colin Bull, Chet Husk, Taylor Wood, Steffen Forkmann, Don Syme and others.
- *The Gamma* (<https://github.com/the-gamma>) is a simple data exploration environment for non-programmers such as data journalists. It implements the iterative prompting mechanism for data access (Chapters 10 and 7) and live preview mechanism (Chapter 8). I created most of the implementation. May Yong, Nour Boulahcen, and Tom Knowles implemented support for further data sources and worked on case studies using the system. Live demos using the environment in a web browser can be found at <https://thegamma.net> and <https://turing.thegamma.net>.
- *Wrattler* (<https://github.com/wrattler>) is an experimental notebook system described in Chapter 9 that tracks dependencies between cells, makes it possible to combine multiple languages in a single notebook and hosts AI assistants for data wrangling described in Chapter 11. I created the initial prototype and oversaw later development done mainly by May Yong and Nick Barlow with contributions from Roly Perera, Camila Rangel Smith, Gertjan van den Burg, and others. More information can be found at <http://www.wrattler.org>.
- *Compost.js* (<https://github.com/compostjs>) is a composable library for creating data visualizations described in Chapter 12. Although the library is implemented in the F# language, it is compiled to JavaScript and provides a convenient interface for JavaScript users. I am currently the sole developer of the library, although it also served as a design inspiration for some aspects of Fluid (below). A range of demos illustrating the use of the library can be found online at <https://compostjs.github.io>.

- *Fluid* (<https://github.com/explorables-viz/fluid>) is a programming language for building linked data visualizations described in Chapter 13. The project has since been developed into a general-purpose language for transparent, self-explanatory research outputs by the Institute of Computing for Climate Science, University of Cambridge. The development has been led by Roly Perera with recent contributions from Joseph Bond and Achintya Rao. I participated in the project in an advisory role and collaborated on some of the research behind the implementation. A live example can be found at <https://f.luid.org>.

14.3 New look at data exploration

From a narrow technical perspective, the work presented in this thesis may be seen as an assorted list of contributions to a wide range of research areas including type systems, programming languages, provenance tracking, interactive programming environments, data wrangling, data visualization, and program analysis. But looking at the work from this perspective would be missing the forest for the trees. Collecting the individual contributions in a single body of work reveals two unifying themes behind the research.

The first unifying theme is the broader motivation. If society is to benefit from the increasing availability of open data and data processing capabilities, we must make working with data accessible to a broader audience. Experts who are not trained as programmers need to be able to gain valuable insights from data. They also need to be able to do so in ways that support transparency and openness and encourage critical engagement with data. Unlike in much programming languages research where the typical user is a professional programmer, the typical user for much of the work presented in this thesis has been a data journalist, who is exploring an interesting dataset in order to share relevant insights with the broader public.

The above motivation justifies a number of technical choices made in the presented work. I typically tried to make some aspect of programming simpler, reduce the complexity of programming or design, and develop tools that will assist with the task. The focus made it possible to restrict problems in ways that would, in other contexts, seem too constrained. Examples include the data exploration calculus, which does not let users introduce custom abstractions, and the iterative prompting mechanism, which restricts aggregations in a query to a fixed set of pre-defined operations. I believe in the value of restrictions like these. A programming language researcher is as much a designer as a scientist and designers “tend to (...) seek, or impose a ‘primary generator’ (...) which both defines the limits of the problem and suggests the nature of its possible solution” (Cross, 2007). The focus on simple tools for users like data journalists has been such ‘primary generator’ for some of the research presented in this work.

The second unifying theme of this thesis is methodological. The contributions presented here generally approach a problem related to working with data through the perspective of programming languages and systems research. I do not claim to be the first or the only one to view data science tooling from this perspective, but my work shows that the perspective can be fruitful for tackling problems across the entire data science lifecycle. In other words, I strongly believe there is a strong mutually beneficial relationship between programming languages and systems research and data science tooling. On the one hand, methods from programming languages and systems research can be used

to build new powerful data science tools. On the other hand, data science tools provide interesting challenges and design constraints that force us to rethink established assumptions in programming language research and can inspire new techniques and approaches. I also believe there is more to be done in the space explored by this thesis, both in terms of building simpler and more open data science tools and in terms of advancing programming language and systems research.

Despite the recent developments in large language models (LLMs), I believe that the direction outlined in this thesis is still the right one. In many of the systems presented in this thesis, my aim has been to make the code of a data analysis or data visualization as simple as possible, possibly to the point where a non-programmer would be able to read and understand it. With the rise of LLMs, the ability to review and understand code is becoming even more important. If we are faced with a data processing task and use a 100-line script generated by an LLM, it may be difficult to gain confidence in the results. But if we use a 10-line script in a language like The Gamma that has been generated with the assistance of an LLM, as recently explored by [Fromm \(2024\)](#), and if the step-by-step execution of the program can be inspected through live previews, gaining the confidence in the results may be much easier.

14.4 Towards programming systems research

Looking at the problem of data exploration from the perspective of programming languages is beneficial in both directions. On the one hand, the programming languages perspective lets us see the problem in new ways and develop new, simple, practical, and more principled tools for data exploration. This has been the subject of the present thesis. On the other hand, a close look at how data scientists interact with programming tools also forces us to rethink how we conceptualize programming languages. We need to think less about *programming languages* and more about interactive and stateful *programming systems*. I started exploring this perspective in recent joint work with [Jakubovic et al. \(2023\)](#).

When working with data, data scientists often interleave writing of code, running it, and manual tweaking of data and script parameters. If we look merely at the programming languages they use, the work may seem uninteresting. But if we look at the rich interactions between the current state, scripts that data scientists are tweaking and what they see on the screen, we can see that data exploration is a remarkably interesting kind of programming practice. We should thus see programming more as an interaction with a stateful and interactive system than as the process of writing of textual code. To study programming from this perspective, we will need new formal models, that can account for the interactivity lacking from conventional programming language theories, as well as new research methodologies, which make it possible to contrast and evaluate different kind of interactions with the programming system. Data science can provide a convenient and familiar testbed for exploring this new perspective on programming.

Bibliography

- Martin Abadi and Luca Cardelli. 2012. *A theory of objects*. Springer Science & Business.
- Gregor Aisch, Amanda Cox, and Kevin Quealy. 2015. *You draw it: How family income predicts children's college chances*. <https://www.nytimes.com/interactive/2015/05/28/upshot/you-draw-it-how-family-income-affects-childrens-college-chances.html> New York Times.
- Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management*. IEEE, 423–424.
- Judie Attard, Fabrizio Orlandi, Simon Scerri, and Sören Auer. 2015. A systematic review of open government data initiatives. *Government Information Quarterly* 32, 4 (2015), 399–418. <https://doi.org/10.1016/j.giq.2015.07.006>
- Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D³ data-driven documents. *IEEE Transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309.
- Liliana Bounegru and Jonathan Gray. 2021. *The Data Journalism Handbook: Towards a Critical Data Practice*. Amsterdam University Press.
- Nadieh Bremer and Marlieke Ranzijn. 2015. Urbanization in East Asia between 2000 and 2010. <http://nbremer.github.io/urbanization/>
- A. Buja, J. A. McDonald, J. Michalak, and W. Stuetzle. 1991. Interactive data visualization using focusing and linking. In *Proceedings of Visualization '91*. 156–163. <https://doi.org/10.1109/VISUAL.1991.175794>
- Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (jul 2021), 98–106. <https://doi.org/10.1145/3469279>
- Nigel Cross. 2007. *Designerly ways of knowing*. Birkhauser Verlag GmbH, Basel.
- Evan Czaplicki. 2016. *A Farewell to FRP: Making signals unnecessary with The Elm Architecture*. <https://elm-lang.org/news/farewell-to-frp>
- William Davies. 2017. How statistics lost their power—and why we should fear what comes next. *The Guardian* (19 January 2017). <https://www.theguardian.com/politics/2017/jan/19/crisis-of-statistics-big-data-democracy>
- Jonathan Edwards. 2015. *Transcript: End-User Programming Of Social Apps*. <https://www.youtube.com/watch?v=XBpwysZtkkQ> YOW! 2015.

- Jonathan Edwards and Tomas Petricek. 2021. Typed Image-based Programming with Structure Editing. CoRR abs/2110.08993 (2021). arXiv:2110.08993 <https://arxiv.org/abs/2110.08993> Presented at Human Aspects of Types and Reasoning Assistants (HATRA'21), Oct 19, 2021, Chicago, US.
- Jonathan Edwards, Tomas Petricek, and Tijs van der Storm. 2025. Schema Evolution in Interactive Programming Systems. *Art Sci. Eng. Program.* 9, 2 (2025). Issue 1. <https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2025/9/2>
- Mikolas Fromm. 2024. Design of LLM Prompts for Iterative Data Exploration.
- Murdoch J. Gabbay and Aleksandar Nanevski. 2013. Denotation of contextual modal type theory (CMTT): Syntax and meta-programming. *Journal of Applied Logic* 11, 1 (2013), 1–29. <https://doi.org/10.1016/j.jal.2012.07.002>
- Richard P. Gabriel. 2012. The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) (Onward! 2012). Association for Computing Machinery, New York, NY, USA, 195–214. <https://doi.org/10.1145/2384592.2384611>
- Jeremy Gibbons. 2010. Editorial. *Journal of Functional Programming* 20, 1 (2010), 1–1. <https://doi.org/10.1017/S0956796809990256>
- J. Heer, J. M. Hellerstein, and S. Kandel. 2015. Predictive Interaction for Data Transformation. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.
- IBM. 2020. *The Data Science Lifecycle: From experimentation to production-level data science*. <https://public.dhe.ibm.com/software/data/sw-library/analytics/data-science-lifecycle/>
- Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. <https://doi.org/10.1145/503502.503505>
- Shaveta Jain and Agrawal Kushagra. 2022. Comprehensive Survey on Data science, Lifecycle, Tools and its Research Issues. In *2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON)*, Vol. 1. 838–842. <https://doi.org/10.1109/COM-IT-CON54601.2022.9850751>
- Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Eng. of Programming* 7, 3 (2023), 1–13.
- S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. Van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono. 2011. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization* 10, 4 (2011), 271–288.
- Helen Kennedy, Martin Engebretsen, Rosemary L Hill, Andy Kirk, and Wibke Weber. 2021. Data visualisations: Newsroom trends and everyday engagements. *The Data Journalism Handbook: Towards a Critical Data Practice* (2021), 162–173.

- Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *20th International Conference on Electronic Publishing*, Fernando Loizides and Birgit Schmidt (Eds.). 87–90. <https://doi.org/10.3233/978-1-61499-649-1-87>
- David Koop and Jay Patel. 2017. Dataflow Notebooks: Encoding and Tracking Dependencies of Cells. In *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*. USENIX Association.
- Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. DBpedia—a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic web* 6, 2 (2015), 167–195.
- J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) (POPL '88). Association for Computing Machinery, New York, NY, USA, 47–57. <https://doi.org/10.1145/73560.73564>
- Sean McDermid. 2007. Living it up with a live programming language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) (OOPSLA '07). Association for Computing Machinery, New York, NY, USA, 623–638. <https://doi.org/10.1145/1297027.1297073>
- Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
- Stefan K. Muller and Hannah Ringler. 2020. A rhetorical framework for programming language evaluation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward! 2020). Association for Computing Machinery, New York, NY, USA, 187–194. <https://doi.org/10.1145/3426428.3426927>
- Greg Myre. 2016. *If Michael Phelps Were A Country, Where Would His Gold Medal Tally Rank?* <https://www.npr.org/sections/thetorch/2016/08/14/489832779/>
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 23. <https://doi.org/10.1145/1352582.1352591>
- Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. 2004. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20, 17 (2004), 3045–3054.
- Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (jan 2019), 32 pages. <https://doi.org/10.1145/3290327>

- Raymond R. Panko. 2015. What We Don't Know About Spreadsheet Errors Today. In *Proceedings of the EuSpRIG 2015 Conference "Spreadsheet Risk Management"*. European Spreadsheet Risks Interest Group, 1–15.
- Roly Perera, Minh Nguyen, Tomas Petricek, and Meng Wang. 2022. Linked visualisations via Galois dependencies. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. <https://doi.org/10.1145/3498668>
- Tomas Petricek. 2017. Data Exploration through Dot-driven Development. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPICs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:27. <https://doi.org/10.4230/LIPICs.ECOOP.2017.21>
- Tomas Petricek. 2020. Foundations of a live data exploration environment. *Art Sci. Eng. Program.* 4, 3 (2020), 8. <https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2020/4/8>
- Tomas Petricek. 2021. Composable data visualizations. *J. Funct. Program.* 31 (2021), e13. <https://doi.org/10.1017/S0956796821000046>
- Tomas Petricek. 2022. The Gamma: Programmatic Data Exploration for Non-programmers. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2022, Rome, Italy, September 12-16, 2022*, Paolo Bottoni, Gennaro Costagliola, Michelle Brachman, and Mark Minas (Eds.). IEEE, 1–7. <https://doi.org/10.1109/VL/HCC53370.2022.9833134>
- Tomas Petricek, James Geddes, and Charles Sutton. 2018. Wrattler: Reproducible, live and polyglot notebooks. In *10th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2018, London, UK, July 11-12, 2018*, Melanie Herschel (Ed.). USENIX Association.
- Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. Types from data: making structured data first-class citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 477–490. <https://doi.org/10.1145/2908080.2908115>
- Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. 2023. AI Assistants: A Framework for Semi-Automated Data Wrangling. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 9295–9306. <https://doi.org/10.1109/TKDE.2022.3222538>
- Simon L. Peyton Jones and Philip Wadler. 1993. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, South Carolina, USA) (POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. <https://doi.org/10.1145/158511.158524>
- João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. 2015. Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow. In *Workshop on the Theory and Practice of Provenance (TaPP)*. 155–167.
- Tye Rattenbury, Joseph M Hellerstein, Jeffrey Heer, Sean Kandel, and Connor Carreras. 2017. *Principles of data wrangling: Practical techniques for data preparation*. O'Reilly.

- Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (2019). <https://doi.org/10.22152/programming-journal.org/2019/3/1>
- Advait Sarkar and Andrew D Gordon. 2018. How do people learn to use spreadsheets?. In *Proceedings of the Psychology of Programming Interest Group (PPIG)*, Mariana Marasoiu Emma S oderberg, Luke Church (Ed.).
- Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350.
- C. A. Sutton, T. Hobson, J. Geddes, and R. Caruana. 2018. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. In *24th ACM SIGKDD Conference*.
- Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, et al. 2012. *Strongly-typed language support for internet-scale information sources*. Technical Report MSR-TR-2012-101. Microsoft Research.
- Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. 2013. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming (DDFP '13)*. ACM, New York, NY, USA, 1–4. <https://doi.org/10.1145/2429376.2429378>
- S. Thirumuruganathan, L. Berti-Equille, M. Ouzzani, J.-A. Quiane-Ruiz, and N. Tang. 2017. UGuide: User-guided discovery of FD-detectable errors. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '17)*. 1385–1397.
- Gerrit JJ van den Burg, Alfredo Nazábal, and Charles Sutton. 2019. Wrangling messy CSV files by detecting row and type patterns. *Data Mining and Knowledge Discovery* 33, 6 (2019), 1799–1820.
- Jacob VanderPlas, Brian E. Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Iliia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *The Journal of Open Source Software* 3, 32 (2018). <https://doi.org/10.21105/joss.01057>
- Bret Victor. 2012a. *Inventing on Principle*. <http://worrydream.com/InventingOnPrinciple>
- Bret Victor. 2012b. *Learnable programming: Designing a programming system for understanding programs*. <http://worrydream.com/LearnableProgramming>
- Richard Wesley, Matthew Eldridge, and Pawel T. Terlecki. 2011. An analytic data engine for visualization in tableau. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (Athens, Greece) (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1185–1194. <https://doi.org/10.1145/1989323.1989449>
- Hadley Wickham. 2010. A layered grammar of graphics. *Journal of Computational and Graphical Statistics* 19, 1 (2010), 3–28.

Hadley Wickham. 2016. *ggplot2: Elegant graphics for data analysis*. Springer.

Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, et al. 2019. Welcome to the Tidyverse. *Journal of open source software* 4, 43 (2019), 1686.

Leland Wilkinson. 1999. *The grammar of graphics*. Springer-Verlag New York. <https://doi.org/10.1007/978-1-4757-3100-2>