Simple programming tools for data exploration

Tomas Petricek

June 8, 2024

Preface

Cambridge, MSR, Kent, ATI
DNI
CUNI

Contents

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.

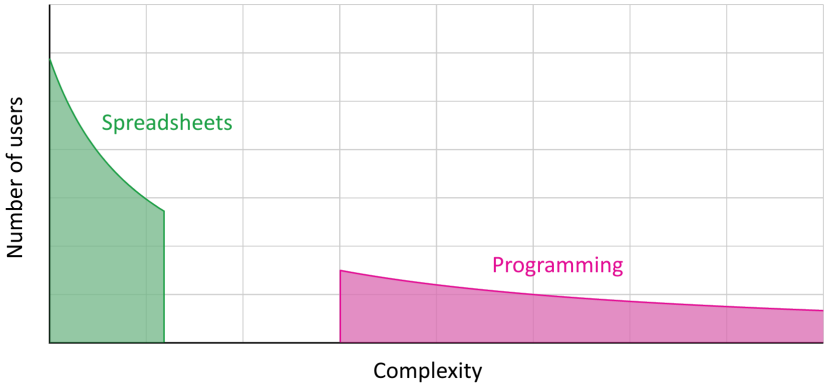

Figure 1.1: The gap between programming and spreadsheets – spreadsheets can be used by many people, but solve problems of a limited complexity. Programming scales arbitrarily, but has a high minimal complexity limiting the number of users. Adapted from Edwards (2015).

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

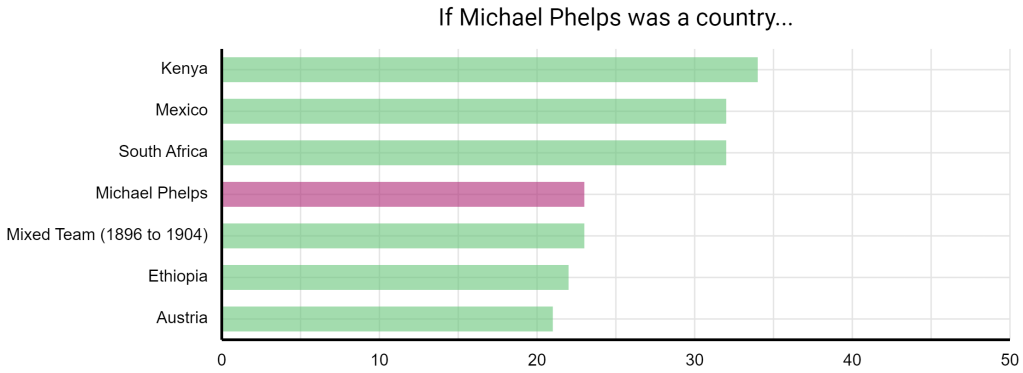

Figure 1.2: A visualization comparing the number of gold Olympic medals won by Michael Phelps with countries that won a close number of gold medals. Inspired by e.g., Myre (2016)

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 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 of 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 Figrue 1.2, which puts Phelps on a chart showing countries with similar number of medals. Even such simple visualization raises multiple questions. Is the table counting Gold medals or all medals? 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. It has to acquire and and clean the source data, aggregate medals by country and produce 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 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

```
let data = olympics.'group data'.'by Team'.'sum Gold'.then
    .'sort data'.'by Gold descending'.then
    .paging.skip(42).take(6)
    .'get series'.'with key Team'.'and value Gold'

let phelps = olympics.'filter data'.'Athlete is'.'Michael Phelps'.then
    .'group data'.'by Athlete'.'sum Gold'.then
    .'get series'.'with key Athlete'.'and value Gold'

charts.bar(data.append(phelps).sortValues(true))
    .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 language that can be read and understood without sophisticated programming skills. 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 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 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 pick a different athlete or increase the number of displayed countries. Such engagement has the potential to aid their positive percpetion 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 above criteria point at the gap between spreadsheets and regular programming systems illustrated by Figure 1.1 and there are multiple possible solutions and approaches to satisfy the above 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 langauges) 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 simpify 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 environments 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 programmatic interfaces, 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 hotchpotch of systems and tools is a challenge. The view taken in this thesis is that, most of the systems and tools can be considered as programming systems. This view makes it possible to apply the powerful methodology of programming languages research to the problem of data exploration. However, the programs that are constructed during data exploration have a number of specific characteristics that distinguishes them from programs typically considered in programming langauge 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 generally hold for simple data explorations that are done by data journalists, public servants and analysts that this thesis is primarily concerned with. The characteristics do not apply to all programs that work with data, which may include complex reusable and parameterized models, general-purpose algorithms and rich data processing pipelines. However, focusing on simple data exploration problems 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 to be included in a formal model of a programming language (in Chapter 8, an invocation of an external function becomes more important than lambda abstraction).

1.4 Utilized 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 langauge 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 langauge 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 langauge that impements it can be used to write programs that exhibit certain kind of incorrect behaviour.

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 laguage 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 reproducibe software artifact.

I use the programming systems research methodology primarily in Chapter 9, which presents the architecture and an 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 communitites 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 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 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 contributions of 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, Figure 1.4 shows where they fit in the data science lifecycle as defined by IBM (with slight modifications). As the diagram shows, the focus of this thesis is on work done in the initial data exploration phase.

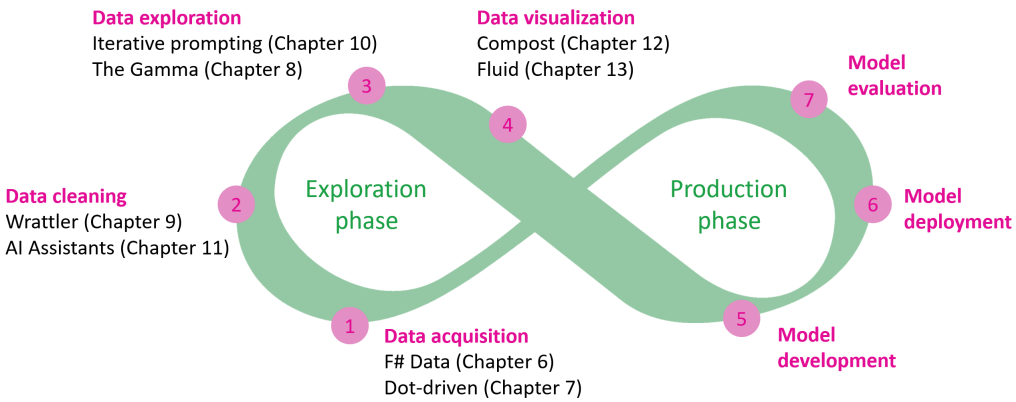

Figure 1.4: Illustration showing the data science lifecycle, as understood by IBM (2020), alongside with the contributions of this thesis to the individual steps of the data exploration phase.

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, incompete 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.

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 exporation and one from the domain of data cleaning. The rest of this section summarizes 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 langauge. 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 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 adress 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, 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 algorithnm.

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 exporation 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 parameters of the tool. The augmented tools are evalauted 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 small number of combinators (such 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 formalised and shown to have a desirable formal structure. The technique is used as the basis for a high-level programming langauge Fluid.

1.7 Open-source software contributions

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 concepts discussed in the previous section 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 became 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). The initial version based on the work presented in Chapter 6 has since been extended by multiple contributors from the F# community.

- *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). 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 langauges in a single notebook and hosts AI assistants for data wrangling described in Chapter 11. 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. A range of demos illustrating the use of the library can be found online at https://compostjs.github.io.
- *Fluid* (https://github.com/explorable-viz/fluid) is a programming langauge for building linked data visualizations described in Chapter 13. The project has since been developed into a general-purpose langauge for transparent, self-explanatory research outputs by the Institute of Computing for Climate Science, University of Cambridge. A live example can be found at https://f.luid.org.

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 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 langauges 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 coversions. 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 langauge 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("titel").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.

```
olympics = pd.read_csv("olympics.csv")
olympics[olympics["Games"] == "Rio (2016)"]
  .groupby("Athlete")
  .agg({"Gold": sum})
  .sort_values(by="Gold", ascending=False)
  .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 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 on 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 access read a database schema or access 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

```
type WorldBank = WorldBankDataProvider<"World Development Indicators">
let data = WorldBank.GetDataContext()

data.Countries.``United Kingdom``.Indicators
  .``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.

returns an 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 auhtor 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 Inicators"`) 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 langugae 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.

```
// worldbank.json - a sample response used for schema inference
[ { "page": 1, "pages": 1, "per_page": "1000", "total": 53 },
  [ { "indicator": { "id": "GC.DOD.TOTL.GD.ZS" },
      "country": { "id": "CZ" },
      "date": "2011", "value": null },
    { "indicator": { "id": "GC.DOD.TOTL.GD.ZS" },
      "country": { "id": "CZ" },
      "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 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 have 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 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 analysed 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:

$$\begin{array}{rcl} \hat{\sigma} & = & \nu\,\{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} \\ & \mid & \texttt{float} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{string} \\ \sigma & = & \textbf{nullable}\langle\hat{\sigma}\rangle \mid [\sigma] \mid \textbf{any} \mid \textbf{null} \mid \bot \end{array}$$

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 consist of primitive shapes (inferred from a corresponding value) and a record shape. The record shape has an (optional) name ν and consists of a 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 $\bot$, representing no information about the shape. The anove definition does not include handling of choice shapes (corresponding to sum types), which is introduced later.

A key techincal operation of the shape inference is expressed using the *common preferred shape* function written as $\sigma_1 \triangledown \sigma_2 = \sigma$. Given two shapes, the function returns a shape the most specific shape that can be used to represent 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.

- $\texttt{int} \triangledown \texttt{float} = \texttt{float}$ 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.
- $\{x : \texttt{int}\} \triangledown \{x : \texttt{int}, y : \texttt{int}\} = \{x : \texttt{int}, y : \textbf{nullable}\langle\texttt{int}\rangle\}$ 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 modelled 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:

$$\begin{array}{rcl} d & = & i \mid f \mid s \mid \textbf{true} \mid \textbf{false} \mid \textbf{null} \\ & \mid & [d_1; \ldots; d_n] \mid \nu\,\{\nu_1 \mapsto d_1, \ldots, \nu_n \mapsto d_n\} \end{array}$$

The shape inference is then defined as a function $S(()d_1, \ldots, d_n) = \sigma$ 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 case where values are of incompatible shape, the system infers the shape **`any`**.)

$$
\begin{array}{rclrclrcl}
S(i) & = & \texttt{int} & S(\textbf{null}) & = & \textbf{null} & S(\textbf{true}) & = & \texttt{bool} \\
S(f) & = & \texttt{float} & S(s) & = & \texttt{string} & S(\textbf{false}) & = & \texttt{bool}
\end{array}
$$

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

$$S(\nu\,\{\nu_1 \mapsto d_1, \ldots, \nu_n \mapsto d_n\}) = \nu\,\{\nu_1 : S(d_1), \ldots, \nu_n : S(d_n)\}$$

$$S(d_1, \ldots, d_n) = \sigma_n \text{ where } \sigma_0 = \bot,\ \forall i \in \{1..n\}.\ \sigma_{i-1} \triangledown S(d_i) \vdash \sigma_i$$

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 $\triangledown$ 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 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:

$$[\![\sigma]\!] = (\tau, e, L) \qquad (\text{where } L, \emptyset \vdash e : \texttt{Data} \to \tau)$$

The mapping $[\![\sigma]\!]$ takes an inferred shape σ and returns a tripple 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 langauge research. The second describes the stability of provided types and concerns 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 $\sigma_1 \triangledown \sigma_2 = \sigma_2$). 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 $d_1, \ldots, d_n$ are samples, $\sigma = S(d_1, \ldots, d_n)$ is an inferred shape and $\tau, e, L = [\![\sigma]\!]$ 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 usable 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 produced 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 shape 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 $d_1, \ldots, d_n$, a provided type based on the samples $\tau_1, e_1, L_1 = [\![S(d_1, \ldots, d_n)]\!]$ 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 $\tau_2, e_2, L_2 = [\![S(d_1, \ldots, d_n, d_{n+1})]\!]$.*

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* $C[$**`match`** e **`with`** `Some`$(v) \rightarrow v \mid$ `None` $\rightarrow$ `exn`$]$

(ii) $C[e]$ *to* $C[e.$`M`$]$ *where* `M` $=$ `tagof`(σ) *for some* σ

(iii) $C[e]$ *to* $C[$`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

[1] The 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.

```
olympics
  .'filter data'.'Games is'.'Rio (2016)'.then
  .'group data'.'by Athlete'.'sum Gold'.then
  .'sort data'.'by Gold descending'.then
  .'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 furhter 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 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 Formalizing 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):

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

When the type provider is invoked, it takes the schema and generates a class for querying data of the given schema. The types of the 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 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):

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

Here, $\textbf{type}\ C(x : \tau) = \overline{m}$ is a definition of a class C that consists of a sequence of members $\overline{m}$ 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 modelled 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 the 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 $\texttt{pivot}(F)$, which is similar to the function $[\![\sigma]\!]$ defined for the F# Data type providers:

$$\begin{aligned}&\texttt{pivot}(F) = C, \{C \mapsto (\textbf{type}\ C(x : \texttt{Query}) = \ldots, L)\}\\ &\quad \text{where } F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}\end{aligned}$$

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 $\texttt{drop}(F)$ 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 memebr `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:

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

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 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 and are expressed using the primitives of the relational algebra, will always be correct. They will never result in an undefined runtime behaviour that one may otherwise encounter when accidentally accessing a non-existent field. This is important result, because the sequence of operations transforms the fields in an interesting way. Operations like `drop` remove fields from the schema, while `group by` not only changes the set of fields, but also 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 $D = \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,m}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,m}\rangle\}$. A specific kind of data value is a data series $\texttt{series}\langle\tau_k, \tau_v\rangle(D)$ 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 $F = \{f_1 \mapsto \tau_1, \ldots, f_n \mapsto \tau_n\}$, let $C, L = \texttt{pivot}(F)$ then for any expression e that does not contain relational algebra operations or `Query`-typed values as sub-expression, if $L; x : C \vdash e : \texttt{series}\langle\tau_1, \tau_2\rangle; L'$ then for all $D = \{f_1 \mapsto \langle v_{1,1}, \ldots, v_{1,m}\rangle, \ldots, f_n \mapsto \langle v_{n,1}, \ldots, v_{n,m}\rangle\}$ such that $\vdash v_{i,j} : \tau_i$ it holds that $e[x \leftarrow C(D)] \rightsquigarrow^*_{L'} \texttt{series}\langle\tau_k, \tau_v\rangle(\{f_k \mapsto k_1, \ldots, k_r, f_v \mapsto v_1, \ldots, v_r\})$ such that for all $j \vdash k_j : \tau_k$ and $\vdash v_j : \tau_v$.*

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 labelled 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 Conclusions

In this chapter, I offered a brief overivew of the work on type providers that is included in Part II of this thesis. 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 accessing semi-structured data in XML, JSON

and CSV formats (Chapter 6) and the pivot type provider that makes it possible to express queries over tabular data (Chapter 7).

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 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 a 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 langauge 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 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, 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 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 a 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 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 langauges and tools and also uses the dependency graph to ensure reproducibility of the data explorations.

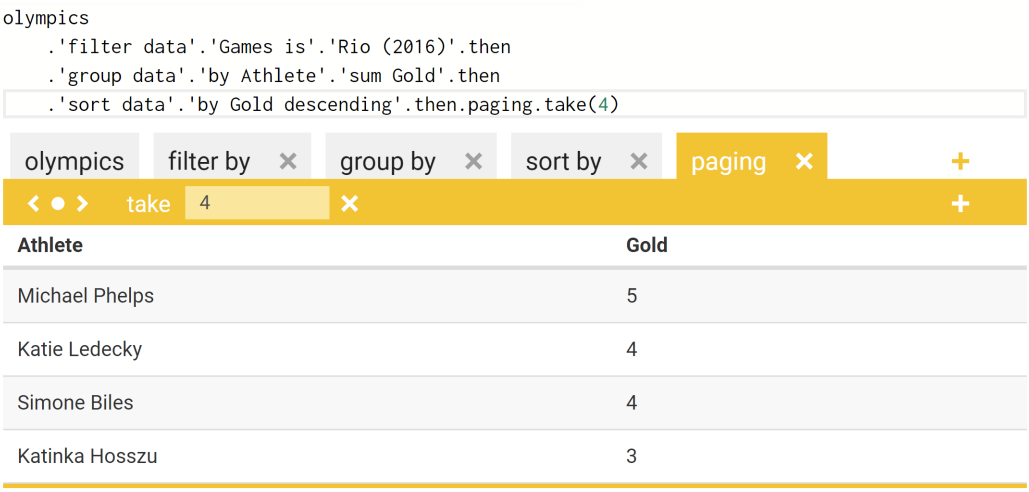

Figure 3.1: A live preview in The Gamma, generated for a code snippet that uses the pivot type provider for data exploration. The interface also lets the user navigate through the steps of the transformation and modify parameters of the query.

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 environment 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 illustratd in Figure 3.1. However, providing live previews in a text-based programming environment is a challenge (McDirmid, 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 langauge design.
- *Responding to code changes*. Code in a text editor can change in arbitrary ways and so it is unclear how to update 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 fucntions (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 analysed and linked to nodes in a dependency graph. When the node of the tree corresponds to dependency graph node that has been created previously (with the same dependencies), the graph node is

Programs, commands, terms, expressions and values

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

Evaluation contexts of expressions

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

Let elimination and member reduction

$$
\begin{array}{l}
o_1;\; \ldots;\; o_k;\; \texttt{let}\; x = o;\; c_1;\; \ldots; c_n \rightsquigarrow \\
\qquad o_1;\; \ldots;\; o_k;\; o; c_1[x \leftarrow o];\; \ldots; c_n[x \leftarrow o]
\end{array}
\quad \text{(let)}
$$

$$
\frac{o.m(v_1, \ldots, v_n) \rightsquigarrow_\epsilon o'}{C_p[o.m(v_1, \ldots, v_n)] \rightsquigarrow C_p[o']} \quad \text{(external)}
$$

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 argument 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 $o_1;\ \ldots;\ o_n$ (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:

$$\begin{array}{l}\textbf{let}\ l = \texttt{list.range}(0, 10)\\ l.\texttt{map}(\lambda x \rightarrow \texttt{math.mul}(x, 10))\end{array}$$

An external library provides the `list` and `math` objects, as well as numbers n, lists of objects $[o_1, \ldots, o_k]$ and failed computations $\bot$. 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:

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

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 $\bot$.

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 $o.m(v_1, \ldots, v_n)$ 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 $\bot$ object. Compositionality implies deterministic behaviour 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 $n, o_1, \ldots, o_n$ such that $p \rightsquigarrow^* o_1; \ldots; o_n$ 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 formalise an efficient mechanism for showing live previews during the editing of data exploxation 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.

(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 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 labelled 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:

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

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 labelled with arg indicating the index of the argument (0 for the instance and $1, 2, 3, \ldots$ 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 `take`(x) depends on the argument – the number 15 – and the instance, which is a node representing `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 mem$(\texttt{skip}, s_0)$ 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 arg(1) dependency of the `take` call changed and so we create a new node mem$(\texttt{skip}, s_2)$ 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 preview 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 $v_1, v_2, \ldots, v_n$ 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:

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

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

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 memeber 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 suitable value for the variable. In this case, our mechanism evaluates a delayed preview $[\![e]\!]_\Gamma$, 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 interesting theoretical connection to work on Contextual Modal Type Theory (Nanevski et al., 2008) and comonads (Gabbay and Nanevski, 2013).

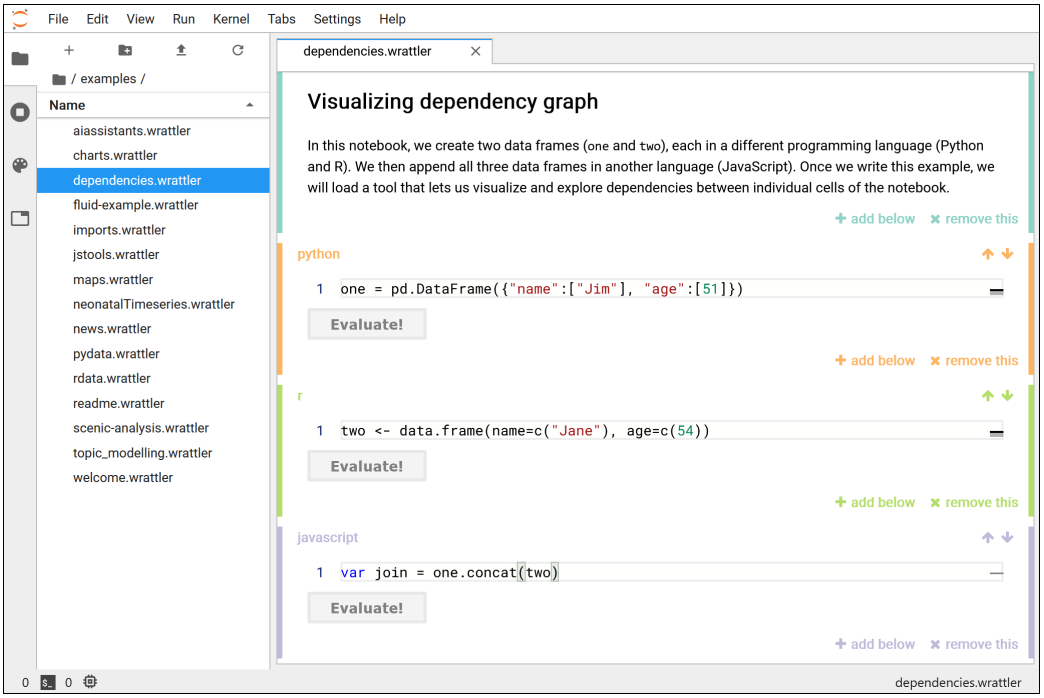

Figure 3.4: Wrattler running inside the JupyterLab system. The opened notebook passes data between cells written in three different programming languages (Python, R and JavaScript).

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 formalisation 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 tackled by the open-source Wrattler notebook system presented in full in Chapter 9. Wrattler is an extension for the industry standard JupyterLab platform. As illustrated in Figure 3.4, Wrattler adds a new type of document format that allows programmers to mix cells written in multiple different programming languages in a

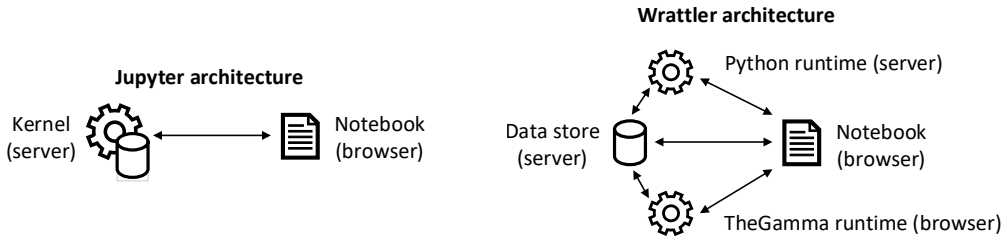

Figure 3.5: In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are split between data store and language runtimes. Language runtimes can run on the server-side (e.g. Python) or client-side (e.g. The Gamma).

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 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 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. 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 langauge runtimes.

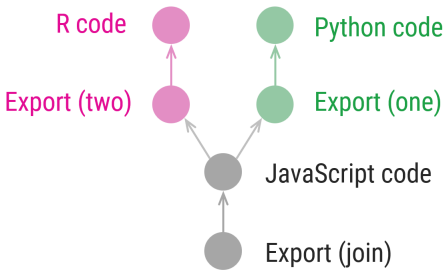

Figure 3.6: Dependency graph of a notebook from Figure 3.4. For each cell, the graph contains a code node and one (or possibly more) export nodes that represent exported data frames. The R and Python cells are independent and map to independent graph node. The node corresponding to the final JavaScript 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 a 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 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 case of Python, R and JavaScript, this is done using a lightweight code analysis. In case of The Gamma, which can also be used in Wrattler, the full parse tree and its associated dependency graph is 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 langauge 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 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 Conclusions

xx

xx

Chapter 4

Iterative prompting

Rattenbury et al. (2017) 50-80 with messy data. Surveys **?** indicate that up to 80% of data engineering is spent on *data wrangling*, a tedious process of

Chapter 5

Data visualization

what? what?

Part II

Publications: Type providers

Chapter 6

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

Chapter 7

Data exploration through dot-driven development

Part III

Publications: Data infrastructure

Chapter 8

Foundations of a live data exploration environment

Chapter 9

Wrattler: Reproducible, live and polyglot notebooks

Part IV

Publications: Iterative prompting

Chapter 10

The Gamma: Programmatic data exploration for non-programmers

Chapter 11

AI Assistants: A framework for semi-automated data wrangling

Part V

Publications: Data visualization

Chapter 12

Composable data visualizations

Chapter 13

Linked visualisations via Galois dependencies

Part VI

Conclusion

Chapter 14

Contributions

thanks who did what (especially galois)

Chapter 15

Conclusion

if the society is to benefit...

AI and LLMs

in other words, technology has democratized opinions, but can it also democratize facts

Bibliography

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

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

Liliana Bounegru and Jonathan Gray. 2021. *The Data Journalism Handbook: Towards a Critical Data Practice*. Amsterdam University Press.

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

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.

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

Jeremy Gibbons. 2010. Editorial. *Journal of Functional Programming* 20, 1 (2010), 1–1. https://doi.org/10.1017/S0956796809990256

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

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.

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.

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 McDirmid. 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

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.

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.).

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

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, 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.