Data-centric programming in Haskell

Jakub Malý

4th Year Project Report
Computer Science and Physics
School of Informatics
University of Edinburgh

2024

Abstract

In this thesis, I analyse the feasibility of doing data science in Haskell. The language's applicability to data-centric computation is considered from two perspectives: that of an educational tool for teaching computation basics; and that of an alternative to other languages commonly used for data science.

The main output of this thesis is a data-centric computation tutorial in Haskell, designed for first year Computer Science students. This tutorial combines beginner-friendly features of functional languages such as simple syntax with Haskell's data science library ecosystem. The tutorial was also evaluated by this year's first year students, and the feedback was overall very positive.

The secondary output is a comparison of Haskell and Python via performance and feature benchmarks designed to compare the suitability of various languages for data science and machine learning. The results show that Haskell (both compiled and interpreted) is slower than Python. However, Haskell's support for streaming I/O allows us to use map-reduce computations on datasets larger than memory. Results show that streaming performance improves with dataset size, and that Haskell is a viable alternative for large datasets.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 746665
Date when approval was obtained: 2024-01-10
Details about the participants' information sheet and consent form are included in Appendix A.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jakub Malý*)

Table of Contents

Chapter 1

Introduction

In this thesis, I explore the viability of data-centric computations in Haskell. This chapter includes the motivation for why data-centric computation in Haskell is interesting based on previous work in this area (Section 1.1); outlines the aims of the thesis (Section 1.2); summarises the results (Section 1.3); and includes an outline (Section 1.4).

1.1 Background

1.1.1 Data-centric computation

The initial motivation for this honours project was to design a data-centric tutorial in Haskell for first-year computer science students at the University of Edinburgh (class INF1A—Introduction to Computation).

The idea of using data to motivate learning Computer Science comes from Fisler, who claims that it is important to teach computer science with data science in mind—many fields nowadays require basic skills in computing to enable people to work with data specific to their field.[1] During Fisler's talk at Lambda Days in 2021, they outlined how creating introductory computing curricula around data-centric computation better complemented the learning objectives of students in degrees outside of Computer Science.[2] Furthermore, this approach was also a good way to motivate learning core concepts in computer science theory from data structures to types, since these are necessary to understand in order to work with data in a programmatic context.

Earlier work by Krishnamurthi and Fisler claims that data analysis (explicit manipulation of aggregated data such as cleaning, splitting, or refactoring) is done through operations that resemble higher-order functions.[3] This means that functional syntax is better suited for expressing data-centric computation, since we can express operations as transformations of the data, rather than an iterative element-wise computation used in imperative languages.

To facilitate teaching data-centric computation, Fisler et al. wrote a textbook for Brown University's introductory computing class.[4] This textbook uses a bespoke language called Pyret, which combines features from common languages like Python with syntax often found in functional languages such as Haskell or OCaml.[5]

My work focuses on how we can take the same principles of data-motivated computation and apply them to Haskell, since the University of Edinburgh's introductory class uses Haskell. Unlike Pyret, which is still under development, Haskell is a well-established language with a stable user base.[6] This means that Haskell already has a relatively large collection of data science and machine learning libraries. A lot of these are collected under dataHaskell—an open-source organisation focused on creating an ecosystem of libraries that leverage Haskell's features such as its type system, lazy computation, and support for streams.[7]

1.1.2 Functional languages for beginners

While the idea of using data to motivate learning computation is relatively new, using functional programming as an introductory computing class is not. Long before the advent of imperative languages with less boilerplate like Python, lecturers were looking for more beginner-friendly languages that offered a higher level of abstraction. Back in the late 1980s, Joonsten et al. switched to teaching Miranda[8]—a lazy, purely functional language that strongly influenced the design of Haskell[9]—in their first semester introductory Computer Science class at the University of Twente in the Netherlands. Their findings suggest that students were better able to describe problems formally, had an easier time with abstractions, and solved more complex problems than their counterparts that were taught imperative programming. Moreover, they found that combining their teaching with imperative programming in semester two lead to the most well-rounded year one education.

A different study by Yuen et al. done on a group of high school students found that teaching logical programming made them see Computer Science as more than just programming—student surveys mentioned that they saw computer science as problem solving and the study of technologies more than just coding.[10] While there are differences between functional and logical programming, Paulson and Smith argue that they are both styles of declarative programming that offer similar benefits in abstraction and simple syntax.[11] Haskell offers a lot of the desirable features such as a relatively simple syntax structure with intuitive semantics and a natural connection to abstraction and logic. The INF1A course exploits this fact by already including logic as a core component alongside programming in Haskell.

While Haskell's syntax might be relatively simple, it is also important to keep in mind that the objective of an introductory computer science course should be an introductory overview to computing in general first, and a class in a specific language second. Studies by Chakravarty and Keller,[12] as well as Felleisen et al.[13] argue that the focus of an introductory computing class should not be on the syntax specific to the functional language that the course is being taught in. They claim that focus should primarily be placed on design concepts (that are language-agnostic), rather than design constructs (that are language-dependent). This might mean limiting the use of abstract concepts not seen in many languages, such as monads. Thankfully, imperative languages like Python (and even Java) are slowly adopting features from functional languages. This makes it easier to justify teaching constructs such as list comprehensions and pattern matching in Haskell, since they are becoming more commonplace.

There are also measurable benefits to using a language that can be used outside of an educational setting. Hughes suggests that students engage better with content that they perceive as being useful in the future more than content that is interesting from a purely academic perspective.[14] Therefore, teaching students to use a language that has a reasonably sized ecosystem and user base might engage their interest further than a purely educational language.[15] While a language like Pyret that is tailor-made for beginners can be easier to learn, it does incur a large opportunity cost because the language has no use case outside of the classroom.[1] Even Fisler's introductory computing textbook only uses Pyret for the first half of the course, after which it switches to Python. They argue that this transition should be relatively seamless because the two languages have similar syntax structure.[2;4] However, there is also the possibility that learning two mutually incompatible languages with very similar syntax can be confusing to students. Therefore, having a language like Haskell that borrows from both sides (it has relatively clear syntax, and a decently sized user base) might offer a good compromise.

1.2 Aims

The primary goal of this project was to design a Haskell tutorial about data-centric computation for first year informatics students at the University of Edinburgh. This required creating back-end code scaffolding that allowed the students to load, manipulate, summarise, and visualise tabular data (CSV files). Once this framework was completed, the second step was to write a tutorial for the INF1A class that demonstrates how this code framework can be used, its capabilities, and limitations. Then, code solutions to all exercises needed to be made so that students could check their work throughout the tutorial. Finally, the tutorial needed to be evaluated by the current INF1A cohort.

The secondary goals were to analyse the viability of using Haskell for data-centric computation, both as an educational tool and as a tool for data science. This analysis can be viewed from three perspectives: ease of use, feature availability, and performance. Therefore, in order to compare Haskell to other languages used for data science, the project needs to define methodology based on which comprehensive and unbiased comparisons to other languages can be made.

1.3 Results

1.3.1 Tutorial

I have created a code scaffold that supports data-centric computation using libraries included in the dataHaskell environment. I then used this framework to create a tutorial consisting of seven exercises which ask questions about a dataset to motivate learning

[1]The Pyret website does state that the end goal for Pyret is to become a fully featured language.[5] However, there is no guarantee that the language will develop a stable user base.

and revising material on functional programming students learned in class. The tutorial also incorporates data science concepts into both questions and implementation.

To evaluate the efficacy of the tutorial, I sent the tutorial out to the current INF1A cohort to collect feedback, which I then used to make adjustments for the final tutorial version. The feedback shows that the tutorial is well designed: students were able to learn new data science concepts and solidify their functional programming skills; the dataset was interesting; and students used critical thinking when engaging with the material.

1.3.2 Benchmarks

To test Haskell's viability as a language for data science in general, I implemented tests from the Sanzu benchmark in Haskell. I ran benchmarks in both Python and Haskell on synthetic datasets at five orders of magnitude (10^3 to 10^7 rows) to investigate how performance scales with dataset size. I also ran memory profiling to probe whether Haskell can stream datasets that are larger than memory.

Implementing these tests showed that Haskell's ecosystem is not as feature-rich or user-friendly when compared to more mainstream languages such as Python. Benchmarks performed on datasets at five orders of magnitude show that Haskell (both compiled and interpreted) performs worse than Python on all tests, and that both languages scale at a similar rate with respect to dataset size for most tests.

Memory profiling benchmarks show that memory used when streaming is independent of dataset size, and that the performance of streaming a map-reduce operation on a dataset is comparable to loading the dataset to memory and then using the map-reduce operation on a column. Further benchmarks suggest that as dataset sizes grow, streaming becomes more efficient due to higher productivity (less time spent on garbage collection), making streams a better paradigm for dealing with large datasets.

Testing suggests that Haskell can be a good language to use for very large datasets.

1.4 Thesis outline

Chapter 2 gives a detailed overview of the data science tutorial in Haskell I created, justifies all design decisions made, and evaluates the success of the tutorial based on feedback collected from this year's INF1A cohort.

In Chapter 3, I outline the technical difficulties that I encountered while making the tutorial. Then, I use the Sanzu benchmark to compare Haskell's data science and machine learning capabilities with Python.[16] Introduced by Watson et al. in 2017, this benchmark was designed to test the performance and features of popular data science languages. The last part of this chapter includes benchmarks on streams, and compares them to in-memory data frames in Haskell.

Lastly, Chapter 4 contains the conclusions. These include a comprehensive synthesis of the viability of Haskell for teaching and data science, and future work needed to make Haskell easier to use for data-centric computation in the classroom.

Chapter 2

Making a Tutorial

This chapter contains a detailed overview of everything related to the data-centric tutorial I made for first-year computer science students at the University of Edinburgh. All design decisions made are justified with literature. I start by giving an overview of the tutorial file and text structure in Section 2.1, followed by my reasoning for the choice of dataset in Section 2.2. Section 2.3 talks about how the tutorial code scaffold was set up. Next, I include details about the educational aims and design considerations for each of the exercises (Section 2.4) and the solutions (Section 2.5). Finally, in Section 2.7 I talk about student feedback for the tutorial. The full tutorial text can be found in Appendix B, and all the tutorial and solutions code is included in Appendix C.

It is also important to understand the context within which the tutorial was designed. The Introduction to Computation (INF1A) course consists of both logic theory and coding in Haskell. As such, the students are assumed to have a semester's worth of experience in Haskell, but the specifics of assumed knowledge will be stated explicitly and explained whenever relevant. The tutorial also aims to lay down groundwork for data-centric computation while not duplicating the knowledge that is taught in the core second year class of the Computer Science degree called Foundations of Data Science (FDS) taught in Python.

2.1 Tutorial structure

The tutorial consists of a PDF file with all the instructions and exercises, a CSV dataset file, an example plot, and several Haskell files: the main tutorial file; two files with helper functions; and two solution files (the full file structure with descriptions is listed in Table 2.1).

The code is deliberately split into multiple files based on their functions. Scaffolding for the dataset (the `Frames` library) is located in the main `Tutorial.hs` file; functions that operate on columns are located in the `Helpers.hs` file; and plotting functions (wrappers for the `Chart` library) are placed in the `Plotter.hs` file. Non-code files are placed within their own directories to separate code from auxiliary files. The tutorial text is left outside of the tutorial folder, since this is the first file the students should

`tutorial/`	A directory that contains all the code.
`|--plots/`	A directory into which all plots are generated.
`|--|--number_of_astronauts.svg`	Example plot (included in tutorial text).
`|--datasets/`	The dataset directory.
`|--|--astronauts.csv`	The CSV(comma-separated value) dataset.
`|--Helpers.hs`	Scaffolding for operating on columns (lists).
`|--Plotter.hs`	Scaffolding for making plots (`Chart`).
`|--Tutorial.hs`	The main tutorial file, to be filled in by the student.
`|`	Includes scaffolding for dataset operations (`Frames`);
`|`	imports `Helpers` and `Plotter`.
`|--Solutions.hs`	Solutions and solution printers; imports `Tutorial`.
`|--AltSolutions.hs`	Streaming solutions alternative; imports `Tutorial`.
`tutorial.pdf`	The tutorial text.

Table 2.1: **File structure of the tutorial.** Files are hyperlinked to appendices.

open.

The tutorial text starts with a short overview, which shows the students what they can expect to be able to do by the end of the tutorial—load and analyse a dataset, and even create plots. This serves two purposes: it gives the students an idea of what to expect in the tutorial as well as motivates why the skills taught within might be interesting. This approach is motivated by a 2008 paper by Hughes, which suggests that students engage more in concepts that they perceive as having 'real-world applications'.[14] This is further supported by cognitive theory, such as the 1990 paper by Hidi.[15] They claim that learning is enhanced when a text leverages both innate interestingness of the material and interest due to personal significance (usefulness of the material for the future).

Then, the tutorial guides the students through how to set up their environment. I modelled this section after the first tutorial the students are given in their INF1A course, which guides them through how to install Haskell via `ghcup`. I found that using GHC (Glasgow Haskell Compiler) version 9.4.7 was preferable to the recommended version 9.4.8, since some libraries had issues with incompatible dependencies if installed in the newer version. The tutorial contains full instructions to install and switch compiler versions using `ghcup`, and install all libraries with `cabal` (Haskell package manager) in three commands that can be copy-pasted directly into the command line. This was done to minimise the amount of work needed for students to get started. The instructions were tested on Windows 10 and 11, Linux Ubuntu 22.04, and MacOS 14; both with fresh Haskell installations and an already in-use system. While these instructions are specific to this year's INF1A cohort, they use standard tooling and can be easily adapted.

The next tutorial section introduces the dataset (more detail in Section 2.2), gives a refresher on the basics of I/O in Haskell, and instructions on how to work with the dataset (loading, viewing, and getting a column). The students should have knowledge of basic input and output operations and how to handle side effects in Haskell, but this is a topic that is introduced very late into the course, so there is little time for the students to get much hands-on experience. As Hughes suggests, topics taught later in a course are viewed as harder by students because they have had less time to get familiar with the concepts.[14] This is why I spend almost a full page of the 11 page tutorial on reviewing how I/O works in GHCi (the GHC Read-Eval-Print-Loop) as well as writing and running `IO ()` functions.

The main body of the tutorial consists of guided exercises. These were designed to help students solidify their understanding of functional programming by applying what they have learned in class to a real-world problem—data analysis (see Section 2.4 for details).

The last two pages of the tutorial discuss when Haskell is useful compared to other data science languages.

2.2 Finding a suitable dataset

Before I could design any exercises, I first needed to select a dataset. The criteria for a good dataset look relatively simple on the surface, but each presents their own set of challenges. Ideally, for a data science tutorial, a dataset needs to:

1. **Be interesting enough for the students to engage with the material.**

   This point is relatively self-explanatory—using a dataset that contains information the students want to know more about will motivate them to engage with the material. A 1991 study by Schiefele found that interest in a topic was significantly correlated with involvement, enjoyment, concentration, and even levels of comprehension.[17] They also found that student interest has the strongest positive correlations with elaboration (finding concrete examples) and seeking information. We can leverage these benefits by asking interesting questions about the data and having students find the answers via data analysis.

2. **Have diverse enough data to facilitate the analysis of numerical, textual, and categorical data.**

   This requirement is needed to provide a more well-rounded learning experience. Most real-world datasets include all types of data, and teaching how to work with both numerical and textual data is an important foundation. Numerical data can be used to introduce concepts of summarising data, while textual (which oftentimes means categorical) data lends itself well to aggregation operations.

3. **Be either pre-processed or contain data that is clean by design.**

   Despite using a lot of data science, it is important to keep in mind that the purpose of this tutorial is not to teach data science, but rather to use data science to motivate students to use functional programming. As such, data cleaning is out of the scope of the tutorial.

My first idea was to use the an Edinburgh weather dataset (which is a part of the HadUK-Grid dataset collection).[18] I reasoned that this would be of interest to a lot of people given its relevance as a practical everyday interest. However, there were several issues with this choice. Firstly, I had no way to justify whether this dataset would be of interest to the students. Secondly, I realised that this could clash with a dataset that

Column name	Type	Description
`numberOverall`	`Int`	Astronaut world-wide number
`numberNationwide`	`Int`	Astronaut nation-wide number
`profileName`	`Text`	Astronaut full name
`profileSex`	`Text`	Astronaut sex (male/female)
`profileYearBirth`	`Int`	Astronaut birth year
`profileNationality`	`Text`	Astronaut nationality
`profileMilitary`	`Text`	Affiliation with military (military/civilian)
`profileYearSelection`	`Int`	Selection year
`profileMissionsNumber`	`Int`	Astronaut's mission number
`profileMissionsTotal`	`Int`	Astronaut's lifetime missions
`profileMissionsDuration`	`Double`	Astronaut's lifetime mission duration (hrs)
`profileMissionsEVA`	`Double`	Astronaut's lifetime EVA duration (hrs)
`missionRole`	`Text`	Astronaut's role on mission
`missionYear`	`Int`	Mission year
`missionName`	`Text`	Mission name
`missionVehiclesAscent`	`Text`	Ascent vehicle name
`missionVehiclesOrbit`	`Text`	Orbit vehicle name
`missionVehiclesDescent`	`Text`	Descent vehicle name
`missionDurationTotal`	`Double`	Mission duration (hrs)
`missionDurationEVA`	`Double`	Astronaut's EVA duration (hrs)

Table 2.2: **Description of all columns in the** `astronauts.csv` **dataset.** EVA stands for extravehicular activity.

is used for the second-year FDS class, and I wanted to have this tutorial fit into the students' learning with minimal duplication of knowledge.

In order to find a more objective way to select datasets, I used CORGIS (Collection Of Really Great, Interesting, Situated Datasets)—a project that includes pre-processed datasets that educators can use with minimal adaptation for their computing classes.[19] While this dataset collection is mostly intended for computing in Python and Java, they also offer raw CSV files. I ended up choosing a dataset of publicly-known information about astronauts who went to space prior to 2020, since it satisfied all three criteria for a good dataset.

I later found that the dataset was not cleaned properly, as some non-ASCII characters were missing from the name fields in the CORGIS dataset. I ended up downloading the raw dataset from the TidyTuesday GitHub[20] and cleaning it manually. This included reordering and renaming columns to be more descriptive and easier to work with in Haskell (formatted in camelCase), as well as checking for (and cleaning) faulty text in a few rows instead of simply dropping them (as was the approach that CORGIS took). The tutorial text also included a table that contained a column name, Haskell type, and a short description of what the data meant (including units for mission duration—yet another piece of information missing from the CORGIS website). This table is also included here (see Table 2.2).

There were a number of design considerations that the dataset makes, and they are explained in the tutorial, with the goal to explicitly point out as many good design decisions to the students as possible. As Felleisen et al. claim in their 2004 paper on computer science curricula: instructors should aim to explicitly teach design concepts, not just design constructs that the students have to assemble themselves.[13] This is supported by Fisler's paper, which posits that people construct new programs by recalling programs they have written or saw before and adapting the high-level structure to the current problems.[1] This means that explicitly highlighting good design practices helps students solve similar problems effectively in the future.

The first design choice that is explicitly explained to the students is the dataset structure. The dataset can be viewed as a list of rows, where each row corresponds to an observation (an astronaut that went to space on a specific mission) and each column contains a single piece of information. This way of structuring datasets was popularised by Wickham's `tidyverse` R packages, and is therefore referred to as tidy data.[21] As Wickham argues, this is a good way to structure a dataset since whenever a new observation is made, we only need to add a new row. Similarly, if we want to add a new piece of data to the dataset, we can simply add a new column. This dataset includes three column categories: index columns (which start with `number`), columns that contain information about the astronaut (starting with `profile`), and columns with information about the mission (start with `mission`).

Another explicit design choice was the use of `Text` instead of `String` to store textual data. `String` is the default Haskell implementation, and it is a type synonym for a list of characters `[Char]`, while `Data.Text` is a type defined in the `text` library and uses UTF-16 UNICODE encoding.[22] This means that while `String` is better for manipulating strings (since it is a list), it uses $5n$ words compared to `Text`'s 6 words + $2n$ bytes, where n is the number of characters in a string.[23] It is easy to miss this detail unless stated explicitly, since the tutorial uses the `XOverloadedStrings` pragma (additional compiler instruction), which renders `Text` and `String` literals the same way.

2.3 Effective scaffolding

The next big design decision was figuring out how to work with the dataset. There were two options to choose from: either implement something from scratch or use a library. The advantage of implementing something from scratch would be transparency for how files are loaded to the students. Early drafts of the tutorial included teaching the students how to work with I/O and parse their own CSV file. However, since the tutorial was supposed to be about solidifying concepts through data science, this idea was scrapped early. The same reasoning applied when I was deciding whether to choose a relatively light-weight CSV parsing library such as `cassava`,[24] or a more feature-rich library that included an efficient data frame implementation (in-memory representations of a table) such as `Frames`.[25]

Ultimately, I decided to use several data science libraries: `Frames` for data frames, `Chart` for plotting,[26] `statistics` for linear regression,[27] and `foldl`[28] and `pipes`[29] for streams. I could then guide the students through how to use a 'real-world' library

```
ghci> :load Tutorial.hs
ghci> astronauts <- loadDataset
ghci> getColumn astronauts profileName
```

Figure 2.1: **Getting a list of astronaut names in the GHCi REPL.**

```
ghci> printFrame " " (takeRows 3 f)
profileName missionName
"Gagarin, Yuri" "Vostok 1"
"Titov, Gherman" "Vostok 2"
"Glenn, John H., Jr." "MA-6"
ghci> printDataset 3 f
profileName           missionName
"Gagarin, Yuri"       "Vostok 1"
"Titov, Gherman"      "Vostok 2"
"Glenn, John H., Jr." "MA-6"
ghci> viewDataset 3 f
{profileName :-> "Gagarin, Yuri", missionName :-> "Vostok 1"}
{profileName :-> "Titov, Gherman", missionName :-> "Vostok 2"}
{profileName :-> "Glenn, John H., Jr.", missionName :-> "MA-6"}
```

Figure 2.2: **Comparison of different ways to view the frame** `f`. **TOP:** Using only functions that come with the `Frames` library. **MIDDLE:** Print the first *n* rows of a frame as aligned columns. **BOTTOM:** Print the first *n* records of a frame.

with the use of understandable scaffolding code. The students can then reuse the scaffolding, should they wish to use Haskell for data science themselves.

The `Frames` library is used in the tutorial to provide the data frame back-end. It only requires a few lines of code to get the dataset loaded, so these functions are kept within the main `Tutorial.hs` file. They include defining the row type, creating a stream of rows, and then loading the dataset into memory (`loadDataset`). I also wrote a helper function `getColumn` that extracts a column out of the dataset as a list. Figure 2.1 shows all the necessary steps to get all the astronaut names once GHCi is started. Notice how simple the syntax is—loading the dataset only requires a call to `loadDataset`, which performs an I/O action that loads the dataset to memory, and getting a column only requires them to call the `getColumn` function with a getter that the `Frames` library defines. These getters are eponymous with the column headers shown in Table 2.2, which was why it was important to properly name the dataset columns—Haskell variables cannot begin with uppercase letters, or contain dots.

Alongside the bare necessities (loading the dataset and extracting a column), I also wrote and included two helper functions to print each row as a record (`viewDataset`) and pretty print (`printDataset`) the dataset to help the students view the dataset structure more easily. Figure 2.2 shows a comparison between the library functions

```
getColumn :: (Foldable t, Functor t) => t a -> Getting b a b -> [b]
getColumn frame col = Data.Foldable.toList $ view col <$> frame
```

Figure 2.3: **Definition of the `getColumn` helper function.**

(`printFrame` and `takeRows`) compared to the more accessible helpers `viewDataset` and `printDataset`. The frame `f` is a two-column subset of the `astronauts` frame, which we can obtain by applying a lens onto each record (row), that selects the desired record fields from the larger row record (that contains all 20 fields corresponding to the 20 dataset columns). The students were encouraged to try to use these functions to interactively analyse of the dataset via the GHCi REPL (Read-Evaluate-Print Loop).

While helpers like `printDataset` might seem like very basic functions, column alignment is crucial for effectively viewing and exploring the dataset. Because GHCi is a command line environment, many systems simply wrap overlong lines. Therefore, while viewing only a few rows from a two-column subset might be legible, viewing even just a few lines from the full 20-column dataset quickly becomes extremely confusing.

Indeed, some of these scaffolding functions are nothing more than wrappers for a few basic functions. For example, the `getColumn` function shown in Figure 2.3 simply applies a column getter (a 1-field lens) to all records in the frame and then converts the one-column frame to a list. The question then becomes: are these helper functions even necessary? Is there a benefit to including simple function wrappers such as `getColumn` as part of the scaffolding?

I believe there is. The `getColumn` function abstracts a lot of detail that is necessary for working with the `Frames` library, while at the same time being superfluous to actually completing the tutorial exercises. Unlike what we might expect at first glance, the `view` function is defined in the `microlens` library,[30] and `Frames` does not include any mechanism for extracting a single column—this syntax is copied from the `Frames` tutorial.[31] Therefore, the purpose of these helper functions is to isolate the implementation details from the desired functionality (or, in Fisler's words: teaching should focus on the effects of the program, not its implementation).[1] The `getColumn` function abstracts the code necessary to work with `Frames` which is new to the students, and instead reverts to a standard Haskell list which they are familiar with. This means we do not need to explain new syntax (such as `fmap` and its infix equivalent `(<$>)`) needed to get a column from a data frame.

There is a large amount of literature that supports this view. Chakravarty and Keller argue that while functional programming languages are good candidates for beginners, care has to be taken to remain as language-agnostic as possible.[12] Another paper by Felleisen et al. also supports the idea that functional programming can lead to the same tyranny of syntax that imperative programming often results in.[13] This results in the focus being placed on the language-specific implementation rather than focusing on more generally-applicable knowledge. Therefore, effective scaffolding is not only about abstracting implementation details but also implementation syntax that the students do not need to know.

```
loadDataset :: IO (Frame Astronaut)
loadDataset = Frames.inCoreAoS Tutorial.astronautStream
```

Figure 2.4: **The type and implementation of the `loadDataset` function.**

```
inCoreAoS :: (primitive-0.8.0.0:Control.Monad.Primitive.PrimMonad m,
              Control.Monad.IO.Class.MonadIO m,
              exceptions-0.10.5:Control.Monad.Catch.MonadMask m,
              Frames.InCore.RecVec rs) =>
              Producer (Record rs) (SafeT m) () -> m (FrameRec rs)
```

Figure 2.5: **Type signature of the `Frames.inCoreAoS` function.**

Also notice that I took special care to adhere to a common naming convention—all helper functions have the form `actionObject` (such as functions that deal with the dataset `loadDataset`, `viewDataset` or the column to list function `getColumn`). This helps distinguish scaffolding functions from functions defined by the `Frames` library.

2.3.1 Understandable types

Another reason why scaffolding is important is that it lets us make functions with simple and understandable types. In Haskell, a function's type signature can tell us a lot about what that function does—it describes the full list of inputs types and the output type. The order of input arguments is sometimes relevant too, since Haskell supports partial function application.

Let us take the `loadDataset` function as an example. Figure 2.4 includes the full implementation of this function. Notice: it is very simple! The type signature immediately tells us that calling this function will perform an I/O action that will give us a `Frame` of `Astronaut` records (rows). The implementation simply calls a function that converts a stream into an in-memory array-of-structures (AoS) representation of the data frame. However, the type signature of the `inCoreAoS` function (shown in Figure 2.5) is too complex. Not only does it introduce concepts which are out of scope for the tutorial like monads, it also includes monads from three different libraries: the `base` Haskell library, the `primitive` library, and the `exceptions` library. Since this is not relevant to the tutorial (data-centric programming), and unexplained syntax in a tutorial is very bad practise, this type signature should not be seen by a student.

This example illustrates how writing scaffolding functions that have understandable (or easily explainable) types provides educational value. I introduce all functions in the tutorial with their type signature which is then explained, since looking at functions in terms of their types is a very useful skill to have, and one that is even more important for a functional language like Haskell.

```
ghci> (group . sort) ["a", "b", "a", "c", "b"]
[["a", "a"], ["b", "b"], ["c"]]
```

Figure 2.6: **A code example from the tutorial.** This example shows how the `group` and `sort` functions can be used together to aggregate elements into nested lists.

2.3.2 Code examples as a teaching tool

Unfortunately, the abstraction-oriented approach to code scaffolding I chose for this tutorial directly clashes with the aims stated in Section 2.1—abstracting the implementation details also means that the students are exposed to an idealised version of the code for the purpose of education, rather than using functions that come with the library.

In order to fix this disparity, we can take advantage of what Sweller describes as the worked example effect in his 2006 paper.[32] They argue that people construct new programs based on a high-level solution structure to similar problems. This approach is further supported by a 2015 paper about teaching types in programming by Tirronen et al., who found that providing students with worked examples can have a positive impact on learning.[33]

This is why the scaffolding code for working with `Frames` is not hidden in a helper file but instead remains at the top of the main `Tutorial.hs` file. These functions also provide a worked solution for the students to use, should they want to use the `Frames` library outside of this tutorial. At the same time, a student can simply read through the instructions in the tutorial text on how to use these functions and use the scaffolding without ever looking at how it is implemented.

I used the worked example effect throughout the whole tutorial. Whenever a new function is introduced, a textual explanation is accompanied by a small worked example of using the function. These range from small GHCi snippets such as the example in Figure 2.6 to a list of multiple function calls and their effects such as the examples shown in Figures 2.1 or 2.2.

2.4 Exercises

The tutorial includes seven exercises grouped under three questions based on the topics the exercises cover:

(1) Basics (3 exercises);

(2) Aggregation (2 exercises);

(3) Plotting (2 exercises).

Each exercise was designed to introduce as few concepts as possible. As such, the exercises build on concepts introduced in previous exercises, making the learning path as linear as possible to remove confusion. Section 2.4.1 includes a breakdown of the data science concepts, and Section 2.4.2 breaks down the design decisions behind each exercise in detail.

Each exercise starts by posing a question about the data. These questions motivate working with the dataset, and encourage the students to think of their own questions they could answer using the same tools. The body of each exercise then introduces any new functions that are needed to complete the exercise. As mentioned in Section 2.3.2, this is accompanied with a code example if the function is one the students might not have encountered in their course.

Throughout the tutorial, the students are encouraged to extract columns from the dataset and work on them with functions that they already know. This helps them get hands-on time with concepts that are taught later in the semester that students do not have much practise time with—namely I/O (which is taught within the last two weeks of the semester) and composing several functions to transform columns into the desired result (which is taught within the last month). There are multiple reasons behind this choice of focus for the tutorial.

The first is the aforementioned study by Hughes, which found that students perceived concepts taught later in the course as harder since they did not get familiar with them. The second was an analysis conducted by Tirronen et al. that analysed beginner-level coding sessions for common mistakes.[34] They found that some of the main areas students struggled with were functional composition (`f (g x)` is equivalent to `(f . g) x`) and application (`f x (g y)` is equivalent to `f x $ g y`) alongside pattern matching and partial application. Unlike features specific to functional languages like pattern matching, functional composition and application are concepts that are present in some form in most data science-oriented languages, since working with the dataset can be thought of as transforming and reducing a large data structure (the map-reduce paradigm). Therefore, if we want to teach students the concepts behind computation rather than a specific syntactic implementation (as Fisler claims we should for introductory computing courses),[1] functional composition and application are core paradigms we should aim for the students to master.

Alongside implementing the code, some exercises also include framed sections. These include tips on how to break down a data science problem, or a question about whether a relation that the students found is causal or simply a correlation. Some even encourage students to think of potential real-world events that might have influenced the data but are not included in the dataset. I designed these sections to encourage students to think about the context within which they are writing code.

I also took inspiration from existing tutorials. The formatting of the tutorial closely resembles the style that INF1A tutorials use to make the experience as cohesive as possible for the students. Similarly to INF1A tutorials, I also point out functions that might be helpful to the students. In this way, the exercise points them in the right direction without giving them the solutions outright, which I believe strikes a good balance between hand-holding and freedom. The introduction of functions to load and view the dataset follows the same general structure that the FDS course uses in their Python tutorials, since this structure has been tested over multiple years by hundreds of students.

2.4.1 Data science concepts

Discounting procedures to transform the dataset (selecting columns or aggregating data), the tutorial introduces three tools used in data science. This was done on purpose, since the focus of the tutorial is on using Haskell for data science, rather than doing data science in Haskell. Therefore, instead of introducing many data science concepts (which are taught in the second year FDS course anyway), I encourage the students to think about what their results mean in a larger context. I do this through a series of boxed questions mentioned previously.

(1) **Descriptive statistics (range, mean, median).**

The first question introduces descriptive statistics of a list of floating point numbers. These include measures of central tendency (mean and median), as well as measures of variability (minimum and maximum). The tutorial includes a helper function `stats` that computes the minimum, mean, median, and maximum of a list of floating point numbers. This function is introduced in the very first exercise, alongside an aside on the mean and median—their definition, uses, and limitations are all explained. The implementation for `stats` is located in the `Helpers.hs` file, since it operates on lists, not on data frames.

The function explicitly implements mean and median. I did not manage to find any library that computes the median of a list—the `statistics` library can calculate the median of a vector,[27] but requires the parameters α and β. These are non-trivial, since even the documentation refers to a 1996 paper by Hyndman for their definition.[35] Explaining these parameters would be out of scope of this tutorial, and it would also clash with the α and β coefficients from regression defined later, so I deemed a manual implementation necessary. Note that the code also has the same asymptotic runtime as the `statistics` function, since the median implementation is limited by the $O(n \log n)$ runtime of the sorting algorithm. The `mean` function is also implemented manually for transparency.

(2) **Fitting data with a curve.**

The second question introduces linear regression as a way to create a linear fit. This adds a concept from machine learning to the students' data science toolkit. The tutorial purposefully omits as much mathematical notation as possible, since we want to focus on the effects rather than notation. This approach is recommended by Hicks and Irizarry in their paper on teaching data science.[36]

The regression is also located in the `Helpers.hs` file, since it deals with a list of floating point numbers. Its implementation converts the two lists of numbers into `Vector` types, which are then passed to a linear regression function implemented by the `statistics-linreg` package.[37]

As the tutorial explains, the regression function returns two coefficients α and β, where the best fit line can be constructed as $y = \alpha + \beta x$. The function also returns the R^2 coefficient of determination, which quantifies how good the fit is with a value between 0 (no correlation) and 1 (data perfectly described by fit).

(3) **Plotting.**

```haskell
scatterPlot :: (PlotValue x, PlotValue y) => FileName -> [(x, y)] ->
            -> AxisTitle -> AxisTitle -> Title -> IO ()
scatterPlot fileName data xtitle ytitle title =
    toFile fileOptions ("plots/" ++ fileName ++ ".svg") $ do
        layout_all_font_styles      .= fontStyle
        layout_title_style          .= titleFontStyle
        layout_title                .= title
        layout_x_axis . laxis_title .= xtitle
        layout_y_axis . laxis_title .= ytitle
        -- create a line plot with custom styling via liftEC, which
        -- nests computation of the points within the plot computation
        plot $ liftEC $ do
            plot_points_style  .= (filledCircles 5 $ opaque black)
            plot_points_values .= data
```

Figure 2.7: **Implementation of the `scatterPlot` scaffolding function.** This code can be found in the `Plotter.hs` file. Note that `fileOptions`, `fontStyle`, and `titleFontStyle` are defined earlier in the file. Most comments (marked with `--`) are removed to fit the function on page.

Exercises in the last question deal with basic visualisation of data. This is purposefully introduced all the way in question 3. The students are encouraged to go back to their previous exercises and see if their answers to the questions change now that they can see the data visualised. In this way, the tutorial forces the students to fully consider the pitfalls of using summarising statistics. The tutorial includes two wrapper functions called `scatterPlot` and `linePlot2` (which plots two lines in one plot) that act as wrappers for the `Chart` library. I once again did this with a specific learning objective in mind—the wrapper functions require the students to put proper labels on their data and axes, which implicitly teaches good graphing practices.

I decided not to include plotting practices as an explicit point, since the second-year FDS course spends a considerable amount of time going over good practices in plotting. Furthermore, there is a difference between using plotting for data exploration and data visualisation—data exploration focuses more on the shape of the data, whereas visualisation focuses on communicating data to a wider audience. Plot features like a legend or axis labels are only necessary for the latter, which is why tools such as `gnuplot` that focus on interactive visualisations for a live audience do not include features such as axis labels by default.[38]

However, unlike functions within the `Helpers.hs` file, the tutorial encourages students to explore and modify the plotting code, which combines `Chart`'s descriptive syntax with comments to fully explain how the plotting function works and even directs them to the `Chart` wiki page,[39] which includes more simple examples for plots. As an example, Figure 2.7 includes a full definition of `scatterPlot`.

I use plots to once again solidify Haskell as a language that has ‘real-world’ uses. INF1A students spend a whole semester working with Haskell within GHCi and primarily deal with textual input and output. Reinforcing the idea that Haskell can create images without much trouble once again feeds into enhancing individual interest for students—programs that have visual I/O are often perceived by students as being more capable of building fully-fledged applications and therefore more interesting.[14]

2.4.2 Detailed overview of tutorial exercises

As I mentioned in Section 2.1, the tutorial gives the students an overview of the dataset, as well as functions to load and view the dataset (both the full dataset and column subsets), and the `getColumn` function to extract a column as a list. This means that the students should be somewhat familiar with the dataset and the corresponding helper functions by the time they get to the exercises. This allows the focus of the exercises to be firmly on manipulating data extracted via the `getColumn` function.

(1) BASIC OPERATIONS AND STATISTICS.

(1.a) **How long is a mission?** $\rightarrow$ Getting and filtering a single column. Calculating basic statistics (minimum, mean, median, maximum).

This exercise introduces the students to the `stats` function, and asks them to calculate simple statistics about the length of a mission using data from the `missionDurationTotal` column.

Since there are rows in this dataset that have a duration of 0.0, the students should see that the minimum duration of a mission is zero. I use this to motivate filtering the dataset to obtain only non-zero duration missions and ask how many missions have zero duration. I tell the students to use the `filter` function, which takes a predicate function and a list, and removes list elements that do not satisfy the predicate.

This exercise was designed to get the students to think about the meaning behind data—what does it truly mean to have a zero-duration mission?—as well as give them practice with the `filter` function they learned in class.

(1.b) **What is the training time and age of an astronaut?** $\rightarrow$ Working with two columns. `Int` to `Double` conversion. Filtering by another column.

When doing data science, we rarely only use a single column from the dataset. As such, I designed the second exercise to calculate an estimate for the training time and age of the astronaut by subtracting the mission year from the selection and birth year respectively. The students are pointed to the `zip` function—which takes two lists and creates a list of two element tuples—and told to apply a lambda function on each tuple to calculate the ages.

Another common issue in data analysis is that data is often stored in incompatible types. Sometimes numbers are stored as text, sometimes as an integer, sometimes as a floating point number. This exercise introduces this concept by pointing out that the students will need to use the `fromIntegral` function to convert their numerical values from integers to a floating point format that `stats` can accept.

Lastly, this exercise introduces the concept of filtering one column by values of another column—we want to get the training time per astronaut for their first mission only. This is the first time the students need to use a function they might not have encountered in their course, so I point out in the tutorial text that the three list `zip3` alternative to the two list `zip` function exists. The students can then do the filtering via the `filter` function on an element of the tuple, which is an extension of exercise (1.a). This can either be done by a lambda function or the helper functions `fst3` and `tls3` included in the `Helpers.hs` file. These are a 3-tuple variant of the standard Haskell functions `fst` (extracts the first tuple element) and `tail` (removes the first tuple element and returns a two-element tuple).

(1.c) **What percentage of mission time is spent on EVAs?** $\rightarrow$ Selective filtering by a different column. Calculating percentages.

This exercise is a follow-up to the filtering done in (1.b). However, this time the focus is on missions instead of astronauts. Since there are multiple observations per mission (one for each astronaut), this exercises introduces the `nubBy` function with a code example. This function filters out duplicates based on a provided predicate function, and acts as a counterpart to `filter`. The intention was for students to test for equality of the `missionName` column in the predicate, thus filtering out duplicate entries for the same mission.

Instead of also introducing a new statistic to calculate that the students may not know, I decided to have the exercise ask for percentages instead. This is a topic that the students are familiar with and it is a common way to describe data, making it the perfect candidate to introduce some variety without another maths explanation.

(2) AGGREGATING DATA AND FITS.

(2.a) **What is the number of astronauts per country?** $\rightarrow$ Grouping data by a column.

Another extremely common operation in data science is aggregating data. In this exercise, the students are asked to get a list of tuples which contain the country and the number of astronauts who went to space from that country. Since each astronaut can go to space multiple times, the tutorial also warns the students that they need to only include the first mission per astronaut (filter by column `profileMissionsNumber`). This serves as a reminder to once again consider what data we are working with before we start, since details like this are easy to miss.

The actual grouping operation relies on students extracting a list of country strings from column `profileNationality` (after filtering out for astronaut duplicates). The intention was for students to then use the provided hint of using `(group . sort)` to group the country names into nested lists (this example can be seen in Figure 2.6). Each nested list will contain as many elements of the country name as there are astronauts, so the data can be transformed into the desired form by mapping a lambda function such as `\l -> (head l, length l)` over the list of lists.

(2.b) **What is the percentage of female astronauts before and after the Cold War?** $\rightarrow$ Combining grouping and (categorical) filtering. Linear regression.

This exercise takes a slightly different approach to aggregation. The intent of the question is to count the data based on whether an entry occurs before (inclusive) or after 1991. I designed the question to include both explicit categorical data (filtering by male and female) as well as creating our own categorical data (pre- and post-1991).

I also explicitly introduced a real-world event (the Cold War) that is not directly related to the space program but could have some influence on the data. The exercise includes a framed question section that asks whether the student thinks that there is a causal relation in this trend; as well as encourages them to think of other possible factors that could have influenced this trend. There is no right answer, and as such no answer is provided, but possible answers might include looking at the general societal trends on male-female equality or using the same code to probe whether the astronauts were associated with the military or not via the `profileMilitary` column (since the U.S. space program used to require military training which was only accessible to men during the 1960s).

This exercise also introduces the `linearRegression` function. I made the decision to include it here since this exercise was otherwise a bit shorter than the others, and I wanted to have each exercise contain an approximately equal amount of concepts. The regression also acts as a counterpoint to the more qualitative discussion that the open question about causality opens. The students now have a tool to not only look at trends based on categorical separation but also trends based on linear regression. Not only are these more descriptive, the `linearRegression` also outputs the coefficient of determination R^2 (where R is Pearson's correlation coefficient), which allows the students to assess the quality of the fit.

(3) CREATING PLOTS.

This question introduces plotting. Since visualising data is a different paradigm from numerically summarising data, there is a longer introduction to ensure everything is explained in sufficient detail. The type signatures of both plotting functions that the students will use are introduced.

Figure 2.8 shows the type signature of `scatterPlot`. This type signature is very useful. First, it tells us that `x` and `y` are the values that are being plotted via the `PlotValue` type constraint (the tutorial text explicitly states that both integers and floating point numbers satisfy this type constraint). Second, it tells us that `x`

```
scatterPlot :: (PlotValue x, PlotValue y) => FileName -> [(x, y)]
            -> AxisTitle -> AxisTitle -> Title -> IO ()
```

Figure 2.8: **Type signature of the `scatterPlot` scaffolding function.** `PlotValue` is a type constraint that both `Int` and `Double` values satisfy. `FileName`, `AxisTitle`, and `Title` are both `String` type synonyms.

and `y` values are taken as a list of tuples, which implicitly ensures that there is an equal number of x and y values to form valid plot coordinates. Lastly, by using `String` type synonyms, it both reminds students that they exist and describes what each string is for.

(3.a) **What are the trends in the number of missions per year?** → Making a scatter plot.

I designed this exercise to have a similar solution structure to (2.a)—the students are now grouping by year instead of country (integers instead of textual data). This means that this exercise is taking advantage of the worked problem effect, since the students already have a high-level solution structure that they can reuse.[1] In this way, the tutorial also provides an opportunity for students to reuse their own design constructs, since this is how data analysis is often done in practice.

The reason why I made this exercise have similar structure to an earlier exercise is to allow the students to obtain a novel piece of data (motivated by a new question) but still allow enough time to properly introduce plotting. The tutorial includes a code example of how to call `scatterPlot` if the students already have a list of years and a list of corresponding counts to reduce any remaining confusion regarding how to use the plotting functions. Figure 2.9 shows the plot the students should produce.

The plots are produced in an SVG (scalable vector graphic) format using the `toFile` back-end function provided by the `Chart` library directly. Unfortunately, the library does not support making PNG files (I expand on this in Section 3.1). The advantage that SVG plots offer is that they can be opened in a browser, meaning that the file can be replaced even while it is being rendered, and refreshing the browser updates the plot in place.

In order to connect plotting to the different summarising statistics that the students have been calculating, I encourage the students to also use the `stats` and `linearRegression` functions on the data they obtained in this exercise, and to see whether they look like they would expect them to. The intention behind this question was to highlight that statistics and visualisations can provide two different (and mutually complimentary) ways to describe data.

[1]The worked effect comes from a 2006 study by Sweller that describes how worked examples can be an effective learning tool,[32] and I have already described how this effect is used to create code examples in Section 2.3.2.

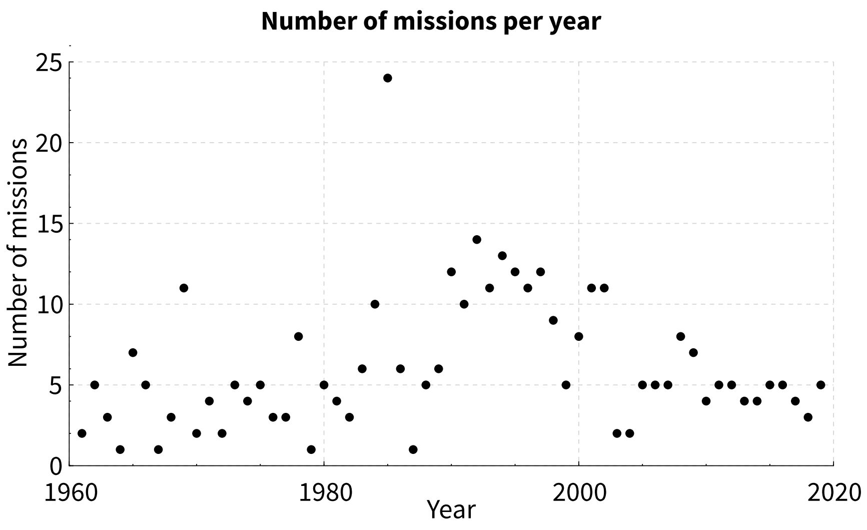

Figure 2.9: **Plot output of exercise (3.a).**

This exercise also features another open-ended question, encouraging students to think of possible factors that might have influenced this trend. I connect this question to exercise (2.b) by asking the students whether they think their answers to the two exercises are correlated. These questions were designed to make the students see the dataset as a set of correlated columns rather than a collection of several independent observations.

(3.b) **What is the cumulative time spent in space?** $\rightarrow$ Making a line plot with two lines. Cumulative sums.

Since the `linePlot2` function is very similar to the `scatterPlot` function (the difference being that the students must provide two `[(x, y)]` lists, as well as two labels to make the legend), this exercise instead introduces cumulative sums. I designed this exercise to combine all the the skills the students have learned throughout the tutorial.

The first step is to filter out by mission duplicates via `nubBy`, which the students learn in exercise (1.a). Then, we need to group both total and EVA time by year, which is similar to what students did in (3.a). The next step is reusing the `(group . sort)` construct from (2.a), but this time using the functions `groupBy` and `sortOn` that are introduced to the students. The last part is calculating the cumulative sum, which can be achieved with the library function `scanl`. I introduce this function as a counterpart to the left fold function `foldl` that the students learned in class.

The `foldl` function takes an identity element, a (2 to 1) accumulator function, and a list, and reduces the list into a single value by using the accumulator function on each element. In contrast, the `scanl` function creates a list with the accumulated value at each step.

All three functions (`groupBy`, `sortOn`, and `scanl`) are introduced by their type as well as an example on a list of strings. I used strings, since this both

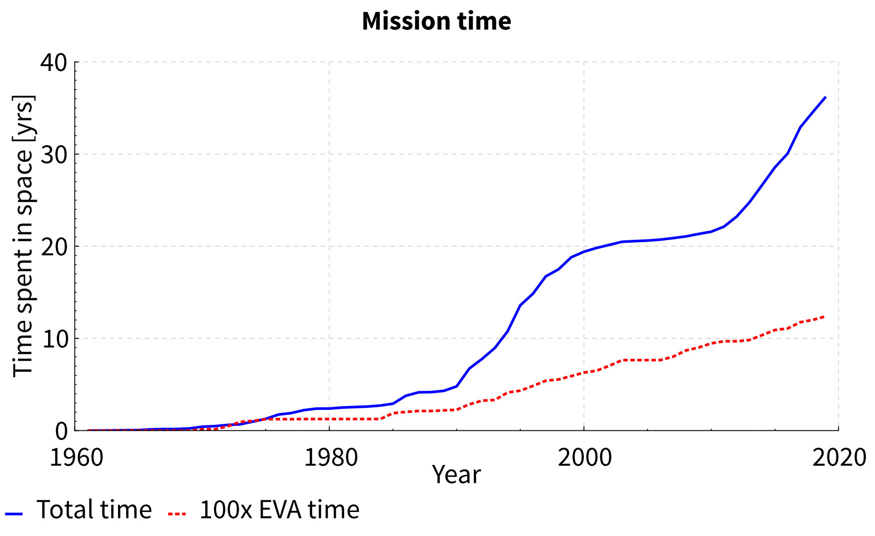

Figure 2.10: **Plot output of exercise (3.b).**

showcases the flexibility of the functions, and requires the students to adapt the solution pattern to numbers.

There is one more step that the students need to do with their cumulative sums. The time spent on extravehicular activities is minuscule compared to the total time in space, so the students need to scale the EVA time in order to see both lines on the plot. The final plot is shown in Figure 2.10. I believe this is a good example of trying to visualise two sets of data that are on a different order of magnitude, which is another problem that has multiple solutions in practice. Other ways to approach this would be to use two y-axes or a logarithmic scale, but these approaches significantly increase code complexity.

2.5 Solutions

The tutorial solutions are contained in the file `Solutions.hs`. I encourage the students to check these solutions after each exercise since each exercise builds on skills from previous ones. I also included a reminder that there are several ways to obtain the correct solution, and that the solutions provide only one possible way to get an answer.

The students have two ways to interact with the solutions: they can load the solutions file into GHCi directly and call the solutions function, or they can open the file and look at the implementation. However, since opening the `Solutions.hs` file would show all the solutions at once, I made functions to print each solution function individually.

These printed solutions include comments, `let` statements and `print` calls even though these can be omitted in GHCi. I did this on purpose, since these functions are meant to be useful for students writing exercises in GHCi or in an `IO ()` function. Figure 2.11 shows the output of the two function calls for exercise (1.a). Notice that the printer skips lines used to print verbose descriptions of the output in `exercise1aSolution`.

```
ghci> :load Solutions.hs
ghci> exercise1aSolution
statistics [min, mean, median, max] for mission durations (hrs)
[0.0, 1051.559637931035, 261.115, 10505.0]

statistics for mission durations (hrs), without zero-duration missions
[0.61, 1056.527636220473, 261.7625, 10505.0]

number of zero-duration missions
6
ghci> exercise1aPrint
Exercise (1.a) solution:
|  frame <- loadDataset
|  let ds = getColumn frame missionDurationTotal
|  (print . stats) ds
|  (print . stats . filter (/= 0)) ds
|  (print . length . filter (==0)) ds
```

Figure 2.11: **Output of calling the solution functions for exercise (1.a) in GHCi.**

```
exercise1aSolution' = do
    let s = astronautStream >-> P.map (view missionDurationTotal)
    dsFull     <- foldStream L.list s
    dsFiltered <- foldStream L.list $ s >-> P.filter (/=0)
    print $ stats dsFull
    print $ stats dsFiltered
    print $ (length dsFull) - (length dsFiltered)
```

Figure 2.12: **Streaming solution alternative to exercise (1.a).**

2.5.1 Streams and alternative solutions

The heavy focus on material that the students already covered in class means that most of the potential of the `Frames` library remains unexplored, since the students never have to interact with a frame directly. In order for the students to still get as many examples of how to use `Frames` outside of the tutorial, I also provide alternative solutions called `AltSolutions.hs`. This file provides solutions to the same exercises without loading the full dataset to memory. This is achieved through streaming and folding the dataset using the `pipes` and `foldl` packages. Figure 2.12 shows alternative solutions for exercise (1.a). Note that these solution files do not include verbose descriptions, since they are meant to be viewed after finishing the tutorial. This is also why the `AltSolutions.hs` do not include printer functions. However, the alternate solutions functions have a prime (`'`) at the end of the function name to prevent name clashes, which allows both `Solutions.hs` and `AltSolutions.hs` to be loaded into GHCi at once.

```
foldStream fold stream = Frames.runSafeEffect fold′ stream
     where fold′ = (Control.Foldl.purely Pipes.Prelude.fold) fold
```

Figure 2.13: **Implementation of the `foldStream` helper function.**

The alternate solutions never use the `loadDataset` I/O operation. Instead, we use the `astronautStream` directly, and then use the pipe operator `Pipes.(>->)` to sequence multiple operations on a single stream in a style similar to UNIX piping. For example, in exercise (1.a) we can map the `view missionDurationTotal` lens onto each record in the stream, reducing the records from 20 fields to 1. The stream is then consumed by a fold `Control.Foldl.list` from the `foldl` library, which converts it to a list. This means that instead of loading the whole dataset, we only need to load one column, which yields less than 1/20th of the memory footprint.

Despite these solutions looking simple, there is quite a bit of code required for the streaming solutions to be this concise. Most of the heavy-lifting is done by the `foldStream` function (shown in Figure 2.13), which combines functions from three libraries. The `runSafeEffect` function is a wrapper for the `pipes` library function `runEffect`, implemented by `Frames` in order to execute a piping operation with frame rows. The implementation of `foldl′` transforms a regular fold into a consumer for the output of a pipe, and is detailed in the `foldl` documentation.[28] This interoperability is probably due to the fact that `foldl` and `pipes` are made by the same author, so this was a design consideration during development of the two libraries.

2.5.2 Motivating data science in Haskell

The alternate solutions work together with the last section of the tutorial, which expands on the idea of lazy computation and streaming as a benefit of doing data science in Haskell. It shows students how the `Frames` library works lazily by comparing a program which prints the first three names from the `profileName` column to a program that prints all the names. The first example uses GHCi with the `+s` flag enabled, which makes GHCi print the run time and an estimate of the total memory used. However, we can do even better—the second example uses a compiled program with runtime system (RTS) profiling enabled. The tutorial includes an example of how to compile a program with profiling enabled, so that students can run their own tests if they want to.

Using RTS flag `-s` allows us to confirm that the maximum memory residency for a program that prints three names is 230kb while a program that prints all names occupies up to 1060kb. This is about a four-fold increase for 400 times as many names! This small example is then used to motivate that Haskell might be a good tool to do data analysis that benefits from lazy evaluation, or for datasets that are too large to fit into memory.

2.6 Changes from draft version

The students were sent a draft version of the tutorial in December in order to gather feedback. This section lists the main differences between the draft and final version of the tutorial, as well as why the changes were made. Then, in Section 2.7, I go over the survey the students filled out.

- All of the tutorial code (all scaffolding functions as well as exercise function stubs) were included in one file called `Tutorial.hs`. There was a second file called `Solutions.hs`, which was a copy of the tutorial file with the solutions written out in place of the function stubs. This made it impossible to load both files into GHCi simultaneously, since each function had two definitions.

  The new version of the tutorial properly splits the tutorial into several modules. This means the new `Solutions.hs` file can simply import the code from `Tutorial.hs`. While I was doing this restructuring, I also decided to split the helper and plotting functions into their own modules to clearly divide functions that work with the dataset, functions that operate on lists, and plotting functions.

- The old plotting functions did not include axis labels. This was because I viewed the plots as more of a part of the data exploration rather than visualisation of results. However, the lack of axis labels was one of the main points of feedback I received when I presented the state of my project in January, so I added them to the new version.

- The old tutorial did not provide alternative solutions. I decided later on that these might be beneficial to provide students with code examples of how streams can be used to solve the same exercises. This also required adding `foldl` to library dependencies.

- The old tutorial did not include linear regression. While I originally wanted to focus the tutorial more on purely manipulating the dataset, I decided to introduce the `linearRegression` function to fill the gap in data science content in question 2 exercises. This also meant that I had to modify the package installation to include the `vector` and `statistics-linreg` libraries.

- Since there were now more libraries required than fit on a single line, I decided it would be beneficial to explicitly state which library is required for each module imported with a comment. This made it much easier to understand why each library was necessary. It also made the tutorial more robust, since it is easier to troubleshoot any library conflicts that could arise if the tutorial was adapted in the future.

- The old version of the tutorial also did not include the `printDataset` function or explanations for how to make multi-column subsets via lenses due to time constraints—I wanted to include pretty printing in the first draft, but could not manage to do so in time, since I wanted to release the tutorial before the winter holidays so that more students could fill the feedback form.

  These features were low on the priority list since they were not necessary to complete the tutorial. The students did not need to explore the dataset since the tutorial text included a dataset descriptor table and I ensured that the dataset was properly cleaned. This is an idealised scenario that is often not true with real datasets. Therefore, I added support for properly visualising the dataset within the console for the final version of the tutorial:

  - The new tutorial version introduces the students to `viewDataset` and `printDataset`, and no longer asks the students to use the library functions `printFrame` and `takeRows`.
  - The new tutorial text also includes a section on how to lens the dataset to get multi-column subsets with a code example as well as an in-text explanation.

- The last difference is that despite exercise (3.b) requiring the `groupBy` function, this was not mentioned in the question. Since the feedback forms suggested that most students did not successfully complete all 7 exercises, I added this function. This also makes the tutorial fully consistent, since all functions needed should be introduced properly within the tutorial text.

2.7 Student feedback on the tutorial

The draft tutorial text included a link to a survey with 8 questions (5 numeric ratings (higher is better), two optional text answers, and one choice question) about the tutorial. Seven students have filled out the survey. This is not enough for the results to hold statistical significance, since the students most likely to fill the survey are ones that enjoyed the tutorial enough to engage with it. However, the answers all show a relatively small standard deviation σ, which means that the students were in agreement in their rating. Figure 2.14 shows the results of the numeric questions.

Five students answered 'just right' when asked about tutorial length, with one reply each for 'too short' and 'too long'. The optional written responses to the two long form questions are included in the text below. Note that there is no way to identify whether the same student answered both questions since all responses are anonymised.

IF YOU DID SOME UNGUIDED ANALYSIS, WHAT DID YOU DO AND WHY WAS IT INTERESTING TO YOU?

- "*I tried to look at intercorrelations between data features, and found some nice results!*"
- "*I calculated the percentage of astronauts that were female before and after 2000 and 1980 too to see if the end of cold war was what caused the increase or if it was a just a general trend independent of the cold war.*"
- "*One thing I did was to plot the yearly astronaut breakdown by country. I found myself using stacked charts, bar charts in this case. I found it interesting because you could get a lot of information that way from just one chart. It was interesting to correlate them with events in the real world such as the fall of the ussr and the challenger disaster.*"

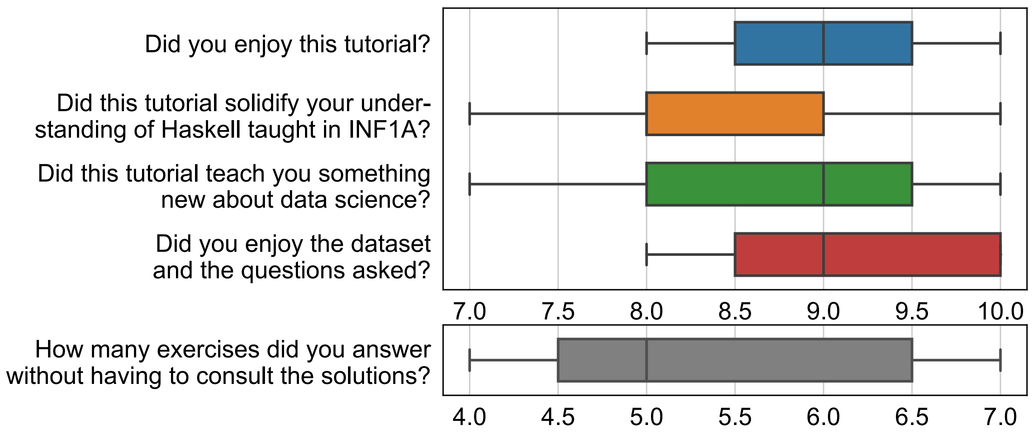

Figure 2.14: **Student responses to the draft tutorial survey.** Numeric questions only. Allowed value range is 1–10 for top plot and 1–7 for bottom plot.

DO YOU HAVE ANY FEEDBACK ABOUT THE DATASET? IF YOU FOUND A QUESTION BORING OR INTERESTING, THIS IS THE PLACE TO WRITE IT.

- "*The pacing of the questions was pretty good, appreciated the asides and helpful tips.*"
- "*I would have liked a more "dirty" dataset so we can practice data cleaning in haskell.*"
- "*It was interesting to see how some countries I didn't know had a space program had sent some astronauts to space.*"
- "*I liked the plotting ones.*"

The reply about bar plots shows that the student went to the `Chart` wiki to look for new plot types, since a bar plot was not provided for the students. The reply which talks about moving the categorical split from 1991 to 1980 and 2000 is also very welcome, since it shows that the students used critical thinking while going through the questions.

Overall, it appears that the tutorial was successful in its aims—it taught the students some foundational data science; reinforced the material they learned in INF1A; and even encouraged students to do unguided analysis and plotting. It is very encouraging to see that even though some students only completed four to five exercises without the solutions, the overall enjoyment of the tutorial remained high, meaning that the solutions were effective as well.

Chapter 3

Haskell for Data Science

In this chapter, I talk about the viability of the dataHaskell ecosystem. In Section 3.1, I describe some of the issues that I encountered while working with dataHaskell libraries. Then, in Section 3.2, I test dataHaskell using the Sanzu language benchmark.[16] Introduced by Watson et al. in 2017, the Sanzu benchmark was designed to test the capabilities of popular data science languages and compare their performance. The benchmark tests the language's capability on both synthetic and real-world datasets in six categories: basic file I/O; data wrangling; descriptive statistics; distribution and inferential statistics; time series; and machine learning. These categories were chosen to mimic the types of operations with datasets in practice. Lastly, in Section 3.3, I use memory profiling alongside scaling benchmarks to compare the performance of streamed and in-memory frames.

3.1 Issues with the dataHaskell ecosystem

The main disadvantage of using Haskell over a more popular language such as Python is that the ecosystem is very small. Even though the dataHaskell website does include a relatively expansive set of libraries, they are often not under active development and oftentimes lack basic features or documentation. I assume that these issues are mostly due to the fact that many libraries are maintained by a single person or a small group.

Below, I outline a few examples of issues that I encountered while I was working with the dataHaskell libraries. Keep in mind that libraries such as `Frames` and `Chart` are already some of the best documented libraries in the dataHaskell ecosystem, and they include materials such as the `Frames` tutorial[31] or `Chart` wiki.[39]

3.1.1 Frames

- The library lacks basic viewing functionality. The implementation of `showFrame` only takes in a fixed-width string separator, which is not sufficient to properly display text with varying widths (see Figure 2.2 for examples).
- There is no support for sorting the data frame based on a column, or reordering

```
sortFrame :: Ord l => Frame r -> Getting l r l -> Frame r
sortFrame frame lens = boxedFrame $ frameRow frame <$> idx
    where idx = map fst $ sortOn snd
                        $ zip ([0..frameLength frame - 1])
                              (F.toList $ view lens <$> frame)
```

Figure 3.1: **Implementation of the `sortFrame` helper function.**

the data frame based on a provided index. Figure 3.1 includes my implementation of this functionality.

To obtain a sorted frame, we have to extract the column in question and create a list of (index, element) tuples; then sort the list based on the element; extract the list of indices; unpack the frame; and repack it into a different object (a boxed vector frame instead of a frame of records). Since we can index frame rows in $O(1)$, this function is still asymptotically $O(n \log n)$ due to the sorting. However, even linear factors can make a big difference when the dataset size goes into gigabytes. Moreover, this approach is inelegant since we are operating on a list of the column values, rather than directly sorting the frame.

- Another basic feature that the library lacks is the group-by operation—taking one column to be a categorical variable and aggregating all other columns based on a provided function. I believe this is a crucial operation that should not be missing. There is a dataHaskell library called `Frames-map-reduce` which should implement this functionality,[40] but it is poorly documented and the installation fails due to issues with dependencies (the library and one of its dependencies require mutually incompatible versions of `base` Haskell).
- The default way to subset the dataset or to access fields is to use the `view` function, which is a part of the `microlens` package. The `Frames` library does include a wrapper function for lensing records called `rcast` (record cast), but makes getters that require another package for single-column lenses. This sort of design inconsistency means that to work with `Frames`, we need to also explicitly install and import another library.
- The documentation on how to go from a frame to a different data structure such as a list is heavily lacking. The `Frames` library tutorial does shows how to convert a single column to a list, but this is basic functionality that should be explained in documentation.

3.1.2 Chart

- The main feature missing from this library is support for heatmaps, which I would argue constitute one of the basic plot types. The Chart GitHub page does include issue tickets requesting the functionality, as well as a pull request that would add basic heatmaps. However, despite being proposed in 2014, with some updates to the pull thread in 2015, the library still does not have heatmap functionality ten years later. This is just one example of a feature that could be supported, if there was a larger team maintaining the package.

- While the `Chart` wiki does have over a dozen code examples, none of them actually work. As I mentioned in Section 2.4.2, the `Chart` library currently only supports vector graphic types. However, all the plot functions in the wiki and the documentation have a PNG file name when calling the `toFile` function. Calling the function this way produces a PNG file that cannot be opened. However, when I manually changed the file extension from PNG to SVG, the file could be opened without an issue. This looks as if this functionality was supported at some point in the past, but has since been removed.

  I was not able to find which version of the library stopped supporting raster rendering, or why it was removed. I also spent a considerable amount of time trying to figure out where in the process I could use the `diagrams` backend to render the file into a raster format directly, but I was unsuccessful.

- Another major issue I ran into while making the plots for the tutorial was a lack of documentation on how to customise the plot. The library includes a `Chart.Easy` module which significantly simplifies the syntax, but there is almost no documentation on how to modify the styling when using this interface. For example, there is no clear way to add axis labels to a plot this way.

  I eventually managed to find how to modify the styling in the following places: I found how to add axis labels by going through library tests on GitHub (this is separate from the wiki code examples); I found a way to change the font size in an issue thread in the `Chart` GitHub; and I found a way to change the styling of the plot markers while on StackOverflow. I was unable to find how to increase the spacing between the axis label and the tick labels.

There are other `diagrams`-based plotting libraries like `plots`[41] which do include features such as heatmaps. However, they lack other core functionality—in the case of `plots`, it does not support error bars. This means that in order to have basic plotting functionality with one syntax, the best solution is not to use Haskell at all. There are different interfaces to plotting programs that do not require Haskell, but this means leaving the single-language environment. For example, there are libraries that interface Haskell with Python's `matplotlib` but require Python to be installed;[42] or one can use Haskell to write to a file and utilise programs such as `gnuplot`.[38]

3.1.3 Statistics

The only function that I needed to use out of the `statistics` library is the linear regression function `olsRegress` found in the `Statistics.Regression` module. The documentation says that function has the following type:

```
olsRegress :: [Vector] -> Vector -> (Vector, Double)
```

It takes a list of predictor vectors (y values for each dimension) and a responder vector (x values) and returns a (line coefficients, R^2) tuple.

However, instead of using the `vector` package, they use the `dense-linear-algebra` package's internal `Vector` type defined in the `Statistics.Matrix.Types` module, which itself relies on `vector`. There is no documentation on how to create this type of `Vector`, nor is there a code example of how to use the regression function. This appears to have been enough of an issue that someone else implemented the `statistics-linreg`[37] package which uses the `vector` library directly.

This example of type inconsistency highlights one of the main issues with the Haskell ecosystem—several features are implemented multiple times by multiple people, often with conflicting naming conventions and convoluted dependency structures. This means that compatibility between packages is far from the norm.

3.2 Benchmarks

3.2.1 Test overview

In order to quantify the performance and features of the dataHaskell ecosystem, I implemented and ran the Sanzu benchmark in both Haskell and Python. The Sanzu benchmark also includes two types of tests: the micro benchmark uses synthetic datasets, while the macro benchmark uses real-world datasets. However, since the macro benchmark relies on time series, and this support is not integrated into the dataHaskell ecosystem, I only ran tests from the micro benchmark.

The original Sanzu paper compares five languages: R, Anaconda (Python local computing), PostgreSQL, Dask (Python distributed cloud computing), and PySpark (Python interface for the Scala-based Apache Spark cloud computing platform).[16] I chose Python as the common benchmark language for three reasons:

- Python and R were the only two languages which support all benchmark tests.
- Python offered the best performance out of all five languages in all but one of the Sanzu micro benchmarks (synthetic datasets).
- Python is the most popular programming language,[43] which means that comparing Haskell to Python will be relevant to a large audience.

There is one important point to remember with these benchmarks: most of the data science computation in Python is not actually done in Python. One of the reasons why Python as a language is so successful is that it can act as an interface to glue together different languages. Most functions in libraries like `numpy` and `pandas` are wrappers for highly optimised functions in either C, C++, or FORTRAN.[44;45] There are even libraries like `numba`,[46] which can translate numeric calculations in Python into a just-in-time LLVM architecture.[47] Therefore, when interpreting benchmark results, it is important to keep in mind that the performance has less to do with the language tested and much more to do with the implementation of each specific library. This is why I chose the Sanzu benchmark over a more general-purpose language benchmark.

	Operation	**Description**	**Haskell** memory	**Haskell** streams	**Python**
		BASIC FILE I/O			
*	Read	Read from a CSV file.	`Frames`	`Frames`	`pandas`
*	Write	Write to a CSV file.	`Frames`	`pipes`	`pandas`
		DATA WRANGLING			
*	Sorting	Sort frame based on a column.	M	NF	`pandas`
*	Filtering	Filter frame based on a column.	`Frames`	`pipes`	`pandas`
	Merging	Append two frames (inner join).	`Frames`	NF	`pandas`
*	Group by	Aggregate frame based on a column with a given fold function.	M	NF	`pandas`
*	Duplicates	Remove rows with duplicate entries in one of the columns.	M	NF	`pandas`
		DESCRIPTIVE STATISTICS			
*	Central tendencies	Mean, mode, median of a column.	`foldl` (mean only)		`pandas`
*	Dispersion	Range and standard deviation.	`foldl`		`pandas`
	Rank	Rank rows based on column values.	NF		`pandas`
*	Outliers	Remove outlier rows based on column.	M		`pandas`
*	Scatter plot	Plot data from two columns.	`Chart`		`matplotlib`
		DISTRIBUTION AND INFERENTIAL STATISTICS			
	PDF	Gaussian probability density function.	`statistics` (LC)		`scipy`
	Skew	Skewness of data.	`statistics` (LC)		`pandas`
	Correlation	Correlation of two columns.	`statistics` (LC)		`pandas`
	Hypothesis testing	Shuffling method hypothesis testing.	`statistics` (LC)		M
		TIME SERIES			
	EWMA	Exponentially weighted moving average.	NF		`pandas`
	Autocorrelation	Autocorrelation based on time series.	NF		`pandas`
		MACHINE LEARNING			
	Regression	Linear regression in 2D.	`statistics` (LC)		`numpy`
*	Clustering	K-Means clustering into 2 groups.	`kmeans`		`scikit-learn`
	Classification	Naive Bayes classification.	`sibe` (LC)		`scikit-learn`

Table 3.1: **Sanzu synthetic benchmarks.** Language columns list the library which offers this functionality. Benchmarks performed marked by *. Legend: M (manual implementation); NF (no functionality); LC (library conflict—no benchmark possible).

The full overview of tests in Sanzu's micro benchmark can be seen in Table 3.1. I ran 11 tests out of the benchmark suite. There are three reasons why a test could be excluded:

- Tests could be excluded if the functionality was not supported by a library in the dataHaskell ecosystem. This is the case for time series tests as well as calculating the rank of rows. These tests are marked by **NF**.
- Tests could also be excluded if the functionality was supported, but I could not implement the tests properly due to library installation conflicts. This occurred for the `statistics` and `sibe` libraries. These tests are marked by **LC**.
- I also excluded the merge test, because Python's performance scaled much faster than all four other languages tested in the Sanzu benchmark. The aim of these benchmarks was to use Python as a baseline to compare Haskell to other languages evaluated using the Sanzu benchmark. Python is not representative of other languages in the merge test due to different asymptotic behaviour.

Name	Type	Details
`time`	String	Datetime in `DD/MM/YYYY HH:MM:SS` format.
`city`	String	Chosen randomly from a list of 10^3 city names.
`words`	String	Chosen randomly from a list of 10^5 words.
`rand1`	Integer	Uniform distribution; range 0–10^6.
`rand2`	Integer	Uniform distribution; range 0–2^{31}.
`nor`	Float	Normal distribution; full float range.
`exp`	Float	Exponential distribution; full float range.
`uni`	Float	Uniform distribution; full float range.

Table 3.2: **Columns in the synthetic datasets.**

3.2.2 Implementation and methodology

The Python benchmarks were adapted from Sanzu files (link can be found in the Sanzu paper footnotes).[16] I kept the dataset generation scripts untouched besides troubleshooting minor out-of-bounds indexing errors and fixing CSV writer double-spacing lines. Table 3.2 includes a description of each of the columns in the synthetic dataset. Datasets sizes range from 10^3 rows (corresponds to 91 kilobytes) to 10^7 rows (0.904 gigabytes)—five datasets at five orders of magnitude.

I modified each Python test to run each benchmark ten times instead of four, and I modified the test writers to write a new line entry for each test rather than having columns `run1` to `run4` for each test. This data structure was mentioned earlier as tidy data.[21]

I wrote the Haskell benchmarks to be as close to the Python implementations as possible. For Python tests that index the data frame `df` directly (such as the line `df["rand1"].mean()`), I included column extraction within the timing window. The full benchmark code can be found in Appendix D, and details of the benchmark machine are listed in Table 3.3. Tests were run on an otherwise idle machine.

OS	Windows 11 Pro
CPU	Intel i5-1240P
RAM	16 GB
`python`	3.11.1
`ghc`	9.4.7

Table 3.3: **Benchmark system.**

Timing was done using CPU time. The Python benchmarks are relatively simple—we can obtain the current time via `timeit.default_timer` function before and after the execution of interest. Benchmarks in Haskell were trickier, since I had to make sure that the program evaluated the full computation instead of lazily evaluating a part of it. Haskell benchmarks also included two benchmark functions, since some of the tests included I/O, which required different syntax to handle impure functions. I used the bang pattern language pragma (`XBangPatterns`)[48] to force computations when assigned to a variable, and methods from the base Haskell `Data.Time.Clock` module to obtain the CPU time.

It is important to note that while CPU clock values are given with picosecond (10^{-12} s) precision, any computation that runs under a millisecond (10^{-3} s) will be hard to benchmark due to the implementations of each language's timing methods. Python's time method has a granularity of 1/100th of a second,[49] and Haskell's precision is

```
benchmark computation = do
    t1 <- getCurrentTime
    let !r = computation
    t2 <- getCurrentTime
    (putStr . show . nominalDiffTimeToSeconds) $ diffUTCTime t2 t1
```

Figure 3.2: **Implementation of the `benchmark` function.** Forces the full evaluation of a computation and prints the time taken in seconds.

similar. Therefore, tests for small datasets show times around a microsecond, which are larger than actual execution time.

The implementation of `benchmark` can be seen in Figure 3.2. The I/O variant `benchmark'` replaces the let statement with a bind to a dummy variable (`_ <- ioComputation`). This requires no explicit forcing.

The results shown were obtained by computing the geometric mean of the data, with error bars given by the standard deviation σ. As Fleming and Wallace claim in their 1986 paper on summarising benchmarks,[50] the geometric mean ($\bar{t}_G = \sqrt[N]{\prod x_i}$) is superior to the arithmetic mean ($\bar{t}_A = \sum x_i/N$) since it includes variability. However, we can take advantage of the fact that $\bar{t}_G \leq \bar{t}_A$ to also include the arithmetic mean. This is useful for comparing Haskell to other languages that were tested by the Sanzu benchmark, since Watson et al. used arithmetic means for their results.[16]

3.2.3 Results

Figure 3.3 shows the geometric and arithmetic means of all 11 tests. As we can see, Haskell (both compiled via `ghc` and interpreted via the GHCi REPL) is slower than Python in all but one test. As we would expect, the compiled version performs better than the interpreted version since GHCi cannot perform any compiler optimisations. However, the groupby and sort tests show that these do not always result in lower performance. Note that since most benchmarks have $\bar{t}_G \cong \bar{t}_A$, I will be using arithmetic means later for ease of computation.

The one test where Haskell is faster than Python is the groupby test. This is expected, since I implemented this test manually, and therefore only aggregate the column of interest, whereas the Python implementation has to aggregate all columns into a `DataFrameGroupBy` object which we can then index to get the desired results.

It is important to note the major time disparity on the mean and dispersion tests. In Python, these are implemented by calling a function directly on a column of a data frame (such as `df["column"].mean()`), while in Haskell they are implemented by folding over a lensed dataset (such as `L.fold L.mean $ view column <$> df`). This also explains the outliers time disparity, since these rely on filtering by a $\pm 3\sigma$ range.

The duplicates test relies on base Haskell more than a library, and therefore benefits from compiler optimisations more than other tests. The test is implemented similarly to

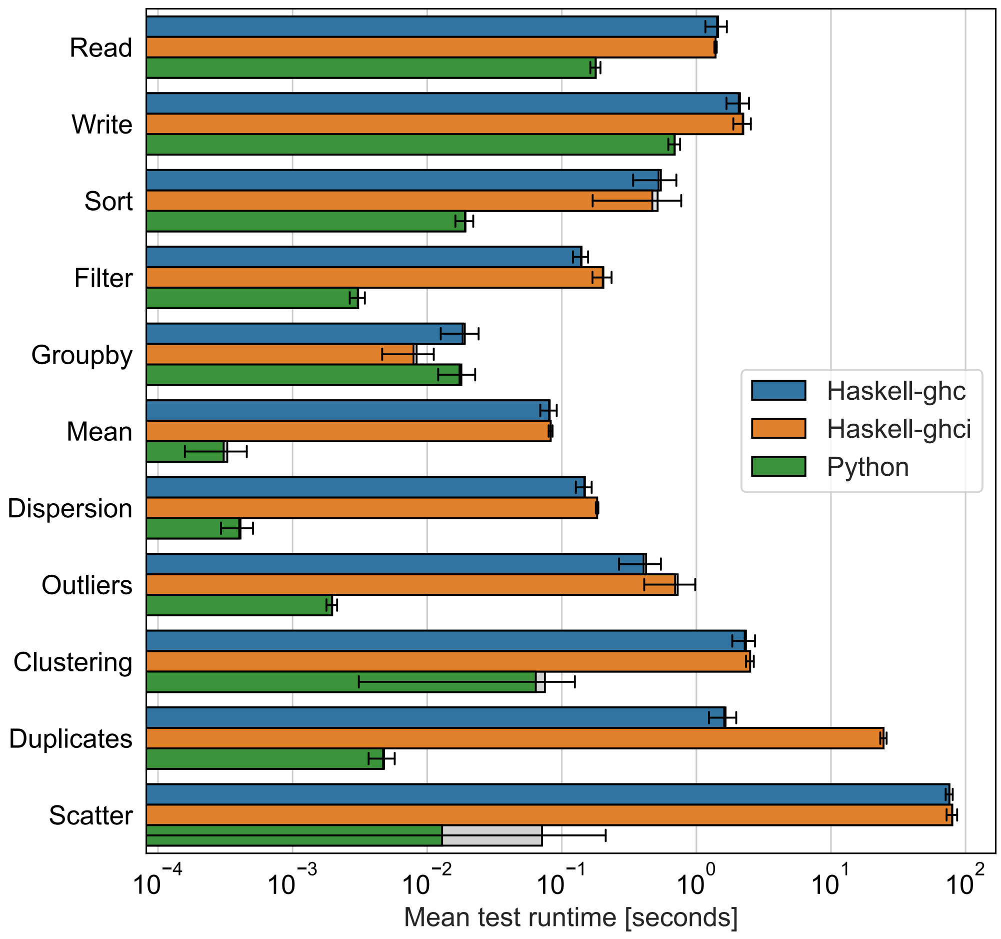

Figure 3.3: **Geometric mean of benchmark runtime.** Arithmetic means shown in grey (note that $\bar{t}_G \cong \bar{t}_A$ in most tests). Dataset with 10^5 rows. Errorbars given by σ.

the sort test—the column by which we are filtering is extracted, zipped with the index, and then filtered by element with `nubBy` to obtain the new index list.

The last test I will comment on here is the scatter test. I suspect the reason why Haskell performs way worse is because the output is a SVG file rather than a raster file, and `Chart` tries to plot all 10^5 points—the figure file size is 106.7 megabytes! I believe that Python's `matplotlib` performs some sort of optimisation that does not plot points that would not be seen. This is confirmed by Figure 3.4, where we see that the plot test scales slower with dataset size in Python compared to Haskell.

In general, we see Haskell and Python scale similarly in both I/O tests (read, write) as well as descriptive statistics (mean, dispersion), even though they are significantly slower (keep in mind that ties below 10^{-3} s are rounded to nearest clock Δt). This is expected behaviour, since both loading and statistics should have $O(n)$ runtime.

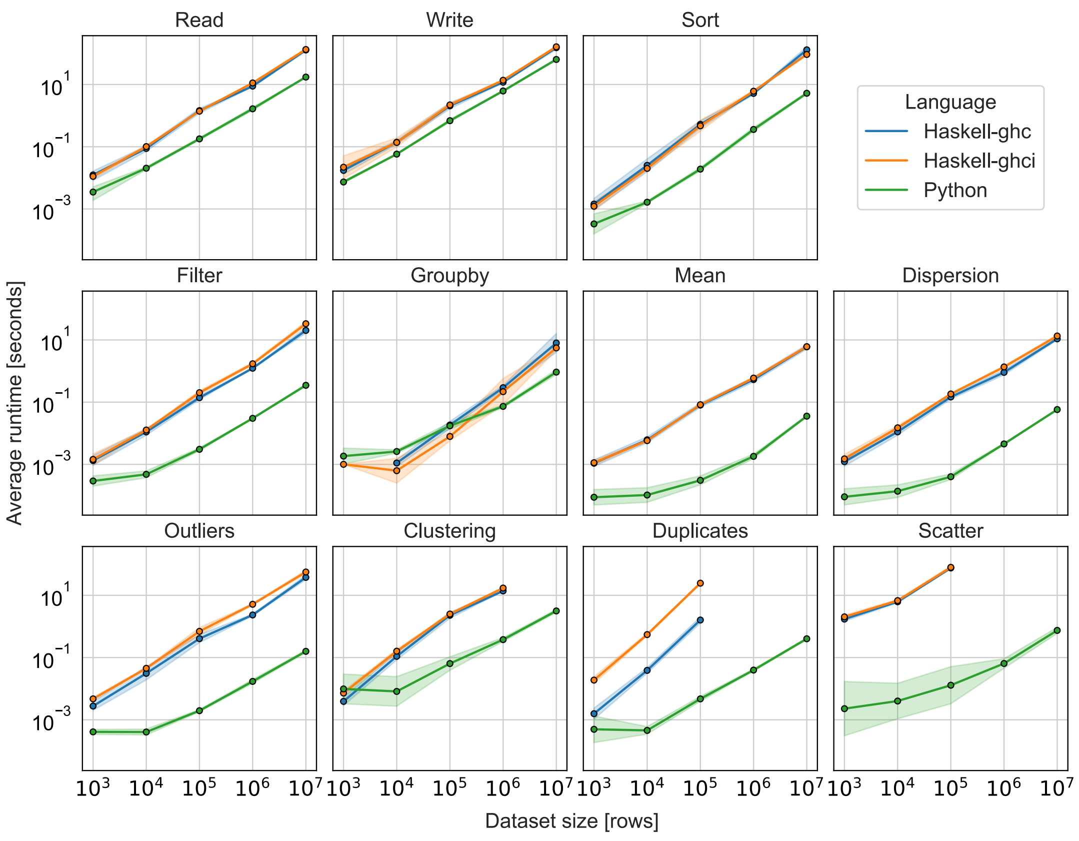

Figure 3.4: **Arithmetic mean of benchmark runtime based on dataset size.** Some benchmarks were only run on smaller datasets due to long benchmark times.

Groupby and duplicates tests also exhibit interesting behaviour—Haskell tests exhibit non-linear asymptotic time. This implies that the `groupBy` and `nubBy` functions used to implement these tests are not asymptotically linear (linear fit yields $\alpha_{\text{duplicates}} = 1.505$).

3.3 Stream benchmarks

Unlike a full in-memory data frame, we can only use streams for map-reduce computations because streams give us access to one element at a time. Since streams combine reading the dataset with computation, we need to redefine our benchmarks to include reading the dataset into the computation time in order to obtain values which can be compared. A second consideration that we need to address is how to benchmark making streams that use multiple folds for multiple computations at once.

To capture the option of performing multiple folds, I created two tests: a one fold test that computes the mean of a list and a two fold test that also computes the standard deviation. I also implemented equivalent in-memory tests that time reading the data frame, extracting a column, and performing the same fold operations as the stream tests.

```
foldStream L.mean $ stream >-> P.map (fromIntegral . view rand1)

foldStream ((,) <$> (L.premap (fromIntegral . fst) L.mean)
                <*> (L.premap snd L.std))
           $ stream >-> P.map ((,) <$> (view rand1) <*> (view uni))
```

Figure 3.5: **Implementation of folding a stream with one and two folds.** `L` refers to the `Control.Foldl` module.

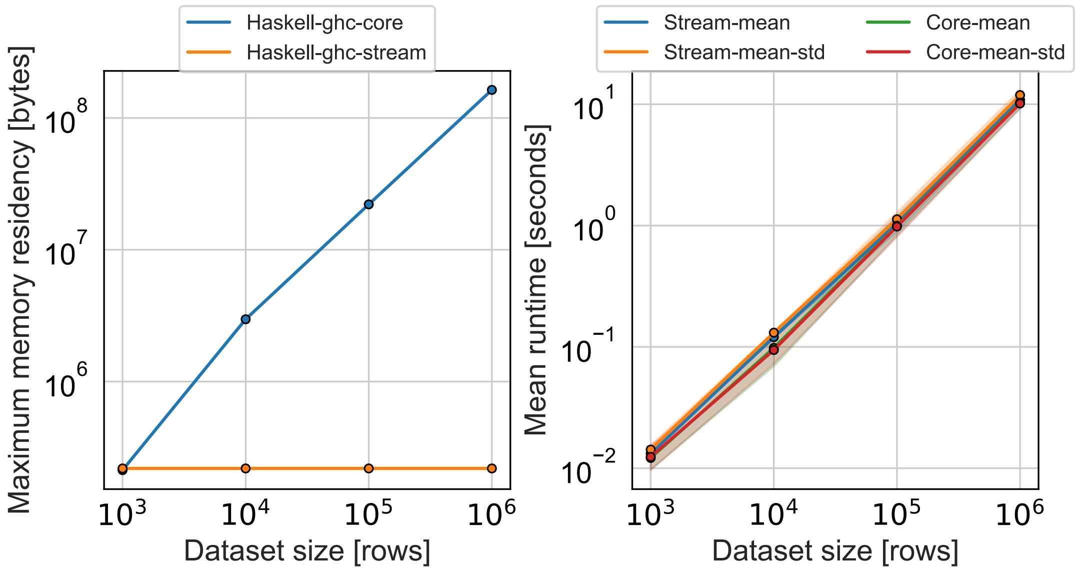

Figure 3.6: **Stream benchmarks. LEFT:** Maximum memory residency (10 runs). **RIGHT:** Geometric mean of runtime $\pm\sigma$ (10 runs).

Figure 3.5 shows the implementation of a one- and two-computation fold using the `Control.Foldl` module defined by the `foldl` library. I chose the mean and standard deviation as the two test computations, since these are implemented directly by the `foldl` library. Most tests from inferential statistics or machine learning categories (see Table 3.1) would require us to convert the stream to a list (via `Control.Foldl.list`) or vector (via `Control.Foldl.vector`) and then pass the output to another library. While this is possible—and would require a much smaller memory footprint than loading the dataset into memory—we would be testing the performance of computation that is not directly related to streams.

I plot the stream test results on datasets sized 90 kB (10^3 rows) to 90 MB (10^6 rows) in Figure 3.6. From the (geometric) mean runtime, we see that the difference between running one and two folds is negligible. Moreover, the performance of reading the dataset into memory and then performing a computation (tests marked `core`) is effectively equal to stream performance (tests marked `stream`). This means that streams are just as good as in-memory data frames if we only need to do one computation, since reading the dataset into memory is slower than performing the fold (see Figures 3.3 and 3.4).

Performing linear regression on the mean runtime data shows that the slope is almost perfectly linear for both the stream ($\alpha_{\text{stream}} = 0.976$, $R^2_{\text{stream}} = 0.994$) and in-memory ($\alpha_{\text{in-core}} = 0.970$, $R^2_{\text{in-core}} = 0.998$) benchmarks.

Results from memory profiling are perhaps even more important. When we look at maximum memory residency (maximum number of bytes occupied by the program), we see that the stream variant stays constant at 209 kB for all dataset sizes while the memory residency of in-memory data frames scales linearly with dataset size (the 90 MB dataset occupied at most 163 MB when loaded into memory). This confirms that we can use streams to analyse datasets larger than memory.

Furthermore, if Haskell tried to allocate more memory than the system could provide (for example if the dataset was too large), the operating system would start virtualising memory—writing currently unused parts of memory onto permanent storage, slowing down the computation. Therefore, while the computation speed is comparable for streams and in-memory data frames for smaller datasets, streams are preferable for large datasets.

We can quantify this slow down by looking at program productivity, which refers to the percentage of time that the program was doing useful calculation. Non-useful work mostly refers to time spent on garbage collection—lazy languages like Haskell are known to have less efficient heap management (or even memory leaks) that can cause low productivity.

When we look at productivity percentages in Figure 3.7, we can indeed see that the in-memory frames are a lot less productive, further slowing down computation compared to streams.

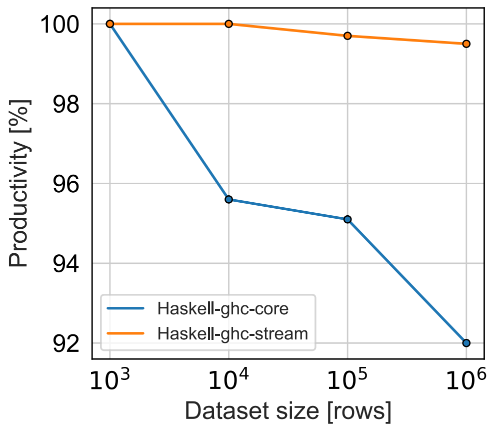

Figure 3.7: **Program productivity**

We can therefore conclude that streams are a compelling alternative for large datasets.

Chapter 4

Conclusions

4.1 Summary of work done

In this thesis, I have created a beginner-friendly framework for data-centric computation in Haskell. I used this code base to explore how Haskell can be used for manipulating datasets; both for educational and professional purposes.

In order to demonstrate Haskell's capabilities as an educational tool, I made a tutorial for first year Computer Science students. This tutorial was designed to help students solidify their understanding of functional programming through exercises that require the students to manipulate a dataset of information about astronauts. The code scaffolding and exercises were designed based on well-established techniques from literature on Computer Science education and cognitive theory. The deliverable included seven exercises that cover several concepts from data science and machine learning; beginner-friendly helper functions; and two solution sets. Student feedback shows that the tutorial is engaging and successfully solidifies students' understanding of functional programming.

I also demonstrated that Haskell is a good tool for computations with large datasets thanks to its built-in support for streaming I/O operations. I used the Sanzu benchmark to quantify the performance and feature availability of libraries within the dataHaskell ecosystem, which shows that Haskell is slower than Python in most computations. However, memory profiling tests show that while small datasets have similar performance for in-memory and streamed data frames, streaming allows Haskell to compute with large datasets without memory limitations. This makes Haskell a viable addition to the data science language toolkit.

4.2 Future work

We can build on top of the code base I have created to make more data-centric educational materials. Haskell is already used for introductory-level Computer Science courses, and with proper care it can also be used to create a course that uses data to motivate learning Computer Science. This would make the computing class more accessible for people that do not major in Computer Science, and teach them computing skills they can use in their own field.

In order to fully evaluate Haskell's capabilities, it would be desirable to implement the remaining Sanzu benchmarks and find other ways we can quantify how Haskell compares to other languages. Furthermore, we should use benchmark results to better explain the strengths of Haskell in data science computation, and use these results to motivate further development and optimisation.

In order for Haskell to be more accessible (and therefore widely usable), effort needs to be put into creating a more cohesive library ecosystem. This includes better compatibility for libraries, better documentation with replicable code examples, and creating content such as wiki pages and tutorials.

Bibliography

[1] Kathi Fisler. Data-Centricity: Rethinking Introductory Computing to Support Data Science. In *1st International Workshop on Data Systems Education*, pages 1–3, Philadelphia PA USA, June 2022. ACM. ISBN 978-1-4503-9350-8. doi: 10.1145/3531072.3535317.

[2] Kathi Fisler. Functional Programming the Glue for Introducing Computing. *YouTube*. URL `https://www.youtube.com/watch?v=2rXVDwKQ8S0`.

[3] Shriram Krishnamurthi and Kathi Fisler. Data-centricity: a challenge and opportunity for computing education. *Communications of the ACM*, 63(8):24–26, July 2020. ISSN 0001-0782, 1557-7317. doi: 10.1145/3408056.

[4] Kathi Fisler, Shriram Krishnamurthi, Benjamin S. Lerner, and Joe Gibbs Pilitz. A Data-Centric Introduction to Computation, 2023. URL `https://dcic-world.org/2023-02-21/index.html`.

[5] Pyret. URL `https://pyret.org/index.html`.

[6] PYPL PopularitY of Programming Language index. URL `https://pypl.github.io/PYPL.html`.

[7] dataHaskell : Current Environment. URL `https://www.datahaskell.org/docs//community/current-environment.html`.

[8] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, January 1993. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796800000599.

[9] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, San Diego California, June 2007. ACM. ISBN 978-1-59593-766-7.

[10] Timothy T. Yuen, Maritza Reyes, and Yuanlin Zhang. Introducing Computer Science to High School Students Through Logic Programming. *Theory and Practice of Logic Programming*, 19(2):204–228, March 2019. ISSN 1471-0684, 1475-3081. doi: 10.1017/S1471068418000431.

[11] Lawrence C. Paulson and Andrew W. Smith. Logic programming, functional programming, and inductive definitions. In Peter Schroeder-Heister, editor, *Ex-*

tensions of Logic Programming, volume 475, pages 283–309. Springer-Verlag, Berlin/Heidelberg, 1991. ISBN 978-3-540-53590-4. doi: 10.1007/BFb0038699. Series Title: Lecture Notes in Computer Science.

[12] Manuel M. T. Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, January 2004. ISSN 0956-7968, 1469-7653.

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, July 2004. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796804005076.

[14] John Hughes. Experiences from teaching functional programming at Chalmers. *ACM SIGPLAN Notices*, 43(11):77–80, November 2008. ISSN 0362-1340, 1558-1160. doi: 10.1145/1480828.1480845.

[15] Suzanne Hidi. Interest and its contribution as a mental resource for learning. *Review of Educational Research*, 60(4):549–571, 1990. doi: 10.3102/00346543060004549.

[16] Alex Watson, Deepigha Shree Vittal Babu, and Suprio Ray. Sanzu: A data science benchmark. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 263–272, December 2017. doi: 10.1109/BigData.2017.8257934. URL `https://ieeexplore.ieee.org/abstract/document/8257934`.

[17] Ulrich Schiefele. Interest, learning, and motivation. *Educational Psychologist*, 26 (3-4):299–323, 1991. doi: 10.1080/00461520.1991.9653136.

[18] Dan Hollis, Mark McCarthy, Michael Kendon, Tim Legg, and Ian Simpson. HadUK-Grid—A new UK dataset of gridded climate observations. *Geoscience Data Journal*, 6(2):151–159, 2019. ISSN 2049-6060. doi: 10.1002/gdj3.78.

[19] Austin Cory Bart, Ryan Whitcomb, Dennis Kafura, Clifford A. Shaffer, and Eli Tilevich. Computing with CORGIS: Diverse, Real-world Datasets for Introductory Computing. *ACM Inroads*, 8(2):66–72, May 2017. ISSN 2153-2184, 2153-2192. doi: 10.1145/3095781.3017708.

[20] Thomas Mock. Astronauts CSV Dataset, GitHub repository. *TidyTuesday GitHub*. URL `https://github.com/rfordatascience/tidytuesday/blob/2e9bd5a67e09b14d01f616b00f7f7e0931515d24/data/2020/2020-07-14/astronauts.csv`. Accessed: 2023-11.

[21] Hadley Wickham. Tidy Data. *Journal of Statistical Software*, 59:1–23, September 2014. ISSN 1548-7660. doi: 10.18637/jss.v059.i10.

[22] Brian O'Sullivan, Tom Harper, and Andrew Lelechenko. An efficient packed Unicode text type. *Hackage*. URL `https://hackage.haskell.org/package/text-2.1.1`.

[23] Johan Tibell. Memory footprints of some common data types. *Haskell and other*

things that interest me, 2011. URL `https://blog.johantibell.com/2011/06/memory-footprints-of-some-common-data.html`.

[24] Johan Tibell. A CSV parsing and encoding library. *Hackage*. URL `https://hackage.haskell.org/package/cassava`.

[25] Anthony Cowley. Data frames for working with tabular data files. *Hackage*, . URL `https://hackage.haskell.org/package/Frames`.

[26] Tim Docker. A library for generating 2D Charts and Plots. *Hackage*, . URL `https://hackage.haskell.org/package/Chart`.

[27] Brian O'Sullivan. A library of statistical types, data, and functions. *Hackage*. URL `https://hackage.haskell.org/package/statistics`.

[28] Gabriella Gonzalez. Composable, streaming, and efficient left folds. *Hackage*, . URL `https://hackage.haskell.org/package/foldl`.

[29] Gabriella Gonzalez. Compositional pipelines. *Hackage*, . URL `https://hackage.haskell.org/package/pipes`.

[30] Edward Kmett and Artyom Kazak. A tiny lens library with no dependencies. *Hackage*. URL `https://hackage.haskell.org/package/microlens`.

[31] Anthony Cowley. Frames tutorial. *GitHub*, . URL `https://acowley.github.io/Frames/`.

[32] John Sweller. The worked example effect and human cognition. *Learning and Instruction*, 16(2):165–169, April 2006. ISSN 0959-4752. doi: 10.1016/j.learninstruc.2006.02.005.

[33] Ville Tirronen and Ville Isomöttönen. Teaching types with a cognitively effective worked example format. *Journal of Functional Programming*, 25:e23, 2015. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796814000021.

[34] Ville Tirronen, Samuel Uusi-Mäkelä, and Ville Isomöttönen. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming*, 25:e11, 2015. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796815000179.

[35] Rob J. Hyndman. Computing and graphing highest density regions. *The American Statistician*, 50(2):120–126, 1996. ISSN 0003-1305. doi: 10.2307/2684423.

[36] Stephanie C. Hicks and Rafael A. Irizarry. A guide to teaching data science. *The American Statistician*, 72(4):382–391, 2018. doi: 10.1080/00031305.2017.1356747. PMID: 31105314.

[37] Alp Mestanogullari and Uri Barenholz. Linear regression between two samples, based on the 'statistics' package. *Hackage*. URL `https://hackage.haskell.org/package/statistics-linreg`.

[38] Demos for gnuplot version 6.0. *gnuplot*. URL `https://gnuplot.sourceforge.net/demo/`.

[39] Tim Docker. Haskell Chart: wiki. *GitHub*, . URL https://github.com/timbod7/haskell-chart/wiki.

[40] Adam Conner-Sax. Frames wrapper for map-reduce-folds and some extra folds helpers. *Hackage*. URL https://hackage.haskell.org/package/Frames-map-reduce.

[41] Christopher Chalmers. Diagrams based plotting library. *Hackage*. URL https://hackage.haskell.org/package/plots.

[42] Andrei Barbu. Bindings to Matplotlib; a Python plotting library. *Hackage*. URL https://hackage.haskell.org/package/matplotlib.

[43] TIOBE Index. URL https://www.tiobe.com/tiobe-index/.

[44] Building from source. *NumPy*, . URL https://numpy.org/doc/stable/user/building.html.

[45] Package overview. *pandas*. URL https://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html.

[46] Numba: an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code. *PyData*, . URL https://numba.pydata.org/.

[47] The LLVM Compiler Infrastructure. *LLVM*. URL https://llvm.org/.

[48] The GHC Team. Bang patterns. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.8.4*. URL https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/bang-patterns.html.

[49] timeit—Measure execution time of small code snippets. *Stackless Python documentation*. URL https://stackless.readthedocs.io/en/2.7-slp/library/timeit.html.

[50] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3): 218–221, March 1986. ISSN 0001-0782. doi: 10.1145/5666.5673.

Appendix A

Student feedback collection

A draft of the tutorial was sent to the INF1A students through Professor Sannella (the current INF1A course organiser and main lecturer). The students received a `zip` file that included the dataset, all the code, as well as the tutorial PDF. The tutorial text included a link to a survey at the front page. The survey was done via Microsoft Forms, with access granted to university accounts. A dataset containing all the responses can be found in the project files.

The first page of the survey had a full copy of the information sheet (see Section A.1 for key information) and then two yes/no consent questions (see Section A.2). If the students provided their consent, they were able to advance to a second page where all the questions were asked. The results of the survey are discussed in Section 2.7.

A.1 Participants' information sheet

This study was certified according to the Informatics Research Ethics Process, reference number 746665. Please take time to read the following information carefully.

Project title: Data-centric programming in Haskell
Principal investigator: Philip Wadler (*contact email*)
Researcher: Jakub Malý (*contact email*)

What is the purpose of this study?
Collect feedback about a new INF1A tutorial.

Why have I been asked to take part?
You are a current or past INF1A student.

Do I have to take part?
No, participation in this study is entirely up to you. You can withdraw from the study at any time until you submit the questionnaire without giving a reason. Your rights will not be affected.

What will happen if I decide to take part?
You will answer 10 questions about this tutorial and model solutions. The feedback is

fully anonymous and optional.

Are there any risks associated with taking part?
No.

Are there any benefits associated with taking part?
Engaging with the provided material can help your understanding of data science and functional programming.

What will happen to the results of this study?
The results of this study may be summarised in published articles, reports and presentations. Quotes or key findings will be anonymised: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a maximum of 1 year. No potentially identifiable data will be stored.

A.2 Participants' consent form

The students had to select 'I agree' or 'I disagree' for each of these two questions:

1. By participating in the study you agree that:
    - I have read and understood the information above, that I have had the opportunity to ask questions (via email to the Researcher or Principal Investigator), and that any questions I had were answered to my satisfaction.
    - My participation is voluntary, and that I can withdraw at any time before submitting the form without giving a reason. Withdrawing will not affect any of my rights.
    - I consent to my anonymised data being used in academic publications and presentations.
    - I understand that my anonymised data will be securely stored for up to 1 year.
2. I allow my data to be used in future ethically approved research, such as follow-up studies. (optional).

Appendix B

Tutorial text

Data Science in Haskell

INFORMATICS 1 – INTRODUCTION TO COMPUTATION

Functional Programming Optional Tutorial

This is an optional tutorial that you can complete to solidify your understanding of concepts such as I/O, functional composition and even learn some data science.

Introduction

In this tutorial, you will learn the tools needed to be able to do basic data science with datasets. You will use functional application and I/O through loading and analysing a dataset and even creating plots such as Figure B.1.

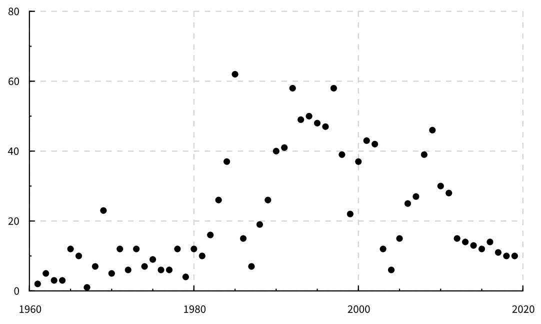

Figure B.1: Number of astronauts per year.

Installing the prerequisite tools

Most of the libraries you will use in this tutorial are a part of the dataHaskell initiative—a collection of data science oriented Haskell libraries.[7] More specifically, you will be using the `Frames` library to read the dataset, and the `Chart` library to visualise data.

First, since you will be working with libraries, you need to make sure you have a suitable version of GHC. To check your version, go to your command line and type:

```
cmd> ghc --version
```

If you have version 9.4.7, you're good to go! If not, you might run into trouble with incompatible versions so I recommend you install GHC 9.4.7. If you have followed the recommended set up process, you can use GHCup to install and set it as default:

```
cmd> ghcup install ghc 9.4.7
cmd> ghcup set ghc 9.4.7
```

Last, you can use your preferred package manager, such as Cabal, to install the required packages (note that the command is all in one line):

```
cmd> cabal install --lib vector split pipes microlens Frames
     Chart Chart-diagrams statistics-linreg
```

The dataset

You will be using a dataset which compiles publicly known information about astronauts who went to space prior to 2020.[20] The dataset is structured as a CSV (comma-separated-value) file. This means that each line contains several values separated by commas, forming a row of data corresponding to an observation. In our case, each line contains information about an astronaut that went to space on a specific mission (1 line per astronaut-mission pair).

You might come across this way of structuring the dataset referred to as tidy data. In essence, it allows us to add observations (rows) to our dataset without modifying the header (changing the number of columns). The first row of the dataset is a list of column headers (labels). You can find more details in Figure B.2.

You might have noticed that textual data is stored as `Text`. This is because `Text` is more efficient for storing large volumes of text while `String` makes modifying text easier. However, this tutorial uses the `OverloadedStrings` pragma, which allows you to treat `Text` like a regular `String` when writing your code.

Working with the dataset

In order for you to be able to work with the dataset, you need to load it into memory. However, since this is an I/O (input-output) operation, it is impure—if you change the dataset, then the function that reads the dataset into memory will have different outcomes!

Column name	Type	Description
`numberOverall`	`Int`	Astronaut world-wide number
`numberNationwide`	`Int`	Astronaut nation-wide number
`profileName`	`Text`	Astronaut full name
`profileSex`	`Text`	Astronaut sex (male/female)
`profileYearBirth`	`Int`	Astronaut birth year
`profileNationality`	`Text`	Astronaut nationality
`profileMilitary`	`Text`	Affiliation with military (military/civilian)
`profileYearSelection`	`Int`	Selection year
`profileMissionsNumber`	`Int`	Astronaut's mission number
`profileMissionsTotal`	`Int`	Astronaut's lifetime missions
`profileMissionsDuration`	`Double`	Astronaut's lifetime mission duration (hrs)
`profileMissionsEVA`	`Double`	Astronaut's lifetime EVA duration (hrs)
`missionRole`	`Text`	Astronaut's role on mission
`missionYear`	`Int`	Mission year
`missionName`	`Text`	Mission name
`missionVehiclesAscent`	`Text`	Ascent vehicle name
`missionVehiclesOrbit`	`Text`	Orbit vehicle name
`missionVehiclesDescent`	`Text`	Descent vehicle name
`missionDurationTotal`	`Double`	Mission duration (hrs)
`missionDurationEVA`	`Double`	Astronaut's EVA duration (hrs)

Figure B.2: Description of all columns in the dataset. EVA stands for extravehicular activity.

There are two ways to work with I/O in Haskell: writing I/O functions and using GHCi. Making I/O functions is a good way to write down your solutions. Here is a simple I/O function that takes a string and prints it out in ALLCAPS when you call it in GHCi.

```
shout :: IO ()
shout message = do
    let bigMessage = toUpper message
    print bigMessage
```

In general, an I/O method contains a `do` block where each line corresponds to an instruction to be executed when it is called. You can also use standard `let` syntax inside the `do` block, but each `do` block must end with an I/O action—such as printing out text into the console with `print`. If you name your function `main`, it will be executed when the `.hs` file is run.

You can also use GHCi. If you enter an expression that has a type `IO a` into GHCi, then the interpreter will execute that expression and print out the result if it can. The `Tutorial.hs` file defines a function

```
loadDataset :: IO (Frame Astronaut)
```

which you can read as "perform an I/O action that gives me a frame of astronauts". The `Frame` is our dataset, and the `Astronaut` is our observation (row). Let's use it!

```
ghci> :l Tutorial.hs
ghci> astronauts <- loadDataset
```

The `<-` syntax binds the result of the I/O function to a variable `astronauts`, so we have a name we can use to refer to the dataset. But how does it look like? If you try

```
ghci> astronauts
```

it will give you an error: no instance for `Show (Frame Astronaut)`. GHCi didn't print anything into console because it doesn't know how to convert the frame into a string.

To actually view the dataset, you can use the function

```
  viewDataset :: Show a => Int -> Frame a -> IO ()
```

defined in the tutorial. It takes an integer `n` and the frame and performs an I/O action that prints the first `n` rows into the console. Alternatively, you can use the `printDataset` function, which takes the same arguments but prints the first `n` rows as formatted columns.

It might be a bit messy, since there are a lot of columns. Let's try to look at only a few columns at a time. We start by typing `:i Astronaut` into GHCi to see the declaration. You will find that `Astronaut` is of type `Record`, which is the internal representation of a row. We can then write a lens—a function that extracts only some fields from each record, and map it over the frame. For example, the following code produces the first ten rows with columns for the mission year, name, and astronaut name. Note that we need to set the `XDataKinds` pragma for us to be able to supply types in a list like this.

```
ghci> :set -XDataKinds
ghci> lens = Frames.rcast @'[MissionYear, MissionName, ProfileName]
ghci> printDataset 10 $ lens <$> astronauts
```

1) Basic operations

Now that we have a dataset, let's calculate some statistics. You can extract a column from the dataset via the function

```
getColumn :: Frame a -> Getting b a b -> [b]
```

The type signature says that `getColumn` takes a frame and a `Getting` function and extracts a column from the frame as a list. `Frames` conveniently defines `Getting` objects to have the same form as the column names. For example, you can get all the astronaut names by typing

```
ghci> names = getColumn astronauts profileName
```

> A good way to approach a data science problem is to break it down into a list of steps that you need to do. For each step, figure out your input and desired output, then find a function that transforms or reduces your dataset. You can repeat this process until you get the final result.

1.a) How long is a mission?

The `Helpers.hs` file includes a function `stats` that calculates the minimum, mean, median, and maximum of a list of `Doubles`. Use it to calculate statistics about the total mission durations. Then, use the `filter` method to filter out zero duration mission entries, and their number. Is there a lot of them? How much did they impact the statistics?

A quick aside on mean and median: a mean is the average value of a list, whereas the median is the value in the middle of a (sorted) list. Comparing these two values can give you some insights about the shape of your data. For example, if your median is much smaller than your mean, it tells you that there are lots of small and a few very large values! But it's not perfect—it can't show you stuff like bimodal data (data with two peaks), which is why exercise 3 shows you how to make plots to fully visualise the dataset.

`[!]` You should check your solutions as you go, since the exercises build on top of each other. You can load `Solutions.hs` into GHCi and call `exercise1aSolution`, or call `exercise1aPrint` to see the solution code. Keep in mind that there are many ways to get the correct result!

1.b) What is the training time and age of an astronaut?

Okay, cool, we can calculate statistics about one column. Let's try more! Let's calculate the approximate age of an astronaut during a mission. We can get a rough idea by subtracting the astronaut's birth year from the mission year. The easiest way to do this is to read both columns, then `zip` them together into tuples and map a function over the list. Since these columns are `Ints`, you need to convert them to floating point numbers before you can use `stats`, so you need to use (stats . map fromIntegral) on the list instead.

We can also estimate the amount of training time they got by subtracting the selection year instead of birth year. However, each astronaut was only selected once, even though they might have multiple missions. You will need the `profileMissionsNumber` column to filter by first mission. You might find the `zip3` function useful.

1.c) What percentage of mission time is spent on EVAs?

Astronauts do many cool things in space: they float, they look at space, and they go out in their spacesuits! The last is called an extravehicular activity, but that is a mouthful so everyone just calls them EVAs. In this task, you need to calculate the percentage of time astronauts spend doing EVAs during a mission. Since there are multiple entries for each mission (one per astronaut), you will need to get one entry per mission. You might find the `nubBy` function useful. It takes a function which can be used to determine the equality of two elements in a list, and then removes duplicates:

```
ghci> nubBy (\(a,_) (b,_) -> a==b) [(1,"hi"), (2, "hello"), (1, "bye!")]
[(1, "hi"), (2, "hello")]
```

2) Aggregating data

2.a) What is the number of astronauts per country?

In this exercise, we want to calculate the number of astronauts per country as a list of ("Country", n) :: (String, Int) tuples. Remember that there can be multiple entries per astronaut, so you need to filter out the duplicates.

You might want to use the methods (group . sort) which first sorts a list and then groups elements next to each other into sub-lists based on equality:

```
ghci> (group . sort) ["a", "b", "a", "c", "b"]
[["a", "a"], ["b", "b"], ["c"]]
```

2.b) What is the percentage of female astronauts before and after the cold war?

A lot of the early space missions had a political significance in the conflict between the USA and the USSR. In this exercise, we want to calculate the percentage of astronauts that were female before and after this conflict. The cold war ended in late 1991, so include this year in your before group.

`[?]` Do you think that there is a causal relation? What other factors can you think of that could have influenced the ratio of women and men in space? It is always good to keep in mind that correlation does not always imply causality!

You can also use the `linearRegression` function defined in `Helpers.hs`

```
linearRegression :: [Double] -> [Double] -> (Double, Double, Double)
```

to make a linear fit through the data. It takes `xs` and `ys` and makes a (α, β, R^2) tuple. The fit line can be made via $y = \alpha + \beta x$ and R^2 is the coefficient of determination. It describes the goodness of fit (1 if the predictors fit perfectly and 0 if the have no explanatory power).

3) Plotting data

So far, you have done a lot of analysis, but it's often very hard to get a full sense of data from just numbers. As the old saying goes, a graph is worth a thousand statistics. The `Plotter.hs` file includes two pre-defined functions you can use for plotting: a scatter plot for a list of x and a list of y values; and a line plot that creates two lines for two lists of y values corresponding to one list of x values. Both functions have a type constraint `PlotValue`, which includes both `Int` and `Double`.

```
scatterPlot :: (PlotValue x, PlotValue y) => FileName -> [(x, y)]
            -> AxisTitle -> AxisTitle -> Title -> IO ()
linePlot2   :: (PlotValue x, PlotValue y) => FileName
            -> [(x, y)] -> [(x, y)] -> Label -> Label
            -> AxisTitle -> AxisTitle -> Title -> IO ()
```

Both plot functions take a file name (not a file path!); a list of (x, y) value tuples; and x-axis, y-axis and plot titles. For the line plot, the function also needs two labels corresponding to the two y values to create the legend. Note that `FileName`, `Label`, `AxisTitle`, and `Title` are all `String` type synonyms. These functions create a SVG (vector graphics file) plot inside the plots folder, which you can view in your browser.

These functions are sufficient for this exercise, but I encourage you to play with the dataset to answer any questions you might have. You can change the styling, as well as add more lines / points by adding another call to the `plot` function (see `scatterPlot`, `linePlot2`). Note that the line plot takes a list whereas the points plot takes a value. For even more plot options, you can look at the `Chart` library documentation.[?]

3.a) What are the trends in the number of missions per year?

In this exercise, we want to plot the number of missions per year. Once you have the lists for years (`years`) and counts (`counts`), you can call the plotting function as follows:

```
scatterPlot "filename" (zip years counts) "Year" "Missions/Year" "Title"
```

You can also calculate the same statistics as before via the `stats` function or make a fit via `linearRegression`. Are the values what you would expect from the plot?

`[?]` Just like in the previous exercise, I encourage you to think of possible factors that could have influenced this trend. Do you think there are correlations between your answers for (2b) and (3a)? Does the visualisation change your answers?

3.b) What is the total amount of time all missions have spent in space?

This exercise combines everything that you have learned in this tutorial. We want to figure out the cumulative time spent in space (both total mission time and EVA mission time) per year and then plot it as two lines using the `linePlot2` function. You might need to scale EVA time to see it. Here are a few functions that you might find useful:

```
sortOn  :: Ord b => (a -> b) -> [a] -> [a]
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
scanl   :: (b -> a -> b) -> b -> [a] -> [b]
```

The `sortOn` function is just like `sort`, but you provide a transformation function. The `groupBy` function works similarly to `group`, grouping elements based on an equality function you provided. Last, `scanl` works exactly like the `foldl` function, but it creates a list where each element is one step of the fold operation—you can use `scanl` for cumulative sums.

```
ghci> sortOn length ["aaa", "b", "cc"]
["b", "cc", "aaa"]
ghci> groupBy (\\a b -> length a == length b) ["aa", "b", "c", "dd"]
[["aa"], ["b", "c"], ["dd"]]
```

```
ghci> scanl (++) "start" ["One", "Two", "Three"]
["start", "startOne", "startOneTwo", "startOneTwoThree"]
```

Benefits of using Haskell

Now that you have done some data analysis in Haskell, you might ask yourself why you would want to use it over languages which are well-established as the de-facto standard of data analysis like Python, R, SQL. The answer: lazy computation!

The cool part about Haskell is that even its I/O is lazy. Throughout the tutorial, the computer never evaluated more than it had to, and more importantly, it also did not read more data than it had to. This means that Haskell is extremely well-suited for operating on very large datasets. Where a more mainstream language like Python or R would have to keep the whole dataset in-memory at all times to operate on it, Haskell only has to store what it currently needs for a computation.

But there are downsides to this mode of operation: each time you want to do anything with the dataset, you need to go and read from a file over and over again, and I/O is very slow compared to memory access. Therefore, whenever you can, you should always load your data to memory. This is what the `loadDataset` function does: it takes the lazy `astronautStream` function and forces it to produce all its rows at once so it can store it as a `Frame` in memory. This is helpful when exploring a small dataset like you did in this tutorial, but it is not necessary.

Remember that data analysis is just a series of steps that transforms and reduces the dataset. This means that you can express your whole data analysis as a sequence of steps that can be applied at once to a stream of data. With a little bit of extra tooling, you can use what you have learned today to work on datasets way larger than your computer's memory!

A lazy example

We can use the `+s` flag in GHCi to display the time taken and total memory used. Let's use it to see Haskell be lazy:

```
ghci> :set +s
ghci> astronauts <- loadDataset
(0.30 secs, 41,698,360 bytes)
ghci> names = getColumn astronauts profileName
(0.01 secs, 16,632 bytes)
ghci> take 3 names
(0.01 secs, 105,554 bytes)
ghci> take 30 names
(0.11 secs, 579,520 bytes)
ghci> names
(1.74 secs, 23,043,536 bytes)
```

From these results, we can see that `names` is indeed just a promise of reading the names in the future when needed. We also see that if we only take a few names, the amount of total memory used is smaller, since only the first 3 (or 30) names had to be read.

While this gives us a general idea of what is happening under the hood, we can go one step further! If we put the commands we just ran through GHCi into a `main` method and then compile the program with GHC, we can run a more rigorous profiling:

```
cmd> ghc -rtsopts --make Tutorial.hs
cmd> Tutorial.exe +RTS -s
```

We make two programs, one which prints `take 3 names`, and the other that prints all of `names`. Both of them have about 42 megabytes allocated in the heap, but the maximum residency (maximum number of bytes occupied at any one time) changes from around 230kb to around 1060kb—that is a four-times increase between printing 3 and over 1200 names! This is exactly what we would expect, since Haskell always reads one name and immediately prints it, freeing up more space.

We can take this a step further by using streams instead of loading the dataset into memory. Feel free to have a look at the `AltSolutions.hs` file which uses the `astronautStream` and the `pipes` library to solve the same exercises without ever loading the whole dataset into memory.

Appendix C

Tutorial code

```
tutorial/
|--plots/
|--|--number_of_astronauts.svg
|--datasets/
|--|--astronauts.csv
|--Helpers.hs
|--Plotter.hs
|--Tutorial.hs
|--Solutions.hs
|--AltSolutions.hs
tutorial.pdf
```

Table C.1: **File structure directory.** Files are hyperlinked.

C.1 Tutorial.hs

```
{-# LANGUAGE DataKinds, OverloadedStrings, TemplateHaskell #-}

module Tutorial where

--------------------------------------------------------------
--     module                              library required --
--------------------------------------------------------------
import Pipes             (Producer)     -- pipes
import Lens.Micro        (Getting)      -- microlens
import Lens.Micro.Extras (view)         -- microlens
import Frames                           -- Frames
import Data.List
import qualified Data.Foldable as F

import Helpers
import Plotter
```

```
---------------------
-- data definition --
---------------------

-- Create a row type (inferred from a CSV file).
tableTypes "Astronaut" "datasets/astronauts.csv"

-- Create a stream of rows (Astronaut) from a CSV file.
astronautStream :: MonadSafe m => Producer Astronaut m ()
astronautStream = readTableOpt astronautParser "datasets/astronauts.csv"

-- Convert the stream into an in-memory data frame.
loadDataset :: IO (Frame Astronaut)
loadDataset = inCoreAoS astronautStream

-- View the first n rows of a frame as records.
viewDataset :: Show a => Int -> Frame a -> IO ()
viewDataset n frame = mapM_ (print . frameRow frame) [0..(n-1)]
-- Pretty printing alternative: view the first n rows
-- of a frame as data-only columns.
printDataset n frame = putStrLn $ prettyFormat "{|}" $ showFrame "{|}"
                              $ takeRows n frame

-- Get a column  of the dataset as a list.
getColumn :: (Foldable t, Functor t) => t a -> Getting b a b -> [b]
getColumn frame col = F.toList $ view col <$> frame

---------------
-- exercises --
---------------

-- exercise 1a: calculating column statistics,
--              mission duration

exercise1a :: IO ()
exercise1a = do
    putStrLn "your code goes here"

-- exercise 1b: grouping by astronaut
--              calculating training time and age

exercise1b :: IO ()
exercise1b = do
    putStrLn "your code goes here"

-- exercise 1c: grouping by mission,
--              calculating percentage of EVA time
```

```
exercise1c :: IO ()
exercise1c = do
    putStrLn "your code goes here"

-- exercise 2a: aggregating by country,
--              number of astronauts per country

exercise2a :: IO ()
exercise2a = do
    putStrLn "your code goes here"

-- exercise 2b: aggregating by time period
--              gender ratio before and after the cold war

exercise2b :: IO ()
exercise2b = do
    putStrLn "your code goes here"

-- exercise 3a: number of missions per year

exercise3a :: IO ()
exercise3a = do
    putStrLn "your code goes here"

-- exercise 3b: cumulative sums
--              total amount of hours spent in space

exercise3b :: IO ()
exercise3b = do
    putStrLn "your code goes here"
```

C.2 Helpers.hs

```
module Helpers where

----------------------------------------------------------
--     module                          library required --
----------------------------------------------------------
import Data.List
import Data.List.Split              -- split
import Frames                       -- Frames
import Data.Vector (fromList)       -- vector
import Statistics.LinearRegression  -- statistics-linreg

-------------------------
-- exploration helpers --
-------------------------
```

```
-- Pretty print a frame by aligning eac column with enough spaces to
--   pad to (maximum word length in column) + 1.
prettyFormat :: String -> String -> String
prettyFormat sep input = unlines
                       $ map (addPadding . zip splitWidth) splitWords
    where splitWords = (map (splitOn sep) . lines) input
          splitWidth = (map (maximum . (map length))
                                    . transpose) splitWords
          addPadding = (concat . map (\(len, str)
                        -> str ++ replicate (1 + len - length str) ' '))

---------------------------------
-- helper operations on tuples --
---------------------------------

-- Extracts the first element of a tuple.
--   (3tuple equivalent of fst)
fst3 :: (a, b, c) -> a
fst3 (x, _, _) = x

-- Produces a tuple without its first element.
--   (3tuple equivalent of tail)
tls3 :: (a, b, c) -> (b, c)
tls3 (_, y, z) = (y, z)

-------------------------------
-- helper statistics methods --
-------------------------------

-- Calculates the mean of a list of values.
mean :: Fractional a => [a] -> a
mean [] = error "mean: undefined on empty list"
mean xs = (sum xs) / (fromIntegral $ length xs)

-- Calculate the median of a list of values.
median :: (Fractional a, Ord a) => [a] -> a
median xs
    | l == 0    = error "median: undefined on empty list"
    | odd l     =   x !! m
    | otherwise = ((x !! m) + (x !! (m - 1))) / 2
        where l = length xs
              m = l `div` 2
              x = sort xs

-- Calculates the minimum, mean, median, and maximum of a list.
stats :: (Fractional a, Ord a) => [a] -> [a]
stats xs = map ($ xs) [minimum, mean, median, maximum]

-- Given a list of responders (xs) and predictors (ys),
--   calculates best linear fit of data as a (a, b, R^2) tuple
--   where the line is given by y = a + bx
--   and R^2 is the coefficient of determination.
--   Note that xs and ys should have the same length.
linearRegression :: [Double] -> [Double] -> (Double, Double, Double)
linearRegression xs ys = linearRegressionRSqr (fromList xs) (fromList ys)
```

C.3 `Plotter.hs`

```
module Plotter where

import Graphics.Rendering.Chart.Easy hiding (Getting, view)
import Graphics.Rendering.Chart.Backend.Diagrams

-- create type synonyms for plots
type FileName  = String
type Title     = String
type AxisTitle = String
type Label     = String

-- create a new file options type
fileOptions = FileOptions (1000, 600) SVG loadSansSerifFonts

-- define custom font styles
fontStyle = FontStyle
    { _font_name   = "sans-serif"
    , _font_size   = 30
    , _font_slant  = def            -- default value
    , _font_weight = def            -- default value
    , _font_color  = opaque black
    }

titleFontStyle = FontStyle
    { _font_name   = "sans-serif"
    , _font_size   = 30
    , _font_slant  = def            -- default value
    , _font_weight = FontWeightBold
    , _font_color  = opaque black
    }

-- function that creates a scatter plot in the plots folder
scatterPlot :: (PlotValue x, PlotValue y) => FileName -> [(x, y)] ->
            -> AxisTitle -> AxisTitle -> Title -> IO ()
scatterPlot fileName data xtitle ytitle title =
    toFile fileOptions ("plots/" ++ fileName ++ ".svg") $ do
        layout_all_font_styles      .= fontStyle
        layout_title_style          .= titleFontStyle
        layout_title                .= title
        layout_x_axis . laxis_title .= xtitle
        layout_y_axis . laxis_title .= ytitle
        -- create a line plot with custom styling via liftEC, which
        -- nests computation of the points within the plot computation
        plot $ liftEC $ do
            plot_points_style  .= (filledCircles 5 $ opaque black)
            plot_points_values .= data

-- function that creates a line plot with 2 lines in the plots folder
linePlot2 :: (PlotValue x, PlotValue y) => FileName
          -> [(x, y)] -> [(x, y)] -> Label -> Label
          -> AxisTitle -> AxisTitle -> Title -> IO ()
linePlot2 fileName vals1 vals2 label1 label2 xtitle ytitle title =
    toFile fileOptions ("plots/" ++ fileName ++ ".svg") $ do
        layout_all_font_styles      .= fontStyle
        layout_title_style          .= titleFontStyle
        layout_title                .= title
        layout_x_axis . laxis_title .= xtitle
        layout_y_axis . laxis_title .= ytitle
        -- make a solid blue line, width 3, with legend label label1
        plot $ liftEC $ do
            plot_lines_style  .= (solidLine 3 $ opaque blue)
            plot_lines_title  .= label1
            plot_lines_values .= [vals1] -- lineplot takes a list of lists
        -- make a dashed red line (pattern 5 line, 3 space), width 3
        plot $ liftEC $ do
            plot_lines_style  .= (dashedLine 3 [5, 3] $ opaque red)
            plot_lines_title  .= label2
            plot_lines_values .= [vals2]
```

C.4 Solutions.hs

```
{-# LANGUAGE DataKinds, OverloadedStrings, TemplateHaskell #-}

module Solutions where

import Data.List

import Tutorial
import Helpers
import Plotter

------------------------
-- exercise solutions --
------------------------

-- exercise 1a: calculating column statistics,
--              mission duration
```

```
exercise1aSolution :: IO ()
exercise1aSolution = do
    frame <- loadDataset
    let ds = getColumn frame missionDurationTotal
    putStrLn $ "\nstatistics [min, mean, median, and max]
            ++ "for mission durations (hrs)"
    (print . stats) ds
    putStrLn $ "\nstatistics for mission durations (hrs), without
            ++ "zero-duration missions"
    (print . stats . filter (/= 0)) ds
    putStrLn "\nnumber of zero-duration missions"
    (print . length . filter (==0)) ds
    putStrLn ""

-- exercise 1b: grouping by astronaut
--              calculating training time and age

exercise1bSolution :: IO ()
exercise1bSolution = do
    frame <- loadDataset
    let get = getColumn frame
        n   = get profileMissionsNumber
        bYr = get profileYearBirth
        sYr = get profileYearSelection
        mYr = get missionYear
    putStrLn $ "\nstatistics for the age of astronauts at mission"
            ++ "(mission year - birth year) in years"
    let ages = map (\(b,m) -> m-b) $ zip bYr mYr
    -- convert to Double else since stats needs Fractional values
    (print . stats . map fromIntegral) ages
    putStrLn $ "\nstatistics for the training time per astronaut"
            ++ "(mission year - selection year) in years"
    -- need to filter out by astronaut (only select first mission entry)
    let firsts = map tls3 $ filter ((==1) . fst3) $ zip3 n sYr mYr
        trains = map (\(s,m) -> m-s) firsts
    (print . stats . map fromIntegral) trains
    putStrLn ""

-- exercise 1c: grouping by mission,
--              calculating percentage of EVA time

exercise1cSolution :: IO ()
exercise1cSolution = do
    frame <- loadDataset
    let ns = getColumn frame missionName
        ds = getColumn frame missionDurationTotal
        es = getColumn frame missionDurationEVA
    putStrLn "\nstatistics for the percentage of mission time spent on EVA"
    -- filter out by mission name (1 entry per mission)
    let missions = map tls3 $ nubBy (\(n1,_,_) (n2,_,_) -> n1 == n2)
                            $ zip3 ns es ds
    -- create valid pairs (mission duration > 0 to not divide by 0)
    let pairs = filter (\(_,d) -> d > 0) missions
    (print . stats . map (\(e,d) -> (e/d) * 100)) pairs
    putStrLn ""

-- exercise 2a: aggregating by country,
--              number of astronauts per country

exercise2aSolution :: IO ()
exercise2aSolution = do
    frame <- loadDataset
    let n   = getColumn frame profileMissionsNumber
        nat = getColumn frame profileNationality
    putStrLn "\nnumber of astronauts per country (nationality)"
    -- aggregate by astronaut (select first mission only)
    let nationalities = map snd $ filter ((==1) . fst) $ zip n nat
    -- aggregate nationalities into groups, then create
    -- (element, length) tuples
        countryCounts = (map (\l -> (head l, length l))
                        . group . sort) nationalities
    -- and sort by number of astronauts
    (print . sortOn (negate . snd)) countryCounts
    putStrLn ""

-- exercise 2b: aggregating by time period
--              gender ratio before and after the cold war

exercise2bSolution :: IO ()
exercise2bSolution = do
    frame <- loadDataset
    let mYs = getColumn frame missionYear
        sex = getColumn frame profileSex
    putStrLn $ "\nfraction of female astronauts before (inclusive)"
            ++ "and after 1991"
    -- filter out by year
    let pairs  = zip mYs sex
        before = filter ((<=1991) . fst) pairs
        after  = filter ((>1991)  . fst) pairs
    -- aggregate by astronaut sex, calculate percentage
    let beforeCount = length $ filter ((=="female") . snd) before
        afterCount  = length $ filter ((=="female") . snd) after
    print $ 100 * (fromIntegral beforeCount)
                / (fromIntegral $ length before)
    print $ 100 * (fromIntegral afterCount )
                / (fromIntegral $ length after)
```

```
-- exercise 3a: number of missions per year

exercise3aSolution :: IO ()
exercise3aSolution = do
    frame <- loadDataset
    let names = getColumn frame missionName
        years = getColumn frame missionYear
    -- filter out by mission name (1 entry per mission)
    let missions = map snd $ nubBy (\(n1,_) (n2,_) -> n1 == n2)
                           $ zip names years
        groups = (group . sort) missions
        xs = map head groups
        ys = map length groups
    scatterPlot "number_of_missions" (zip xs ys)
                "Year" "Number of missions" "Number of missions per year"

-- exercise 3b: cumulative sums
--              total amount of hours spent in space

exercise3bSolution :: IO ()
exercise3bSolution = do
    frame <- loadDataset
    let ns = getColumn frame missionName
        ys = getColumn frame missionYear
        ts = getColumn frame missionDurationTotal
        es = getColumn frame missionDurationEVA
        rows = zip4 ns ys ts es
    -- aggregate by mission into (year, total time, EVA time) tuples
    let missions = map (\(_,y,t,e) -> (y,t,e))
                 $ nubBy (\(n1,_,_,_) (n2,_,_,_) -> n1 == n2) rows
    -- group by year
    let yearGroups = (groupBy (\a b -> fst3 a == fst3 b)
                     . sortOn fst3) missions
    -- extract years
    let years = map (fst3 . head) yearGroups
    -- calculate sums per year
    let times = map (foldl (\(tSum, eSum) (_, t, e)
                           -> (tSum+t, eSum+e)) (0, 0)) yearGroups
    -- convert to years and calculate cumulative sums
    let cSumTotal =              map (/8760) $ scanl (+) 0 $ map fst times
        cSumEVAs  = map (*100)  $ map (/8760) $ scanl (+) 0 $ map snd times
    linePlot2 "time_in_space" (zip years cSumTotal) (zip years cSumEVAs)
              "Total time" "100x EVA time"
              "Year" "Time spent in space [yrs]" "Mission time"

-----------------------
-- solution printers --
-----------------------
```

```
exercise1aPrint :: IO ()
exercise1aPrint = do putStrLn
    $ "Exercise (1.a) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let ds = getColumn frame missionDurationTotal"
    ++ "\n|  (print . stats) ds"
    ++ "\n|  (print . stats . filter (/= 0)) ds"
    ++ "\n|  (print . length . filter (==0)) ds"

exercise1bPrint :: IO ()
exercise1bPrint = do putStrLn
    $ "Exercise (1.b) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let get = getColumn frame"
    ++ "\n|      n   = get profileMissionsNumber"
    ++ "\n|      bYr = get profileYearBirth"
    ++ "\n|      sYr = get profileYearSelection"
    ++ "\n|      mYr = get missionYear"
    ++ "\n|      ages = map (\\(b,m) -> m-b) $ zip bYr mYr"
    ++ "\n|  -- convert to Double since stats needs Fractional values"
    ++ "\n|  (print . stats . map fromIntegral) ages"
    ++ "\n|  -- need to filter out by astronaut (only select"
    ++ " first mission entry)"
    ++ "\n|  let firsts = map tls3 $ filter ((==1) . fst3)"
    ++ " $ zip3 n sYr mYr"
    ++ "\n|      trains = map (\\(s,m) -> m-s) firsts"
    ++ "\n|  (print . stats . map fromIntegral) trains"

exercise1cPrint :: IO ()
exercise1cPrint = do putStrLn
    $ "Exercise (1.c) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let ns = getColumn frame missionName"
    ++ "\n|      ds = getColumn frame missionDurationTotal"
    ++ "\n|      es = getColumn frame missionDurationEVA"
    ++ "\n|  -- filter out by mission name (1 entry per mission)"
    ++ "\n|  let missions = map tls3 $ nubBy (\\(n1,_,_) (n2,_,_)"
    ++ " -> n1 == n2) $ zip3 ns es ds"
    ++ "\n|  -- create valid pairs (mission duration > 0 to not"
    ++ " divide by 0)"
    ++ "\n|  let pairs = filter (\\(_,d) -> d > 0) missions"
    ++ "\n|  (print . stats . map (\\(e,d) -> (e/d) * 100)) pairs"

exercise2aPrint :: IO ()
exercise2aPrint = do putStrLn
    $ "Exercise (2.a) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let n   = getColumn frame profileMissionsNumber"
    ++ "\n|      nat = getColumn frame profileNationality"
    ++ "\n|  -- aggregate by astronaut (select first mission only)"
    ++ "\n|  let nationalities = map snd $ filter ((==1) . fst)"
    ++ " $ zip n nat"
    ++ "\n|  -- aggregate nationalities into groups, then create"
    ++ " (element, length) tuples"
    ++ "\n|  let countryCounts = (map (\\l -> (head l, length l))"
    ++ " . group . sort) nationalities"
    ++ "\n|  -- and sort by number of astronauts"
    ++ "\n|  (print . sortOn (negate . snd)) countryCounts"

exercise2bPrint :: IO ()
exercise2bPrint = do putStrLn
    $ "Exercise (2.b) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let mYs = getColumn frame missionYear"
    ++ "\n|      sex = getColumn frame profileSex"
    ++ "\n|  -- filter out by year"
    ++ "\n|  let pairs  = zip mYs sex"
    ++ "\n|      before = filter ((<=1991) . fst) pairs"
    ++ "\n|      after  = filter ((>1991)  . fst) pairs"
    ++ "\n|  -- aggregate by astronaut sex, calculate percentage"
    ++ "\n|  let beforeCount = length $ filter ((==\"female\")"
    ++ " . snd) before"
    ++ "\n|      afterCount  = length $ filter ((==\"female\")"
    ++ " . snd) after"
    ++ "\n|  print $ 100 * (fromIntegral beforeCount)"
    ++ " / (fromIntegral $ length before)"
    ++ "\n|  print $ 100 * (fromIntegral afterCount )"
    ++ " / (fromIntegral $ length after)"

exercise3aPrint :: IO ()
exercise3aPrint = do putStrLn
    $ "Exercise (3.a) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let names = getColumn frame missionName"
    ++ "\n|      years = getColumn frame missionYear"
    ++ "\n|  -- filter out by mission name (1 entry per mission)"
    ++ "\n|  let missions = map snd $ nubBy (\\(n1,_) (n2,_)"
    ++ " -> n1 == n2) $ zip names years"
    ++ "\n|      groups = (group . sort) missions"
    ++ "\n|      xs = map head groups"
    ++ "\n|      ys = map length groups"
    ++ "\n|  scatterPlot \"number_of_missions\" (zip xs ys)"
    ++ "\n|              \"Year\" \"Number of missions\""
    ++ " \"Number of missions per year\""

exercise3bPrint :: IO ()
exercise3bPrint = do putStrLn
    $ "Exercise (3.b) solution:"
    ++ "\n|  frame <- loadDataset"
    ++ "\n|  let ns = getColumn frame missionName"
    ++ "\n|      ys = getColumn frame missionYear"
    ++ "\n|      ts = getColumn frame missionDurationTotal"
    ++ "\n|      es = getColumn frame missionDurationEVA"
    ++ "\n|      rows = zip4 ns ys ts es"
    ++ "\n|  -- aggregate by mission into (year, total time,"
    ++ " EVA time) tuples"
    ++ "\n|  let missions = map (\\(_,y,t,e) -> (y,t,e)) $ nubBy"
    ++ " (\\(n1,_,_,_) (n2,_,_,_) -> n1 == n2) rows"
    ++ "\n|  -- group by year"
    ++ "\n|  let yearGroups = (groupBy (\\a b -> fst3 a == fst3 b)"
    ++ " . sortOn fst3) missions"
    ++ "\n|  -- extract years"
    ++ "\n|  let years = map (fst3 . head) yearGroups"
    ++ "\n|  -- calculate sums per year"
    ++ "\n|  let times = map (foldl (\\(tSum, eSum) (_, t, e)"
    ++ " -> (tSum+t, eSum+e)) (0, 0)) yearGroups"
    ++ "\n|  -- convert to years and calculate cumulative sums"
    ++ "\n|  let cSumTotal =              map (/8760) $ scanl"
    ++ " (+) 0 $ map fst times"
    ++ "\n|      cSumEVAs  = map (*100)  $ map (/8760) $ scanl"
    ++ " (+) 0 $ map snd times"
    ++ "\n|  linePlot2 \"time_in_space\" (zip years cSumTotal)"
    ++ " (zip years cSumEVAs)"
    ++ "\n|            \"Total time\" \"100x EVA time\""
    ++ "\n|            \"Year\" \"Time spent in space [yrs]\""
    ++ " \"Mission time\""
```

C.5 AltSolutions.hs

```
{-# LANGUAGE DataKinds, OverloadedStrings, TemplateHaskell #-}

module AltSolutions where

---------------------------------------------------------------
--    module                     alias    library required --
---------------------------------------------------------------
import Data.List
import Frames                                -- frames
import Lens.Micro.Extras (view)              -- microlens
import Pipes                                 -- pipes
import qualified Pipes.Prelude as P          -- pipes
import qualified Control.Foldl as L          -- foldl

import Tutorial
import Helpers
import Plotter
```

```
-- Reduces a stream with the provided fold operation.
foldStream fold stream = runSafeEffect $ (L.purely P.fold) fold $ stream

------------------------
-- exercise solutions --
------------------------

-- exercise 1a: calculating column statistics,
--              mission duration

exercise1aSolution' :: IO ()
exercise1aSolution' = do
    let s = astronautStream >-> P.map (view missionDurationTotal)
    dsFull     <- foldStream L.list s
    dsFiltered <- foldStream L.list $ s >-> P.filter (/=0)
    print $ stats dsFull
    print $ stats dsFiltered
    print $ (length dsFull) - (length dsFiltered)

-- exercise 1b: grouping by astronaut
--              calculating training time and age

exercise1bSolution' :: IO ()
exercise1bSolution' = do
    ages <- foldStream L.list $ astronautStream
            >-> P.map (\r -> view missionYear r - view profileYearBirth r)
    (print . stats . map fromIntegral) ages
    -- need to filter out by astronaut (only select first mission entry)
    trains <- foldStream L.list $ astronautStream
              >-> P.filter ((==1) . view profileMissionsNumber)
              >-> P.map (\r -> view missionYear r
                             - view profileYearSelection r)
    (print . stats . map fromIntegral) trains

-- exercise 1c: grouping by mission,
--              calculating percentage of EVA time

exercise1cSolution' :: IO ()
exercise1cSolution' = do
    missions <- foldStream L.list
                $ astronautStream
                >-> P.filter ((/=0) . view missionDurationTotal)
                >-> P.map (\r -> (view missionName r,
                                  100 * view missionDurationEVA r
                                      / view missionDurationTotal r))
    -- filter out by mission name (1 entry per mission)
    (print . stats . map snd . nubBy (\l r -> fst l == fst r)) missions
```

```
-- exercise 2a: aggregating by country,
--              number of astronauts per country

exercise2aSolution' :: IO ()
exercise2aSolution' = do
    -- aggregate by astronaut (select first mission only)
    nats <- foldStream L.list $ astronautStream
            >-> P.filter ((==1) . view profileMissionsNumber)
            >-> P.map (view profileNationality)
    -- aggregate nationalities into groups; create (element, length)
    -- tuples; then sort by count
    (print . sortOn (negate . snd) . map (\l -> (head l, length l))
           . group . sort) nats

-- exercise 2b: aggregating by time period
--              gender ratio before and after the cold war

exercise2bSolution' :: IO ()
exercise2bSolution' = do
    before <- foldStream L.list $ astronautStream
              >-> P.filter ((<=1991) . view missionYear)
              >-> P.map (fromEnum . (=="female") . view profileSex)
    after <- foldStream L.list $ astronautStream
             >-> P.filter ((>1991) . view missionYear)
             >-> P.map (fromEnum . (=="female") . view profileSex)
    print $ 100 * (fromIntegral $ sum before)
                / (fromIntegral $ length before)
    print $ 100 * (fromIntegral $ sum after)
                / (fromIntegral $ length after)

-- exercise 3a: number of missions per year

exercise3aSolution' :: IO ()
exercise3aSolution' = do
    missions <- foldStream L.list $ astronautStream
                >-> P.map(\r -> (view missionName r, view missionYear r))
    let groups = (group . sort . map snd
                 . nubBy (\l r -> fst l == fst r)) missions
    scatterPlot "number_of_missions"
                (map (\l -> (head l, length l)) groups)
                "Year" "Number of missions" "Number of missions per year"

-- exercise 3b: cumulative sums
--              total amount of hours spent in space

exercise3bSolution' :: IO ()
exercise3bSolution' = do
    rows <- foldStream L.list $ astronautStream
            >-> P.map (\r -> (view missionName r, view missionYear r,
                          view missionDurationTotal r,
                          view missionDurationEVA r))
    -- aggregate by mission into (year, total time, EVA time) tuples
    let missions = map (\(_,y,t,e) -> (y,t,e))
                $ nubBy (\(n1,_,_,_) (n2,_,_,_) -> n1 == n2) rows
    -- group by year
    let yearGroups = (groupBy (\a b -> fst3 a == fst3 b)
                    . sortOn fst3) missions
    -- extract years
    let years = map (fst3 . head) yearGroups
    -- calculate sums per year
    let times = map (foldl (\(tSum, eSum) (_, t, e)
                   -> (tSum+t, eSum+e)) (0, 0)) yearGroups
    -- convert to years and calculate cumulative sums
    let cSumTotal = map (/8760) $ scanl (+) 0 $ map fst times
        cSumEVAs  = map (*100)  $ map (/8760)
                  $ scanl (+) 0 $ map snd times
    linePlot2 "time_in_space" (zip years cSumTotal) (zip years cSumEVAs)
              "Total time" "100x EVA time"
              "Year" "Time spent in space [yrs]" "Mission time"
```

Appendix D

Benchmark code

D.1 `Bench.hs`

```
{-# LANGUAGE DataKinds, OverloadedStrings, TemplateHaskell,
    TypeApplications, FlexibleContexts, DeriveGeneric, BangPatterns #-}

module Bench where

------------------------------------------------------------------------
--     module                     alias            library required --
------------------------------------------------------------------------

-- benchmark
import Data.Time.Clock
import Control.Monad (forM_)

-- lists, vectors, folds
import Data.List
import qualified Data.Foldable as F
import qualified Data.Text     as T                -- text
import qualified Data.Vector   as V                -- vector
import qualified Control.Foldl as L hiding (Vector) -- foldl

-- frames
import qualified Frames.CSV                        -- Frames
import Frames                                      -- Frames
import Lens.Micro                     (Getting)    -- microlens
import Lens.Micro.Extras              (view)       -- microlens

-- streams
import qualified Pipes.Prelude as P                -- pipes
import Pipes                                       -- pipes

-- statistics
import Statistics.Sample.KernelDensity (kde_)      -- statistics
import Statistics.Sample               (skewness)   -- statistics
import Statistics.Correlation          (pearson)    -- statistics
import Data.KMeans                     (kmeans)     -- kmeans
import Statistics.LinearRegression     (linearRegression)
                                             -- statistics-linreg

-- plotting (requires Chart and Chart-diagrams)
import Graphics.Rendering.Chart.Easy hiding (Getting, view)
import Graphics.Rendering.Chart.Backend.Diagrams

---------------------
-- Micro benchmark --
---------------------

-- load data
tableTypes "Row" "datasets/data1_1000.csv"
rowStream :: MonadSafe m => Int -> Producer Row m ()
rowStream numRows = readTableOpt rowParser
                    ("datasets/data1_" ++ show numRows ++ ".csv")
readFrame :: Int -> IO (Frame Row)
readFrame numRows = inCoreAoS $ rowStream numRows

------------------
-- test runners --
------------------

-- forces and times a computation, prints time in seconds
benchmark computation = do
    t1 <- getCurrentTime
    let !r = computation
    t2 <- getCurrentTime
    (putStr . show . nominalDiffTimeToSeconds) $ diffUTCTime t2 t1

benchmark' ioComputation = do
    t1 <- getCurrentTime
    _ <- ioComputation
    t2 <- getCurrentTime
    (putStr . show . nominalDiffTimeToSeconds) $ diffUTCTime t2 t1

-- runs a test and prints result into console in a CSV-friendly format
run n frame test name = do
    putStr $ "Haskell-ghci," ++ show n ++ "," ++ name ++ ","
    _ <- test frame
    putStrLn ""

-- runs tests that can run for any size
do_tests_fast n f = do
    run n n read_test "Read"
    run n f write_test "Write"
    run n f sort_test "Sort"
    run n f filter_test "Filter"
    run n f groupby_test "Groupby"
    run n f central_test "Mean"
    run n f dispersion_test "Dispersion"
    run n f outliers_test "Outliers"
    putStr ""

-- runs tests that should not be run on datasets over 100k rows
do_tests_slow n f = do
    run n f duplicates_test "Duplicates"
    run n f scatter_test "Scatter"
    putStr ""

-- runs all the tests ten times for a given size
do_tests_full n r = do
    f <- readFrame n
    forM_ [1..r] $ \_ -> do
        do_tests_fast n f
        -- limits tests which take too long on large datasets
        if n <= 1000000 then run n f clustering_test "Clustering"
                        else putStr ""
        if n <= 100000  then do_tests_slow n f
                        else putStr ""

-- runs all the stream tests
do_tests_stream n r = do
    let s = rowStream n
    forM_ [1..r] $ \_ -> do
        run n s stream_one "Stream-mean"
        run n s stream_two "Stream-mean-std"

-- run all in-core tests
do_tests_core n r = do
    forM_ [1..r] $ \_ -> do
        run n n core_one "Core-mean"
        run n n core_two "Core-mean-std"

sizes = [ 1000
        , 10000
        , 100000
        , 1000000
        ]
repetitions = 10
main = forM_ sizes $ \n -> do
    do_tests_full n repetitions

--------------------
-- basic file I/O --
```

```
--------------------

read_test :: Int -> IO ()
read_test rows = benchmark' $ do
    frame <- readFrame rows
    return frame

write_test :: Frame Row -> IO ()
write_test frame = benchmark' $ do
    Frames.CSV.writeCSV "write_test.csv" frame

--------------------
-- data wrangling --
--------------------

sort_frame :: Ord l => Frame r -> Getting l r l -> Frame r
sort_frame frame lens = boxedFrame $ frameRow frame <$> idx
    where idx = map fst $ sortOn snd
                        $ zip ([0..frameLength frame - 1])
                              (F.toList $ view lens <$> frame)

sort_test :: Frame Row -> IO ()
sort_test frame = benchmark $ do
    sort_frame frame Bench.uni

filter_test :: Frame Row -> IO ()
filter_test frame = benchmark $ do
    filterFrame ((<500000) . view rand1) frame

groupby_test :: Frame Row -> IO ()
groupby_test frame = benchmark $ do
    map (sum . map fst)
           $ groupBy (\one two -> snd one == snd two)
           $ L.fold L.list
           $ ((,) <$> (view Bench.nor)
                  <*> (view Bench.city))
             <$> frame

duplicates_test :: Frame Row -> IO ()
duplicates_test frame = benchmark $ do
    boxedFrame $ frameRow frame <$> idx
    where idx = map fst $ reverse
                        $ nubBy (\two one -> snd two == snd one)
                        $ reverse
                        $ zip ([0..frameLength frame - 1])
                              (F.toList $ view Bench.words <$> frame)

----------------------------
-- descriptive statistics
```

```
----------------------------

central_test :: Frame Row -> IO ()
central_test frame = do
    let f = (fromIntegral . view rand2) <$> frame
    benchmark $ L.fold L.mean f

dispersion_test :: Frame Row -> IO ()
dispersion_test frame = do
    let f = ((,) <$> (view Bench.exp)
                 <*> (view Bench.uni))
            <$> frame
    benchmark $ L.fold ((,,) <$> (L.premap fst L.maximum)
                             <*> (L.premap fst L.minimum)
                             <*> (L.premap snd L.std))
              $ f

outliers_test :: Frame Row -> IO ()
outliers_test frame = do
    let (std, mean) = L.fold ((,) <$> L.std <*> L.mean)
                             $ view Bench.exp
                             <$> frame
    benchmark $ filterFrame (\r
                -> not $ (view Bench.exp r - mean) > (3 * std)) frame

scatter_test :: Frame Row -> IO ()
scatter_test frame = do
    let xs = L.fold L.list
           $ ((,) <$> (view Bench.uni)
                  <*> (view Bench.nor))
           <$> frame
    benchmark' $ toFile def ("plots/scatter_test.svg")
               $ plot $ points "" xs

----------------------
-- machine learning --
----------------------

clustering_test :: Frame Row -> IO ()
clustering_test frame = do
    let points = L.fold L.list $ (\r -> [((fromIntegral . view rand1) r),
                                         ((fromIntegral . view rand2) r)])
                               <$> frame
    benchmark $ kmeans 2 points

-----------------------
-- stream benchmarks --
-----------------------
```

```
-- Reduces a stream with the provided fold operation.
foldStream fold stream = runSafeEffect $ (L.purely P.fold) fold $ stream

stream_one stream = benchmark' $ do
    foldStream L.mean $ stream >-> P.map (fromIntegral . view rand1)

stream_two stream = benchmark' $ do
    foldStream ((,) <$> (L.premap (fromIntegral . fst) L.mean)
                    <*> (L.premap snd L.std))
               $ stream >-> P.map ((,) <$> (view rand1) <*> (view uni))

core_one n = benchmark' $ do
    frame <- readFrame n
    return $ L.fold L.mean $ (fromIntegral . view rand1) <$> frame

core_two n = benchmark' $ do
    frame <- readFrame n
    return $ L.fold ((,) <$> (L.premap (fromIntegral . fst) L.mean)
                         <*> (L.premap snd L.std))
                     $ ((,) <$> (view rand1) <*> (view uni)) <$> frame
```