Dexteris: Data Exploration and Transformation with a Guided Query Builder Approach

Sébastien Ferré

Dexteris: Data Exploration and Transformation with a Guided Query Builder Approach

Sébastien Ferré*

Univ Rennes, CNRS, Inria, IRISA
F-35000 Rennes, France
ferre@irisa.fr

Abstract. Data exploration and transformation remain a challenging prerequisite to the application of data analysis methods. The desired transformations are often ad-hoc so that existing end-user tools may not suffice, and plain programming may be necessary. We propose a guided query builder approach to reconcile expressivity and usability, i.e. to support the exploration of data, and the design of ad-hoc transformations, through data-user interaction only. This approach is available online as a client-side web application, named Dexteris. Its strengths and weaknesses are evaluated on a representative use case, and compared to plain programming and ChatGPT-assisted programming.

1 Introduction

In recent years, data has become ubiquitous and it is increasingly important for stakeholders to derive value from them. To achieve this objective, data analysis methods have been developed to help stakeholders gain insight into their data. However, it is almost always necessary to explore and transform data before applying data analysis methods. *Data exploration* is crucial to help data analysts understanding the data, and choosing the transformations to apply. *Data transformation* refers to the process of converting data from one format, structure, or type to another to meet the requirements of a particular use case or analysis. Examples of data transformation operations include filtering, aggregating, sorting, merging, pivoting, and applying mathematical or statistical functions to the data. In this paper, we focus on fine-grained data transformations, like converting between ad-hoc CSV and JSON files, and extracting or aggregating information from such files.

When choosing a tool for data exploration and transformation, data stakeholders are left with a trade-off between their expressive power and their usability. Expressive power is the range of questions and transformations that can be applied to the data. Usability is the degree of technical skills required to use it, as well as the level of guidance provided by the tool. At one end of the spectrum there is full-fledged programming, e.g. programming in the rich environment of

* This research is supported by the CominLabs project MiKroloG.

Python. This obviously offers the highest expressive power but this requires advanced programming skills and offers little guidance. At the other end of the spectrum there are end-user intuitive applications with a GUI (Graphical User Interface). For instance, spreadsheets are obviously more usable, as testified by their widespread usage. However, they are limited to tabular data, and computations are mostly cell-wise. In between there are ETL tools (Extract, Transform, Load), e.g. Talend, Pentaho. They offer high-level features for common data formats and common data transformations, e.g. merging data, removing duplicate data. However, in many cases, they require to write SQL queries to extract data from relational data tables, or JSON paths to extract data from JSON files, or other kinds of code. They thus require advanced technical skills, although to a lesser degree than full-fledged programming skills.

The `N<A>F` design pattern [4] has been shown to help reconciling expresivity and usability. It does so by relying on a formal language, and by bridging the gap between the end-user and the formal language with a *guided query builder* approach. Complex queries are incrementally and interactively built with data feedback and guidance at every building step. The design pattern has already been applied to the querying of SPARQL endpoints [5], the authoring of RDF descriptions and OWL ontologies [4], and data analytics on RDF graphs [6].

In this paper, we present the application of the `N<A>F` design pattern to the task of data exploration and transformation. We choose JSONiq [7] – the JSON query language – as the target formal language because it combines several advantages. It is a high-level declarative query language and yet a Turing-complete programming language; it lies on W3C standards; and although it uses JSON as a pivot data format, it is interoperable with other data formats such as text, CSV or XML. JSONiq is to JSON data what XQuery is to XML, and what SQL is to relational data. As a declarative and expressive language, it is a good fit for data exploration, data extraction, data transformation, and even data generation. The contributions of this work are:

1. a guided query builder approach to data exploration and transformation, based on the `N<A>F` design pattern, where arbitrary computations can be achieved, and only primitive inputs are required from end-users;
2. an online prototype called Dexteris[1], running as a client-side web application.

The paper is organized as follows. Section 2 motivates our approach with a concrete example. Section 3 discusses related work, and describes the `N<A>F` design pattern. Sections 4, 5, and 6 defines the three parts of `N<A>F` for data exploration and transformation: the intermediate language, the machine side, and the user interface. Section 7 evaluates the strengths and weaknesses of our approach, comparing it with plain programming and ChatGPT-assisted programming. Supplementary materials are available online[2] for the motivating example and evaluation use case (input and output files, Python programs and JSONiq queries, chat logs, and a Dexteris screencast).

[1] Freely available at `http://www.irisa.fr/LIS/ferre/dexteris/`
[2] `http://www.irisa.fr/LIS/ferre/pub/dexa2023/`

2 Motivating Example

As a motivating example, let us consider a scenario where the available input file is a CSV file describing projects by their ID, name, and members.

```
project id,project name,members
P1,Alpha,"Alice, Charlie"
P2,"Beta 2","Bob, Charlie"
```

The objective is to obtain a JSON file organized as a list of unique project members, describing each member by the list of projects she takes part in, and the number of such projects.

```
[ {"member": "Alice",
   "projects": [{"id": "P1", "name": "Alpha"}],
   "project number": 1},
  {"member": "Charlie",
   "projects": [{"id": "P1", "name": "Alpha"},
                {"id": "P2", "name": "Beta 2"}],
   "project number": 2},
  {"member": "Bob",
   "projects": [{"id": "P2", "name": "Beta 2"}],
   "project number": 1} ]
```

Any system that works on tabular data only will have a hard time generating the expected output data because the latter has a nested structure. Indeed, it is a list of objects that have a field ("projects") whose values are again lists of objects. The transformation from the available input to the expected output requires at least the following processing steps, in informal terms:

- reading the tabular data structure (CSV) in the input file;
- iterating over projects, i.e. over rows;
- splitting the lists of members, in the third column;
- grouping all (project, member) pairs by member;
- collecting all projects of each member;
- counting the number of projects per member;
- generating a JSON object for each member;
- collecting them and writing the whole in JSON format.

A concise Python program that performs the transformation is about 30 lines. Using JSONiq, we can make the transformation shorter (12 lines) and higher-level, in particular without assignments to mutable variables, hence without having to reason about computation states [10]. It relies on a JSON view of a CSV file, where each row is represented as a JSON object with CSV columns as fields.

```
for $row in collection("input.csv")
  let $project := {
      "id" : $row."project id"
```

```
    "name" : $row."project name" }
for $member in split($row."members", ", ")
  group by $member // $project is now a sequence of objects
  return {
      "member" : $member,
      "projects" : [ $project ],
      "project number" : count($project) }
```

Using the Dexteris tool that implements our approach, it is possible to build the above JSONiq program by starting from the input file, and then by applying suggested elementary transformations one after another. The only required inputs are elementary values: field names, new variable names, and the splitting separator. The number of required steps is 25, which includes 25 selections and 8 short inputs. Any part of a built transformation can be edited a posteriori. For instance, if one wants the JSON output to be in alphabetical order of members, only 2 additional steps are needed.

3 Related Work and Background

To cope with the difficulty to write data transformations in general-purpose programming languages, high-level data-oriented languages have been defined and even standardized. Notable examples are XQuery for XML data [14], JSONiq for JSON data [7] – which was strongly inspired by XQuery –, and formula languages that back graphical tools, such as the M language in Power BI [2]. Graphical tools like Power BI ambition to make all transformations doable in a graphical way but they recognize that *"there are some transformations that can't be done in the best way by using the graphical editor."* [9] The latter is also limited to tabular data, although it can cope with varied formats. The closest work to ours is probably the educative platform Scratch by MIT [12], which enables users to build arbitrarily complex programs in a purely graphical way, by assembling blocks with syntax-based shapes. However, its programming language is not appropriate for data transformations. Another related domain is *program synthesis* [8], which ambitions to generate programs solely by providing examples of input-output pairs. For instance, given strings like *"Dr. Helen Smith (1999)"*, it can learn to output strings like *"Smith H."* from a few examples. It is however yet too limited in the size of the examples and in the complexity of generated programs to be largely applicable to data transformations. Moreover, some transformations are one-off and therefore producing an example implies producing the expected output. It is also sometimes simpler to specify the transformation in an intentional way rather than by providing examples.

The purpose of the `N<A>F` design pattern [4] is to bridge the gap between an end user speaking a natural language (NL) and a machine understanding a formal language (FL), as summarized in Figure 1. The design pattern has for instance been instantiated to the task of semantic search with SPARQL as the formal language [5]. The central element of the bridge is made of the Abstract

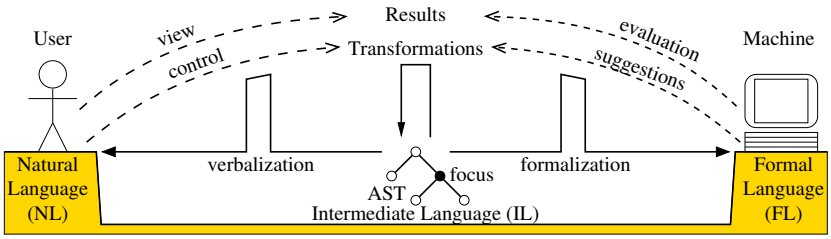

Fig. 1. Principle of the `N<A>F` design pattern

Syntax Trees (AST) of an Intermediate Language (IL), which is designed to make translations from ASTs to both NL (*verbalization*) and FL (*formalization*) as simple as possible. IL may not have any proper concrete syntax, NL and FL playing this role respectively for the user and for the machine.

`N<A>F` follows the query builder approach, where the structure that is incrementally built is precisely an AST. Unlike other query builders, the generated query (FL) and the displayed query (NL) may strongly differ in their structure thanks to the mediation of IL. The AST is initially the simplest query, and is incrementally built by applying *transformations* (not to be confused with the data transformations this paper is about). A transformation may insert or delete a query element at the *focus*. The *focus* is a distinguished node of the AST that the user can freely move to control which parts of the query should be modified. Results come from the evaluation of the formalized query, and are viewed by the user. Transformations are suggested by the machine based on the formalized query and actual data, and controlled by the user. Both results and transformations are verbalized in NL for display to the user. At each step, the user interface shows: (a) the verbalization of the current query with the focus highlighted, (b) the query results, and (c) the suggested transformations.

4 Intermediate Language and Transformations

Given that the target formal language (FL) JSONiq is already high-level and declarative, we also use it as the Intermediate Language (IL). As shown in the next section, this does not make the formalization step from IL to FL void, and it therefore still makes sense to distinguish between IL and FL. Indeed, a key feature of IL is the notion of *focus* that impacts both the results and the suggested query transformations. In particular, the position of the focus modifies the semantics of the query in order to show the internals of the computations performed by the query at the focus point.

In this section, we first define the FL/IL as a large subset of JSONiq plus a few extensions of our own. We then introduce the list of elementary query transformations such that all queries can be built through finite sequences of such transformations.

Table 1. JSONiq constructs: expressions, FLWOR clauses, and syntactic sugar.

Expression constructs (*expr*)	Semantics
VAR	variable (prefixed by `$`)
JSON	JSON value
`{` *expr* `:` *expr* `, ...}`	object construction
`{|` *expr* `|}`	object sequence to object (merge)
`[` *expr* `]`	sequence to array
expr `[]`	array to sequence
FUNC `(` *expr* `, ...)`	function calls and operators
`()`	the empty sequence
expr `,` *expr*	sequence concatenation
expr `.` *expr*	object lookup by field
expr `[[` *expr* `]]`	array lookup by index
`if` *expr* `then` *expr* `else` *expr*	conditional expression
flwor ↩ ... `return` *expr*	expression nested in FLWOR clauses
FLWOR clauses (*flwor*)	Semantics
`let` VAR `:=` *expr*	binding a new variable
`def` FUNC `(`VAR `, ...) =` *expr*	defining a new function
`for` VAR `in` *expr*	iterating on a sequence
`where` *expr*	filtering an iteration
`order by` *expr* `[` `asc` | `desc` `] , ...`	ordering an iteration
`group by` VAR `, ...`	grouping and concatenating
`count` VAR	variable for the position in iteration
Syntactic sugar	
$expr_1$ `!` $expr_2$ = `for $$ in` $expr_1$ ↩ `return` $expr_2$	
$expr_1$ `[` $expr_2$ `]` = `for $$ in` $expr_1$ ↩ `where` $expr_2$ ↩ `return $$`	

4.1 A Language based on JSONiq

Table 1 lists the constructs of JSONiq that are used in our IL (symbol ↩ represents a carriage return). They are presented in concrete syntax for readability and for consistency with the standard JSONiq syntax. However, they are used under abstract syntax only for IL, and verbalized in a slightly different way to make them more intuitive to end users (see Section 6).

The table gives the semantics of each construct in an informal way. An original aspect of JSONiq is that values are *sequences* of *items*, where items are JSON values. For recall, JSON values are one of: strings delimited by double quotes, numbers, Boolean values (`true` and `false`), arrays delimited by square brackets, objects with named fields and delimited by curly braces, and the `null` value. Array members and object field values can be arbitrarily nested JSON values. Another original aspect of JSONiq expressions is the FLWOR clauses (pronounce "flower") that help working with sequences in a declarative way. They are inherited from XQuery [14], and they are analogous to clauses found in SQL and SPARQL. FLWOR is an acronym for the constructs: `for`, `let`, `where`, `order by`, and `return`. The combination of `for` and `where` clauses can express joins like in relational databases. The combination of `for`, `group by`, and aggrega-

tion functions – i.e. functions from sequences to items – can express analytical queries, similar to OLAP cubes [3].

An important ingredient of the semantics and evaluation of JSONiq expressions is the *environment*. It defines for each sub-expression the set of variables that are in scope. Variables are added to the environment by the FLWOR clauses `let`, `for`, and `count`, and are in the scope of the FLWOR clauses coming after, until the expression after `return`. An environment maps each variable in scope to its value, a sequence of JSON values.

Our language has a few differences with JSONiq. First, it currently misses a few constructs, left for future work, namely: anonymous functions and partial function application, switch and try-catch expressions, and type-related expressions (e.g., `instance-of`). Second, it adds two convenient FLWOR clauses to explode JSON objects in as many variable bindings are there are object fields:

- `let * :=` *expr* assumes that the expression returns an object, and is then equivalent to have a let-binding for each object field, making them directly available as local variables;
- `for * in` *expr* is equivalent to `for $$ in` *expr* $\hookleftarrow$ `let * := $$`, thus combining iteration on objects, and exploding them into let-bindings.

Here is the query that defines the expected data transformation in the motivating example (Section 2), in our JSONiq-based language.

```
printJSON(
    for * in parseCSV(file <example_input.csv>)
      let $project := {
          "id" : $(project id),
          "name" : $(project name) }
      for $member in split($members, ", ")
        group by $member
        return {
            "member" : $member,
            "projects" : [ $project ],
            "project number" : count($project) })
```

The expression `file <example_input.csv>` evaluates to the raw contents of the input file, a string. A query can use any number of files. Function `parseCSV` turns this raw string into a JSON representation of the tabular data, i.e. a sequence of JSON objects, each object representing a CSV row with column headers as object fields. The `for *` clause iterates over the CSV rows, and bind each column as a variable (e.g., `$(project id)`). After grouping by member, variable `$project` maps to a sequence of projects, all projects related to the current member. This can be seen as a default aggregation, from which any other aggregation can be computed. For instance, `[ $project ]` aggregates the sequence of projects as a JSON array, and `count($project)` counts the number of projects. Finally, function `printJSON` turns the JSON result, a sequence of JSON objects, into a raw string in JSON format, ready for writing into a file.

4.2 Query Focus and Query Transformations

A *query focus* splits a query into the sub-expression at focus, and the *context* of that sub-expression. For instance, in the expression `split($row.members, ", ")`, if the focus is on the sub-expression `$row.members` then the context is [`split(•, ", ")`], where • (called *hole*) locates the focus position. If the focus is on `$row` then the context is [`split(•.members, ", ")`], which can be seen as the nesting of two elementary contexts: [`•.members`] and [`split(•, ", ")`].

A *query transformation* modifies the expression around the focus or moves the focus. The empty sequence `()` serves as the initial query, and also to fill in new sub-expressions introduced along with constructs. A transformation belongs to one of the following kinds, beside focus moves.

- An *elementary expression* that replaces the sub-expression at focus: a scalar value (e.g. `"id"`), a variable (e.g. `$member`), etc.
- An *elementary context* that is inserted between the sub-expression at focus and its context: e.g., [`•, ()`], [`(), •`], [`for * in •`]. The hole is located at one sub-expression, and other sub-expressions are initialized to `()`.
- The *addition* of an element other than a sub-expression, for which there is no focus: e.g., adding a field to an object, adding a variable to a `group by` clause, adding a parameter to a defined function.
- The *deletion* of the sub-expression at focus, i.e. its replacement by `()`, or the *deletion* of the elementary context at focus.

Some transformations have editable parts, which have to be filled in by the user. This is the case for scalar values, for field names, and for the name of new variables introduced by binding constructs. The query in the above section can be built through the following sequence of transformations (25 steps separated by /, `up` is for moving the focus up in the AST).

```
file <example_input.csv> / parseCSV(•) / for * in • /
{"id": ()} / $(project id) / "name": () / $(project name) /
up / let $project := • / $members / split(•, ()) /
", " / up / for $member in • / group by $member /
{"member": ()} / $member / "projects": () / $project / [ • ] /
"project number": () / $project / count(•) / up[6] / printJSON(•)
```

It can be proved by induction that all expressions can be built in a finite number of steps, proportional to the syntactic size of the expression. The next section explains how the user receives data feedback and guidance at every step, so that what here seems like a purely syntactic process is actually a data-centered and guided incremental process.

5 Formalization and Suggestions (Machine Side)

The formalization process determines the evaluation to be actually performed, and hence the results to be displayed to the user. An important point is that it depends on the focus position. To motivate and illustrate this dependency, suppose we have the following expression

```
for $i in 1 to 3
  return 1 to $i
```

where the operator a `to` b evaluates to the sequence a`,` $a+1$`,` $\ldots$`,` b. The result of the whole expression is the sequence of integers `1, 1, 2, 1, 2, 3`. Now, suppose that the focus is on the sub-expression after `return`, then the expected result is the value of that sub-expression `1 to $i`. However, this value depends on variable `$i`, which is introduced in the focus context. Therefore, a useful result is a mapping from each value of `$i` to the value at focus, bound to an implicit variable `$focus`.

`$i`	`$focus`
1	1
2	1, 2
3	1, 2, 3

Each row of such a table is actually an environment, i.e. a mapping from variables in the focus scope to their values.

5.1 Formalization by Expression Rewriting

Formalization is performed by rewriting the query expression and focus position into a new expression whose evaluation results in a sequence of environments. Let the query be the expression $e = C(f) = c_k(\ldots c_1(f)\ldots)$, where f is the sub-expression at focus, C is the context of the focus. The context C can be decomposed into a series of elementary contexts c_i that need to be applied to the sub-expression f bottom-up in order to get the whole expression e. In the above example, $f = [$`1 to $i`$]$, $k = 2$, $c_1 = [$`return` $\bullet]$, and $c_2 = [$`for $i in 1 to 3` $\hookleftarrow \bullet]$.

The rewriting process starts by initializing the rewritten expression e' as

$e' :=$ `let $focus :=` $f \hookleftarrow$ `return $env`

where variable `$focus` is bound to the sub-expression at focus, and the environment is returned through a special variable `$env`. Then each elementary context is processed bottom up, from c_1 to c_k.

- FLWOR clauses are applied unmodified, so that iterations, bindings, filtering, grouping and ordering are kept in the focus-dependent evaluation. They determine the rows of the table of results.
- Conditional expressions with the hole in one of the branches are simplified by replacing the other branch by the empty sequence. For instance, context $[$`if` e_1 `then` $\bullet$ `else` $e_3]$ applied to the rewritten expression leads to $e' :=$ `if` e_1 `then` e' `else ()`.
- All other contexts are ignored, and hence excluded from the rewritten expression. For instance, ignoring context *func*`(`e_1`,`$\bullet$`)` enables to focus on the second argument of the function, temporarily ignoring the first argument and the function application.

The rewritten expression is therefore a chain of FLWOR clauses (and conditionals) ending with the return of an environment. The evaluation result is therefore a sequence of environments. Given that the environments bind the same set of variables, the result can be presented as tabular data, with one row for each environment – each iteration step –, and a column for each variable. The last column is the `$focus` variable, which plays a central role for computing the suggested query transformations. Note that although the view on results is tabular, the data can be arbitrarily nested JSON data. The table of results contains JSON values, and its shape automatically adapts to the current query and focus.

5.2 Computation of Suggestions

In the `N<A>F` design pattern, the set of suggestions is the subset of query transformations that are well-defined and relevant given the current query, the current focus, and the results of the query formalization. First, a static analysis of the current query and focus is performed in order to identify variables and functions in scope and to derive type constraints. The considered types are the JSON types: numbers, strings, booleans, arrays, and objects. For instance, the focus context [• . e_2] calls for JSON objects, while the focus context [e_1 . •] calls for strings (field name). Also, if the focus sub-expression is a comparison, then its type is boolean, which suggests to insert contexts such as [`if • then () else ()`] (conditional expression) or [`where •`] (filtering).

Second, a dynamic analysis of the results is performed in order to identify which data types are available at focus. For instance, the presence of numbers suggests to apply arithmetic operators; and the presence of arrays suggests to apply a lookup-by-index operator, for instance. We also collect the fields defined in objects, in order to suggest the lookup-by-field operator with pre-defined field names. The dynamic analysis also looks at the number of rows, and at the sequence lengths of focus values. The former conditions the insertion of FLWOR clauses, which are only relevant when there are multiple rows, i.e. in the scope of a `for` clause. The latter conditions the insertion of iterations and aggregations, which are only relevant with non-singleton sequences.

For the sake of efficiency, dynamic analysis is only performed on a sample evaluation of the results, bounding the number of rows, and bounding the number of computed items per sequence. This relies on a lazy evaluation of expressions.

6 Verbalization and Control (User Interface)

Verbalization of the query and results, and control of the suggested transformations characterize the user interface, and hence the user experience. Figure 2 shows a screenshot of the Dexteris tool. The query and focus can be seen at the top left. The results can be seen in the table at the bottom. The suggested transformations can be seen in the three lists at the top right: functions and

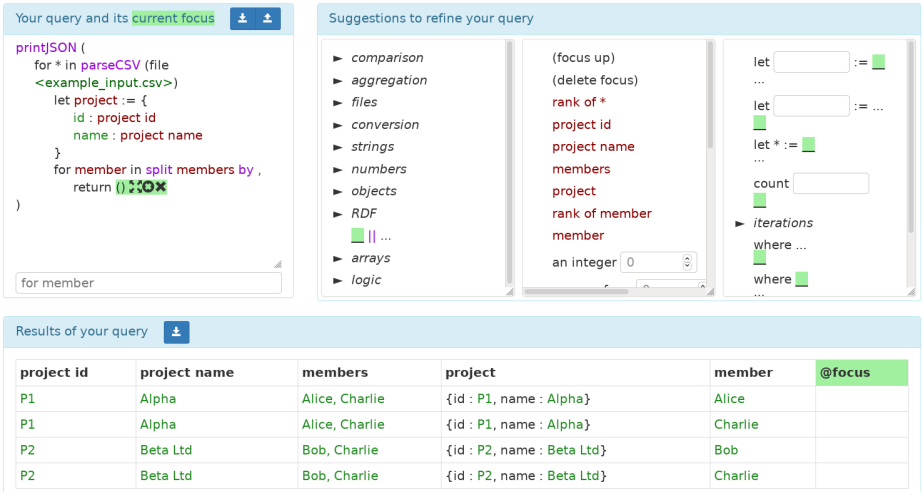

Fig. 2. Screenshot of Dexteris in the course of building the example query.

operators (left), variables and JSON value constructors and accessors (middle), FLWOR clauses (right).

The verbalization remains close to the original concrete syntax of JSONiq, as given in Table 1, and hence it is less natural than in previous `N<A>F` applications. To justify this, note that verbalizing the expression e_1 `[[` e_2 `]]` as *"the e_2-th member of e_1"*, or the expression `{ "id" : $(project id), "name" : $(project name) }` as *"an object whose id is the project id, and whose name is the project name"*, is closer to natural language but it does not make it easier to read. By the way, JSONiq already uses explicit keywords like `where` or `if`, and most special characters correspond to JSON notations (e.g., square brackets and curly braces). We apply the following changes to the concrete syntax of JSONiq in order to lift its more unnatural aspects.

- The dollar sign in front of variables and the double quotes surrounding strings are removed. Colors are used to distinguish numbers (dark blue), strings (dark green), booleans (green/red), variables (dark red), and function names (purple). In particular, this avoids the need for escaping characters in strings.
- The implicit `$$` variable in syntactic sugar is renamed as `this`.
- A mixfix syntax is used for some functions, which includes the usual infix notation for arithmetic and logical functions: e.g., [e_1 `+` e_2], [`not` e_1], [`split` e_1 `by` e_2]. This makes the role of the different function arguments more explicit and readable.
- The semi-colon `;` is used as the sequence separator to avoid confusion with other usages of the comma (arrays, objects, function arguments).

Moreover, as queries are built by applying query transformations rather than edited as text, there is no issue with operator priorities and other ambiguities

```
[   {   "id": "9ace9041",
        "question": "What is the fourth book in the Twilight series?",
        "translations": {"fr": "Quel est le quatrième livre de la série Twilight ?", ...}
        "questionEntity": [
            {   "name": "Q44523",
                "entityType": "entity",
                "label": "Twilight", ...}, ...],
        "answer": {
            "answerType": "entity",
            "answer": [ { "name": "Q53945", "label": "Breaking Dawn" } ],
            "mention": "Breaking Dawn" },
        "category": "books",
        "complexityType": "ordinal"
    }, ... ]

ID,Type,Question,Entities,Answer
9ace9041,ordinal,What is the fourth book in the Twilight series?,Q44523: Twilight,Q53945: Breaking Dawn
...
```

Fig. 3. Excerpt of the input file (top), a JSON list of Mintaka question descriptions and of the expected output file (bottom), a CSV file with one row per question.

so that grouping brackets become useless. Those groupings are made visible by indentation and through focus moves because the sub-expression at focus is highlighted (in light green, see Figure 2).

Suggestions are controlled simply by clicking them. When a suggestion has input widgets, they may be filled beforehand. Those inputs are for primitive values in the middle list, and for variable/function names in the right column. The focus can be moved up with one of the suggestions, and in all directions with key strokes (Ctrl+arrows). Any focus can also be selected directly by clicking on it in the query area. For advanced users, the text input below the query provides a command line interface for quickly inserting data values, and applying query transformations. When a suggestion is clicked, its command is displayed in the text input as hint in order to help the user learning them.

7 Evaluation

Because data transformation tasks are often ad-hoc and incompletely specified, it is difficult to conduct a systematic evaluation similar to what is done for fully automated approaches, e.g. supervised classification tasks. Moreover, we do not know of tools comparable to Dexteris, tools that would support the design of almost arbitrarily complex transformations without requiring the user to write some code at some point. We therefore choose to report on a representative use case that was encountered in a real setting. It involves two file formats (JSON and CSV), and many features of the JSONiq language. It is relatively simple to informally describe while being non-trivial to implement. We compare the user experience and result with plain programming and using ChatGPT [11], which is known for its capability to generate code from textual prompts in a versatile and multi-turn way.

Task. The objective is to transform the JSON files of the Mintaka dataset [13] by extracting some information and formatting it into CSV files. The end goal was to prepare a training set for question answering [1]. A JSON file is a list of questions, where each question is described by an ID, the question in English and other languages, Wikidata entities, answers, and the complexity type of the question (e.g., ordinal, count). Figure 3 shows an excerpt of a JSON file. It has up to 5 levels of nested lists and objects. It also features some heterogeneity in the representation of entities and answers, depending on their type (e.g., Wikidata entities, numbers). The output CSV file should have 5 columns: ID, Type, Question, Entities, and Answer. The three first columns directly correspond to fields of the question objects. However, the last two columns are string aggregations of respectively the list of entities, and the list of answers. Moreover, Wikidata entities should be formatted so as to combine their name and their label, while only the name should be used for literal values. Finally, the rows should be sorted by question type. The task therefore involves nested iterations - on questions, on question's entities, and on question's answers, navigation in JSON objects, string building and aggregation, conditional expressions, and ordering.

Plain programming. We described the task to four experienced Python programmers (students in our lab), provided two example questions, and asked them to write a program for the desired transformation. Two of them overlooked missing fields in some questions so that their program failed on the whole input file. However, they could quickly fix their program for those exceptional cases. The total time they needed to complete the task was consistent, between 30 and 45 minutes. Their programs were between 36 and 68 lines of code. They declared that they would need about 15min to rewrite their program from scratch.

Programming with ChatGPT. On a first attempt[3], we gave it a representative excerpt of the JSON file (2 questions of different types), and the expected output, then we asked it to generate a Python program to do the translation. After 15min and 3 turns, we stopped because it had a very shallow understanding of the task, and when prompted to look better at the data, it started to hallucinate code irrelevant to the task. On a second attempt, we precisely described in text the structure of the input file, and the structure and contents of the output file. On each turn, we tested the generated Python program, sometimes correcting it for obvious small errors (e.g., field names). After 40min and 8 turns of rather constructive chat, we obtained an almost correct program but then it started to diverge on the last remaining bug related to the missing fields. It should be noted that our successive prompts were strongly based on reading the generated code because that code could not be run, and so errors could not be reported in terms of errors in the generated data.

Using Dexteris. After opening the input file and parsing it as JSON data, the results in Dexteris shows a sequence of JSON objects. An obvious step is therefore to iterate over them, the suggestion `for *` is selected in order to expose

[3] The chat logs can be found at `http://www.irisa.fr/LIS/ferre/pub/dexa2023/`.

each object field as a result column. From there, a JSON object is created to define a CSV row, with one field for each expected column. The three first fields (columns) are simply defined by picking the right variable among the exposed object fields (e.g. `$id`). For entities, the user starts from the value of field `questionEntity`, iterates over them as this is a list, then builds a string by concatenating two fields of each entity, with a colon in between. The aggregation `concat_with_separator` can then be applied to the sequence of formatted entities in order to have one string value for the CSV column "Entities". A similar process is applied to the answers, this time using the mapping construct `!`, and a conditional depending on the answer type. Finally, the ordering construct is inserted, and the function `printCSV` is applied to the whole in order to convert the generated JSON objects into CSV rows. A screencast of the whole building sequence is available on the supplementary material page.

The whole query can be built in 49 steps, including focus changes. When the need for the transformation arose, it took us less than 30min to build the query and output the transformed data, including the exploration of data, and thinking about what to generate exactly in the output file. Indeed, in real situation, the task is often incompletely specified, and gets refined when exposed to the actual data and unexpected cases. Building again the query in a straight way, knowing exactly what to do, takes 5 minutes, hence a building speed of about 10 steps/min. For recall, the Python programmers declared that they would need 15min to recode their program from scratch.

Strengths and weaknesses. Comparing the user experience between Dexteris, plain programming, and ChatGPT reveals the strengths and weaknesses of our approach. The main strengths of our approach are:

- *safeness*: the program (the JSONiq query) is valid at all time, there are no issues with syntax errors or runtime errors;
- *program introspection*: every part of the program can be introspected (and modified) by moving the focus around, like consulting values extracted from the inputs, verifying some sub-computation, or unfolding an iteration;
- *data-centric view*: no need to go forth and back between the data and the program, no need to switch between the programming language and textual prompts, the data are right there and determine what program constructions can be inserted, e.g. a JSON list suggests to insert an iteration, only the variables that are in scope and of the right type are suggested;
- *robustness*: peculiar cases in the input data are smoothly handled whereas they trigger runtime errors in Python programs, human- or machine-generated, e.g. an empty sequence is generated in case of a missing field.

On the weakness side, our approach is not immediately usable, unlike ChatGPT, although it provides more control and does not expose the user to a general programming language. It is also less versatile and scalable than plain programming.

Efficiency. Dexteris is a prototype, and it runs entirely in the browser as a client-side application. Its efficiency and scalability are therefore limited. However, the

Mintaka use case demonstrates that it is efficient enough to cope with many practical use cases that data stakeholders encounter. The smallest Mintaka file is 3.8M, and the 280kB output is generated in about 1.5s in Firefox 88.0.1 on Fedora 32 with an Intel Core i7x12 and 16GB RAM. The largest file is 26.7MB and the 1.9MB output (a CSV with 14k rows) is generated in about 15s.

8 Conclusion and Perspectives

We have defined and implemented Dexteris, a tool for data exploration and transformation based on the `N<A>F` design pattern. It allows the end-user to define complex data transformations in a data-centric way, without having to write any piece of code. Compared to plain programming, or ChatGPT-assisted programming, it features a safer and more robust process. In the future, Dexteris will be improved by covering the missing JSONiq constructs, and extending the set of functions and supported data formats.

References

1. Affolter, K., Stockinger, K., Bernstein, A.: A comparative survey of recent natural language interfaces for databases. The VLDB Journal **28**, 793–819 (2019)
2. Becker, L.T., Gould, E.M.: Microsoft power bi: extending excel to manipulate, analyze, and visualize diverse data. Serials Review **45**(3), 184–188 (2019)
3. Codd, E., Codd, S., Salley, C.: Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate. Codd & Date, Inc, San Jose (1993)
4. Ferré, S.: Bridging the gap between formal languages and natural languages with zippers. In: Sack, H., et al. (eds.) Extended Semantic Web Conf. (ESWC). pp. 269–284. Springer (2016)
5. Ferré, S.: Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. Semantic Web: Interoperability, Usability, Applicability **8**(3), 405–418 (2017), `http://www.irisa.fr/LIS/ferre/sparklis/`
6. Ferré, S.: Analytical queries on vanilla RDF graphs with a guided query builder approach. In: Andreasen, T., et al. (eds.) Flexible Query Answering Systems. pp. 41–53. LNCS 12871, Springer (2021)
7. Florescu, D., Fourny, G.: JSONiq: The history of a query language. IEEE internet computing **17**(5), 86–90 (2013)
8. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Symp. Principles of Programming Languages. pp. 317–330. ACM (2011)
9. Microsoft: PowerQuery, `https://learn.microsoft.com/en-us/power-query/`
10. Moseley, B., Marks, P.: Out of the tar pit. Software Practice Advancement (2006)
11. OpenAI: ChatGPT, `http://chat.openai.com`
12. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., et al.: Scratch: programming for all. Communications of the ACM **52**(11), 60–67 (2009)
13. Sen, P., Aji, A.F., Saffari, A.: Mintaka: A complex, natural, and multilingual dataset for end-to-end question answering. In: Int. Conf. Computational Linguistics. pp. 1604–1619. Int. Committee Computational Linguistics (2022)
14. XQuery 3.0: An XML query language (2013), `http://www.w3.org/TR/xquery-30/`, `http://www.w3.org/TR/xquery-30/`, W3C Proposed Recommendation