MyCsv - DSL project report

Mathieu Poirier and Bastien Rousseau
Master students

29th November 2020

Contents

1 Introduction

MyCsv is a DSL (Domain-Specific Language) for CSV file manipulation. MyCsv is able to do classical operations on CSV such as selection, projection, insertion or modification. It can also manipulate algebraic and logical expressions for more expressive selections.

On top of those classical data manipulation operations, it is able to print lot of things, and eventually to export CSV data to the JSON format.

Our implementations and some other useful documents are gathered in the following git repository: https://github.com/BastienRousseau/DSL-MyCSV

We have written a "User Manual" for the language MyCsv 1.0. It explains the full language specification, and serve as a reference for expected behavior of any implementation of this language. We suppose that the reader of this present document has already read the User Manual (you may find it in the git repository).

During this project, we have implemented MyCsv three times. Section 2 present each of those implementations, and their specific features.

To attest the correction and the performance of those implementations, we made some unit tests and benchmarks. The result of those experimentations are presented in section 3.

Section 4 presents the features we have thought of to augment the language we made, but that we didn't implement because that's just a student project.

2 Implementations

This section present all implementations we made for the MyCsv language. We presents their successes and failures, strengths and weaknesses.

2.1 Mastering XTend

To master the XTend language, we started by writing a simple pretty-printer for MyCsv. We did it successfully within 4h, counting the time taken to discover the *very* useful keyword dispatch.

Later during the project we introduced some modification on the XText grammar (adding the command Rename field and the no header option for the Store command). Because we didn't go back on the pretty-printer, it is in the end incomplete.

As it was taken as a training, we didn't come back to it. We knew enough of XTend to now implement our language...

2.2 Compiler Python

The python compiler is the first implementation we have written, because Python was for us the most natural language for CSV manipulation (among the set of Python, bash, and XTend). We finished its initial implementation within 5 hours.

It was pretty straightforward to compile MyCsv into Python because of many facilities:

- precedence of operator (both algebraic and logical) is the same;
- even the syntax of the operator was identical ;
- the use of global variables helped in having no argument to give when compiling sub-part of the AST.

One interesting feature of the python compiler, is that it make the MyCsv user benefit from the arbitrary integer precision from the Python language.

2.3 Interpreter

The XTend interpreter is the second implementation we have coded. We finished its initial implementation within 5 hours.

To benefit from the Object-Oriented approach, we defined a `Csv` class, with all operations needed on it. The state of the execution is almost totally contained as an attribute of this `Csv` class in the interpreter object, naturally named `CurrentCsv`.

Facilities from the Java collections helped a lot in some part of the interpretation. For example, when indexing lines or fields, we want to get rid of redundancy. Getting this done thanks to Java Set was relatively easy.

From our three implementations, the XTend interpreter is the most robust to errors. Not only it raises errors when comparing strings to number for example (see the `Value` class in Csv.xtend), but it also give an explicit message when doing so (even if it is in the form of a Exception Stack Trace printing, it is far better than just resume without noticing).

On the contrary of the python implementation, we are back to the usual int limitation of 2^{32}.

2.4 Compiler Bash

Before writing the bash compiler, we spent time on writing lot of tests. Each test was still run on a single variant, and ground truth was check by hand. So we created our `MyCsvBenchmarkTest`, testing all variant together on several `MyCsv` inputs, and comparing the results obtained with each variant for a common (single) CSV input.

With such a benchmark ready, we faced the implementation of the compiler to bash. We spent much more hours on this one than on the two previous implementations combined.

Our strategy to have a notion of *current CSV* was to store the state of such a CSV *as a real CSV file*. The bash script obtained through our compiler takes advantage of a temporary directory -created at the beginning of script execution- to store sequentially the intermediate states of the *current CSV* on the disk. Regarding the *current header*, on top of having it present in the CSV file, we stored its information in an associative bash array, making it easier to get the index of a given field name for example.

To compute the logical and algebraic expressions, we used the `bc` program, and for relational expressions, we used `awk`. Combination of those two programs were used to compute aggregative operations.

The bash implementation is the weakest of the three. On the one hand there are many cases considered as an error by the `MyCsv` specification in which the scripts won't raise any error, and worse, will continue the execution. On the other hand, there are many cases in which this compiler fails whereas it shouldn't. Those failures are tightly linked with Bash very strict syntax: for example one of the limits of the Bash compiler is that the script produced don't support CSV input with spaces in strings. Another example is that bash script may fails if parenthesis appear in the CSV input.

Moreover, it is the worst implementation in term of execution time as we will see in section 3.2.2.

2.5 One .jar to run them all

To make our language easier to use and operate with, we created a `.jar` executable file. It gathers the three implementations of `MyCsv` we made, and uses options to select which variant we want to use.

To use it, use the following commands in a terminal:

- Bash compiler:

  ```
  java -jar <jarName> compile-bash /path/to/input.mycsv /path/to/output.sh
  ```

- Python compiler:

  ```
  java -jar <jarName> compile-python /path/to/input.mycsv /path/to/output.py
  ```

- Interpreter:

  ```
  java -jar <jarName> interpret /path/to/input.mycsv
  ```

The `.jar` is present at the root of the git repository.

You may obtain it yourself from the code source by editing the `.jar` for the project `org.xtext.myCsv.tests`, the main is in the `src` directory and is named `MyCsvMain.xtend`.

For further explanation, please refer to the `README.md` in the git repository.

3 Benchmarks and tests

3.1 Methods and material

There is a set of tests in `org.xtext.myCsv.tests/examples/`. In the directory `tests`, there are `MyCsv` programs, and in `csvFiles`, there are some CSV files used as input for `MyCsv`programs. The `MyCsv`test program are mostly unit tests, they focus on one specific functionality (we have tested each command one by one) and there is an *integration test* which use multiple commands. We used theses programs to benchmark the three implementations of our DSL.

For the correction part, we took advantage of *differential testing*, that is: we compared the output of the variants together. In fact, we didn't require strict equality of the outputs, since the variant may print in different way an floating point number for example. The equality we used was the structural equality as a CSV object: that is, we re-used our `Csv` class to test output comparison!

For the *printing* output of the programs, the `MyCsv` specification let unspecified the way it should be displayed. Because of this, we may neither establish a ground truth nor automate the differential testing. Therefore we compared the printing manually, by comparing them ourselves.

The same benchmark proceeds to time measurements. About the hardware environment: we ran tests with a Personal Computer, with a processor Intel Core i7 7th Generation and 8Go of RAM. For each test, we made five runs and computed the mean of times, to get more robust results. The benchmark ran during around 6 hours.

Executing conditions were not perfect, many other processes was running on the computer during the tests. We believe this explains the fluctuation we may see in the charts later on.

We didn't benchmark the memory usage, because we don't know how to get this information in Java or XTend in a meaningful way.

3.2 Results

3.2.1 Functional bugs

The use of differential testing has been very efficient in revealing functional bugs in our implementations.

Some were subtle wrong implementation of the specification, like the Python compiler forgetting to truncate too long list of values when inserting a new line, or storing with the default comma delimiter instead of the *current separator* when storing without the `sep` option ; and other bugs were just dumb errors such as evaluation of the `NbField` built-in expression returning the number of lines instead of columns in the interpreter.

Among other bugs we found, some were related with representation of floating point numbers. The `bc` program used to compute values in the bash compiled scripts doesn't write the initial 0 for numbers lower than 1 (.333 is it notation for 0.333), this has been problematic when exporting data, because importing doesn't support this format. Another bug related with floating point numbers is the interpreter import mechanism, which recognised numbers like 1.32 as Strings rather than floats. After telling the scanner to take US writing conventions, it worked as wanted.

Because the benchmark implementing the differential testing has been developed before the Bash compiler, we used it intensively for (almost) test-driven development. We were able to discover bash compiler failures just after having coded them.

We didn't keep track of each bug found thanks to this automated testing, but we found a big dozen of bugs this way.

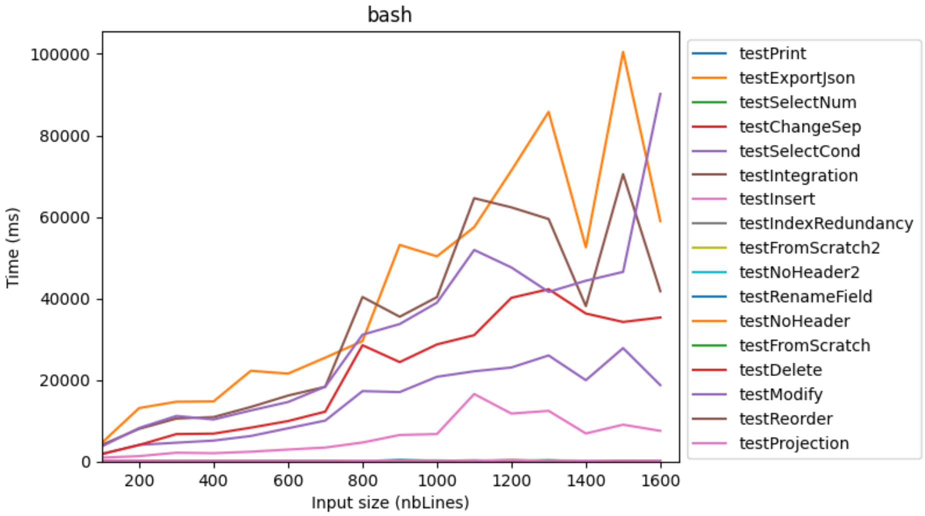

Figure 1: Bash script execution time

3.2.2 Performance

All numerical results are available in the git repository, as CSV file in `report/benchmark`. All plots are also available, in `report/benchmark/plot`.

It is important to compare the scale of time axis in Fig.1, Fig.2 and Fig.3. There is factor 10 between python and interpreter, and factor 50 between interpreter and bash. There is no doubt: bash implementation is the worst, regarding execution time.

While the CSV input grows, interpreter and bash seems to take both more time. We interpret it as our execution time being *linear* in the size of the CSV input. Surprisingly, referring to Fig.3 Python seems to not really depends on the input size (at least - for these entries). Maybe further test with much bigger CSV input would reveal Python script (supposed) linear execution time.

Sometimes, the interpreter curve is constant to 0. This happens because of the lack of precision of the time unit used (milliseconds). Measuring time with nanoseconds, we checked that this computation didn't break any fundamental physical law.

For the specific test `exportJson`, we may notice that the bash execution time overwhelms the others on Fig.4. That's why we also generate plot without bash (Fig. 5). The latter helps in revealing a difference of performance between the interpretation and the python script execution.

Finally, even if bash seems to be the worst implementation, for some commands, interpreter take more time, as shown in Fig.6. Such a difference reveals that the weakness of Bash scripts is more related with data analysis, that is when we explicitly traverse lines of data. The `Rename field` test corresponding to this chart is a very short operation in bash, whereas the interpreter always has the overhead cost of building the objects before working efficiently with them.

The test `FromScratch` on the other hand requires no initial (big) object building. Without this overhead cost, the interpreter has a much better performance as testified by Fig.7 (look closer: yellow line is on the x-axis).

3.3 Reproducibility

To reproduce benchmarks, please refer to README.md. Tests will produce two CSV files:

- `benchmarksMean.csv` which contains the mean of runs for each tests ;
- `benchmarksRuns.csv` which contains all run times.

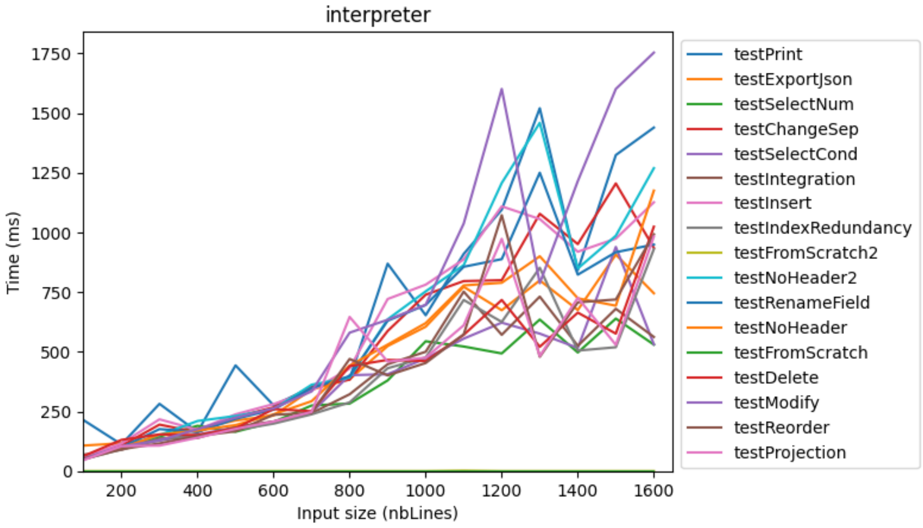

Figure 2: Interpret execution time

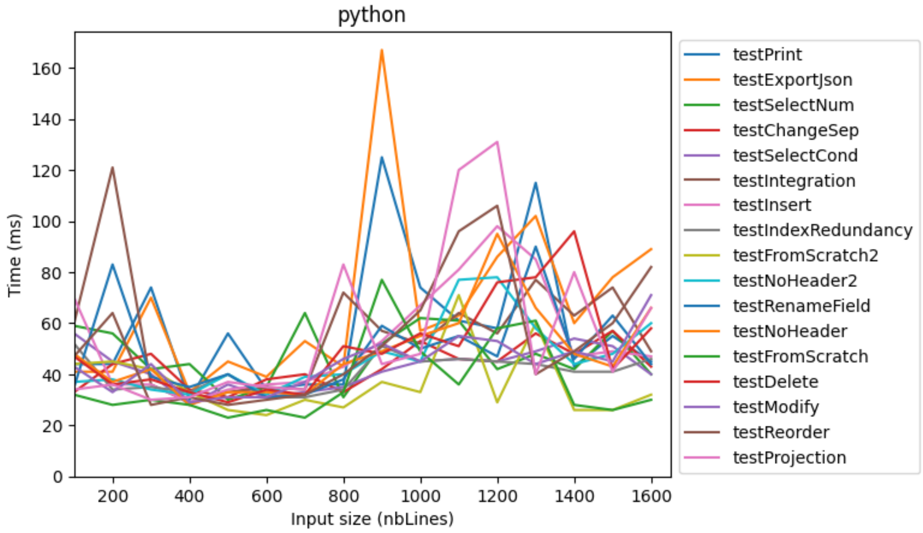

Figure 3: Python script execution time

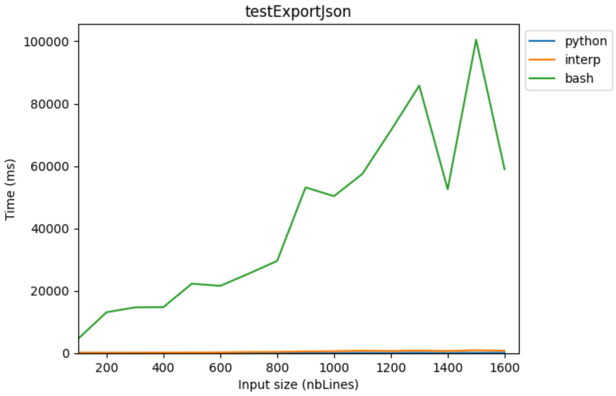

Figure 4: Comparison between each service for ExportJson

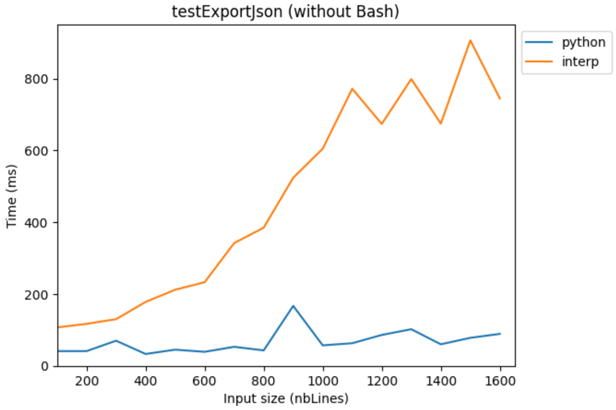

Figure 5: Comparison between each service for ExportJson (without Bash)

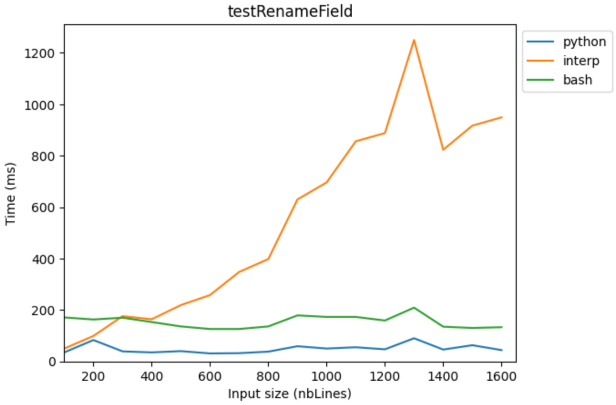

Figure 6: Comparison between each service for RenameField

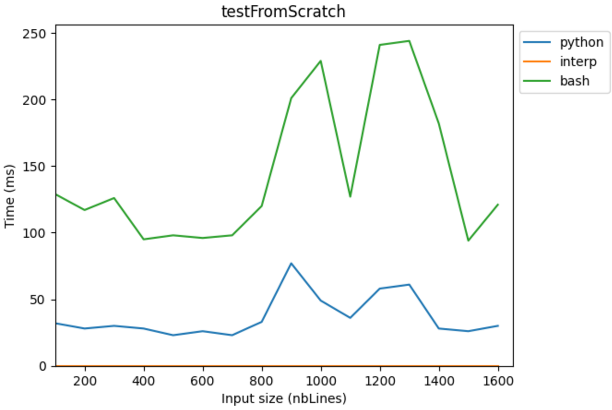

Figure 7: Comparison between each service for a CSV from Scratch

To generate plot, the git repository contains a python script.

3.4 Inter-operability

We tried our DSL in cooperation with the DSL of Lendy Mulot and Margot Masson, who worked on a DSL for ML classification. We biased their data with `MyCsv`, and they used them with their DSL. Results show weaker accuracy of the ML classification, which testify a successful bias introduced by our program.

4 MyCsv 2.0

While implementing our language, we had many ideas on how to augment it, but as we had to finish the project AND sleep, we didn't integrated them.

However just for their prosperity, and as a testimony of our creativity, we gathered them and list them down below, from the easiest to the most difficult to implement:

- An `All` implicit index, allowing to write the code: `Delete field all` to start writing a new CSV from scratch.
- New aggregative operation, like `Min`, `Max`, etc, to compute common other aggregative expressions.
- Allowing the insertion of new field without naming the header, auto-completing it with the same convention as in loading a file without header: `field`*n*
- Range indexing: allowing to index lines and fields with ranged to write something like `Select 1-50`.
- Built-in CSV, loadable with special path or syntax like `Load from <name>`, allowing a special "scratch" built-in CSV allowing to erase easily the current CSV, and maybe some Easter egg built-in CSV...
- A new `Graph` command, allowing to export graphs representing data in the current CSV in a easy way (but don't count on us to implement this feature in bash...).
- Allowing to handle multiple CSV at once, and adding the usual `join` operations from relational algebra.