Branching Compositional Data Transformations in jq, Visually

Michael Homer
mwh@ecs.vuw.ac.nz
Victoria University of Wellington
New Zealand

Abstract

jq is a widely-used command-line tool for filtering and transforming JSON data, in the vein of sed and awk, including a bespoke programming language for writing the filters. The paradigm of that language is unusual: while its appearance is somewhere between a shell pipeline and JavaScript, the language is pure-functional and essentially concatenative, the pipelines branch and interleave data invisibly, implicit output flattening obscures all these effects, and most users are unaware of any of its semantics, leading to confusion when encountering any of these latent elements and difficulty in constructing appropriate non-trivial filters, all the while common debugging techniques are also obstructed by the paradigm. These confusions can be eliminated by visually demonstrating the recursively forking nature of evaluation on actual data, and allowing manipulations of the program or data to be reflected live, but no such tool exists for jq or any similar approaches. We present a visualisation of jq's execution model that makes the branching nature manifest, illustrating its effects on concrete values provided by the user, and editing affordances that allow manipulating the program with reference to real data.

CCS Concepts: • **Software and its engineering** → *Visual languages*; *Functional languages*; *Data flow languages.*

Keywords: jq, dataflow programming, object pipelines, stream processing

ACM Reference Format:
Michael Homer. 2023. Branching Compositional Data Transformations in jq, Visually. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3623504.3623567

PAINT '23, October 23, 2023, Cascais, Portugal

1 Introduction

jq [12] is a command-line tool for filtering and transforming JSON data, and is widely-used in shell scripting, automation, and CI/CD contexts. The jq tool is a Unix-style filter, reading JSON data from standard input and writing JSON data to standard output. Within the tool is a programming language for writing transformation filters over JSON values. This language has unusual semantics, which are not immediately obvious from the syntax, and which are not well-understood by most users. Elements of the language and tooling also make conventional debugging and understanding operations challenging, even those typical for comparable Unix filters like sed and awk, or in other stream-based dataflow systems.

The language combines elements of functional programming, concatenative programming, object pipelines, and stream filters, while the distributive branching of the evaluation model has more in common with the List monad than any comparable tool. Many of these elements are hidden behind a syntax aiming to be familiar to the intersection of Unix shell users and JavaScript programmers; its own developers acknowledge that most users do not realise there is a functional language inside. Only when encountering unexpected behaviour, or trying to accomplish more complex tasks, do users discover the more intricate inner model.

This paper presents an environment for exploratory visualisation and composition of jq programs, aiming to expose the semantics explicitly to the user. The intention is that, by illustrating the intermediate effects of the program on concrete data, the user can better understand the behaviour of the program on their data, and can more easily make changes to the program to achieve the desired effect. For example, a single input value may result in multiple outputs from a single filter, and each of these represents a *separate branch* of the remainder of the program, scoped to that value's context. The conventional jq tool flattens all outputs into a single textual stream of JSON values, concealing this scoped branching, but in our environment the derivation of all values is visible in the displayed grid of values.

The contributions of this paper are:

- An analysis of the semantics of the jq language and evaluation model identifying its atypical elements.
- A user interface for examining and editing a jq program that makes these semantics manifest, showing all the implicit branching, repetition, and intermediate

phases, and allowing changes to the program to be made directly on the data with live feedback.
- A parallel reimplementation of the behaviour of the jq language in an explicative fashion, with tracing and partial evaluation, to support structured analysis of the language.

2 The jq Language and Semantics

jq provides a quasi-concatenative nondeterministic object pipeline, a combination of relatively-uncommon features, presented in a syntax and context very close in aspects to both JavaScript and Unix shell scripting. A brief introduction to the core points of the language follows.

The typical invocation of jq is in the form of a Unix pipeline `curl . . . | jq '.result.points[]|select(.y>0)'`. The standard input (here represented as produced by the `curl` command) is expected to be a JSON document or a sequence of them. The single argument is a program in the jq language, which is our main focus; these programs contain pipe characters inside, not connected to the Unix pipes outside.

The program is a sequence of *filters*, which are composed together to produce an eventual JSON value or values (or nothing). Each filter is applied to an input object provided from the previous step in the pipeline, or from the original input. Filters include the object-index selector `.name`, which selects the value of the named key from the object; identity `.`, which produces its input unchanged; array indexing with `.[index]`; and function calls like `tostring` and `select(condition)`. Filters can be composed by concatenation: `.result.points[3]` composes three filters `.result`, `.points`, and `[3]`, to produce the third element of the array called "points" inside the object called "result" within the original context. They can also be composed by piping from one to another: `.result|tostring` gives the output of `.result` to the function `tostring`. The language is thus almost concatenative and certainly compositional [11] in its program construction. Some compositions of filters can be used with either syntax, and some only with pipes; some more complex operations are also possible, discussed later.

Every filter receives one JSON value as input, either from standard input at the start, or from the filter to its left. The filter produces *zero or more* JSON values as output, and each is given one at a time to the next filter in the sequence. Filters producing multiple outputs include the array/object value iterator `item[]`, which produces each value of an array or object separately, and the comma operator, which produces all of the outputs of each of its arguments in turn. When multiple outputs are produced, the pipeline splits into separate branches for each value: the next filter, and all subsequent filters, are applied starting with each value individually. For example, `.x[].y` produces the value of the `y` key from each value of the `x` key of the input object. Some filters can produce no outputs, such as the `select` function or the iterator filter on an empty array; in these cases, the remainder of the pipeline in this branch does not run.

At the end, jq outputs all JSON values produced at the end of the pipeline, in one flat sequence.

Already there are some complexities apparent in the description: McCarthy nondeterminism [14] is pervasive at every step, a linear-looking pipeline is secretly branching, and concatenation is composition. The sample programs above are ones that most jq users could produce and understand, although some parts of the evaluation may not (need to) be understood precisely. More complex programs uncover further branching, and more advanced features of the language expose other elements that are often unexpected.

For example, the `select` function takes a conditional expression as its argument: `.points[]|select(.y>0)` produces all the values within the `points` array where the value in `y` is greater than zero. However, the condition argument could *also* produce multiple outputs, and the `select` function will emit its input value once for each true value in those outputs: `.[]|select(.[]>0)` expects to consume an array of arrays, and produces each inner array as many times as it contains a positive number.

Variable bindings `. . . | .y as $y | . . .` give a name to a value derived from the input which can be used anywhere to its right, while passing on the input value itself unchanged. These expose the branching of the pipeline directly, while appearing out-of-place within the Unix-style pipeline.

Array literal expressions exist: `[1,2,3]` produces an array with values 1, 2, and 3 — but in fact they are not literals, but a circumfix `[ ... ]` operator that captures all the values produced by the enclosed pipeline branches. This "literal" is actually using the comma operator to produce three number values, while `[ .x[], .y[] ]` will produce an array of all the values in `x` followed by all the values in `y`.

Assignment statements exist, but do not mutate anything: the language is pure-functional. Instead, they identify the locations in the input that *would be* produced by the filter on the left-hand side, and output a new object with those locations replaced by the value on the right-hand side; this holds no matter how complex the left-hand filter is, including when it uses constructs like the `select` function. `.x[].y |= (1,2)` produces *two* output objects for every input, with every location that `.x[].y` would have produced being updated to hold first 1, then 2.

2.1 Comprehension and Debugging

The interactions of all of these features often do not conform to user expectations: there are several hundred questions on Stack Overflow and Unix & Linux Stack Exchange pertaining to unexpected jq behaviour, often involving branching, variable bindings, or updates. The functional language inside jq is mostly hidden and not considered by most users. Even for users who do, properties of the system make debugging challenging:

- the tool inescapably flattens its output, masking the branching that produced it
- typical Unix pipeline debugging strategies, such as truncating the pipeline to see what is produced by its early segments, are often unhelpful, both because of the flattening and because eliminating later branching can be a semantic change
- capturing constructs like array constructors are all-or-nothing, the intermediate values that went into them not only not visible but not able to be *made* visible
- function arguments are not visible at all, and there is no straightforward edit to allow inspecting the argument values alongside their input and output values.

Neither conventional Unix nor JavaScript debugging techniques are helpful, so for many users, jq is a black box.

For example, consider the jq pipeline `.[].x[] | .z as $z | .y[] | select(. % 4 == 0) | {z: $z, y: .}`: it accepts data of the form

```
[
  { "x": [
      { "y": [ 4, 5, 6, 7, 8 ], "z": "Apple" },
      { "y": [ 9 ], "z": "Orange" }
    ]
  },
  { "x": [
    { "y": [], "z": "Banana" },
    { "y": [ 0 ], "z": "Pear" }
    ]
  }
]
```

where any of these arrays could contain many (or no) elements, and produces outputs like

```
{ "z": "Apple", "y" : 4 }
{ "z": "Apple", "y" : 8 }
{ "z": "Pear", "y" : 0 }
```

with the structure flattened away.

It is common to build up Unix pipelines piece-by-piece, or to debug them by truncation to see intermediate results, and so it seems reasonable to use the same approach with jq. However, truncating this pipeline by removing the final component will output only the numbers `4, 8, 0`, removing one further component merely a longer list of numbers, and in neither case is it readily possible to tell what has caused each value to appear or even *how many* of the outputs arose from any given stage or part of the input. Because jq pipelines are in fact defining *trees* of transformations, the derivation is important for the semantics, unlike in Unix pipelines that are flat one-to-one streams, and so flattening out to the eventual results, while likely useful when using a working jq program within a shell pipeline, loses critical information when debugging. At the same time, function arguments are subject to the same nondeterministic evaluation, so `select(.y[] > 0)` selects the input for each time a value in `y` is positive: there is no way to truncate to see the calculation, and removing `select` will produce a sequence of unmarked booleans. While simple jq programs that are only linear chains (such as a series of `.field.accesses`) will not experience this loss of information, as soon as constructs that introduce nondeterminism (such as array iteration), dependency on prior branching (such as variable binding), or discarded intermediate values (such as the `select` function) are used, the derivation tree is important, but hidden.

3 A Visual Interface

Because the focus of a user of jq is on the JSON values being processed, and the programs themselves are generally brief and cryptic, we have designed a system that renders the values primarily. The program source is shown (and editable) at the top, but the body of the display comprises JSON values. The user provides an initial input value, or multiple values, and the effect of the program is shown beside. Figure 1 shows a simple example using the program from the previous section, illustrating both the data-driven branching (with rows splitting going rightwards), and dead threads of evaluation.

The derivation of a jq program is a directed acyclic graph with adjacency constraints on join points, which can be rendered in a tabular layout. Each cell represents a single value that is consumed (or produced) by a stage of the pipeline, and the result of the *next* stage is in the cell to its right. The original input value will be at the leftmost cell, and the final column will contain all the output values. When a single value results in multiple for the next stage, the row splits into multiple cells to its right. When multiple values are combined into one, as in an array constructor, one collecting cell spans multiple rows to the right of all of them. When a thread of evaluation terminates with no output, the remainder of the row will be empty.

Due to the potentially very large number of total stages, by default the display shows only the effects of each pipe-separated combined filter — that is, for a program `.x.y | select(.z>0) | .a[0]`, it will show three columns only, plus the original input. The user can expand any column if desired, so that `.x` and `.y` are shown separately, and similarly for `.a` and `[0]`. A program with *no* pipes will start directly in this expanded form.

Array constructor expressions capture all outputs within them, written by surrounding any subprogram with square brackets. Figure 2 shows different array construction programs in the visualiser; it is clear where the values come from, and how they are combined into a single value for later stages as large multi-row brackets are rendered around all captured values. This figure also illustrates the evaluation of the comma operator, which produces all of the outputs of all comma-separated subprograms as its own output.

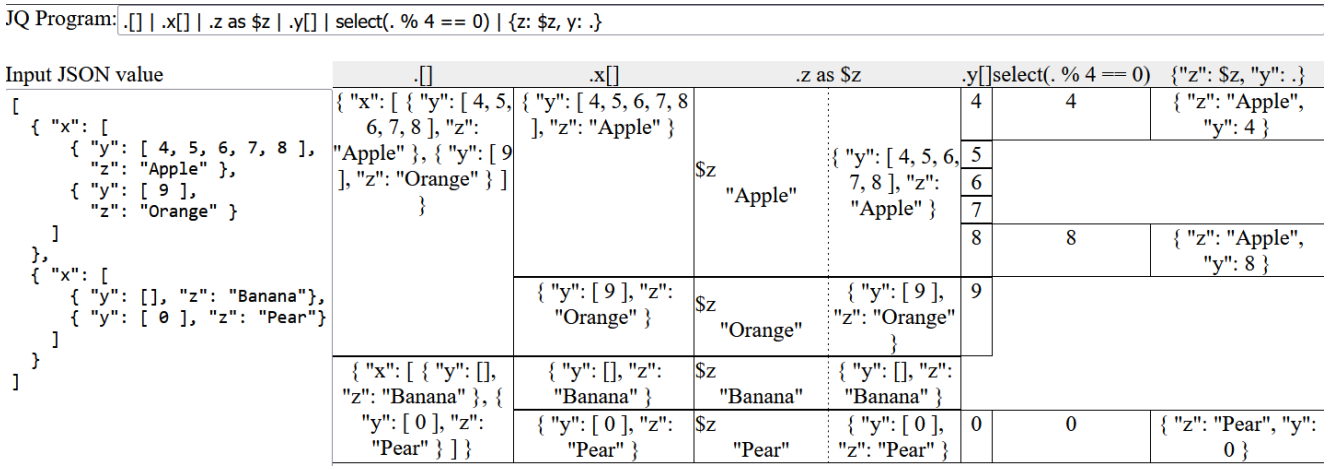

Figure 1. The jq program `.[].x[] | .z as $z | .y[] | select(. % 4 == 0) | {z: $z, y: .}` seen in the visualisation environment, operating on a small JSON object that illustrates the branching (the multiple subrows splitting off to the right of some cells) and non-producing elements (the partially-empty rows). The empty cells indicate that there was no output from this filter for the input value to its left. The non-empty cells in each column correspond to the output of a truncated program as discussed in Section 2.1.

3.1 Interaction

The system described so far renders traces of the program evaluation on the given input, but it is also interactive in two principal ways. The first is a simple live editing arrangement: the program text displayed at the top is editable, and the display updates in real time as the user changes the program, or when changing the input data; temporarily-invalid inputs while typing freeze evaluation until completed. Tightening the feedback loop in this way is presumptively helpful, but allowing interaction with the values themselves brings the problem domain closer to the source code.

Any value cell can be selected and plausible filters to apply to it will be offered, based on the type and shape of the value.

One other case arises: array constructor expressions such as `[.x,.y]` collect multiple produced values into a single output. When writing a program traditionally, the filters are written inside the brackets and so are isolated from the surrounding program, but semantically they run as they would without the brackets and are then collated into a single value. Selecting multiple values that *could* have been produced within a single array constructor will offer this, much more drastic, conversion also. Figuring out where the boundaries on this collation would be is otherwise nontrivial, and can be a complex refactoring, while the reverse direction is sometimes helpful.

4 Implementation

Behind this tool is a reimplementation of jq in JavaScript, based on the documentation and behavioural testing of the original. The original jq is written in C, and is a complex program with many features, but we have focused on the core language and its semantics. The reimplementation lacks several features of the original, such as its robust Unicode support, but includes all of the core semantic elements relevant here. The JavaScript backing library developed for this project is available at https://github.com/mwh/jqjs. It follows a tree-walking approach using generators to represent the production of multiple values in each filter, and is intended to be a more approachable reference for the semantics of jq than the original implementation. A particular element of it important for this tool is support for tracing the evaluation of a pipeline. Tracing produces a structured representation of the steps taken and the intermediate values produced in each branch, which mainline jq cannot provide.

The user interface runs in a web browser, entirely on the client side. The user can provide a JSON input value and a jq program, and the trace of that program and input is rendered across the page. The trace updates live as the program or input is updated, and can be interacted with to make modifications as described in the previous section. The tool can be accessed live at http://ecs.vuw.ac.nz/~mwh/demos/paint2023. While it performs reasonably for relatively large inputs, for either very large initial values or programs where branching blows out exponentially, it eventually becomes slow and unresponsive, and in some cases runs into limitations of browser layout engines.

5 Related Work

A number of online jq tools exist, generally operating by running the native jq tool on a user-provided program and input (either on a server or cross-compiled to WebAssembly). These tools present the flattened results of *entire* jq programs, just like the command-line tool, and do not enable

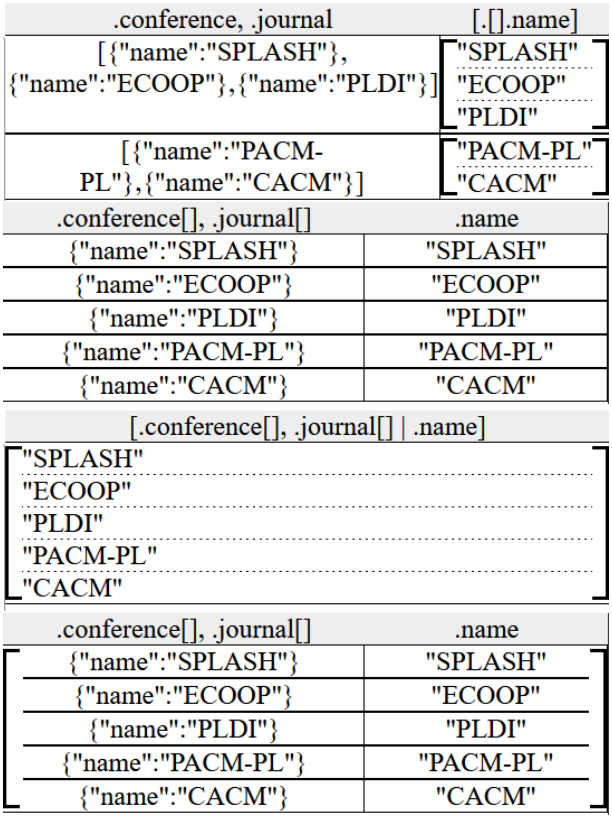

Figure 2. Slightly different jq programs operating on an object containing arrays of conference and journal names. Top: `.conference, .journal | [.[].name]`, producing *two* output arrays. Second: `.conference[], .journal[] | .name`, producing five output values. Third: `[.conference[], .journal[] | .[].name]`, producing a single output array. Bottom: the same, but with intermediate steps expanded within the array constructor.

any deeper analysis. These include https://jqplay.org and https://jqkungfu.com/. There are also native "interactive jq" tools [1, 5] and a "VS Code jq playground" extension, which provide live updating of the results of the full pipeline as the program is edited, with some autocompletion and documentation features. All of these tools produce the result of an entire jq pipeline, and do not give any insight into intermediate values or the structure of the evaluation. However, they do enable a tighter feedback loop than the traditional command-line tool, and by leveraging it internally they have access to the full range of its functionality (other than IO).

The most widely-used dataflow programming environment is the spreadsheet, which is also a two-dimensional grid of cells, each of which can contain a value [13]. Spreadsheets also make (some) intermediate values tangible and legible; however, the layout of a spreadsheet is cosmetic rather than semantic, and dependency relations are determined by formulas not normally visible to the user. In this system, the layout is a direct representation of the dataflow in all cases. Several programming systems deriving from or building on spreadsheets provide additional data types and further "programming" functionality [3, 6, 18].

Userland [16] is a spreadsheet-like system for dataflow programming that incorporates some more of the structural semantics seen in the present system. In particular, Userland supports Unix pipelines within the environment, with the same row-of-cells layout and split of the command across cells at pipe boundaries, which partially inspired the display here. However, as Unix pipelines are "flat", there is only ever a single row of textual cells for each command, with no branching or multiplicities as introduced by jq semantics. The tabular representation of ordered data dependencies here also draws on past work representing concatenative programs in two dimensions [10].

Muhammad [15] analysed semantics of widespread end-user dataflow languages with some level of visual interface, including spreadsheets but not jq. In the terms of Muhammad's taxonomy, jq is a *textual sub-language* of our system, and has no *N-to-1 inputs*; the system we describe has *indirect connections* in the form of jq variable bindings, has *separate program and UI*, and represents *iteration* as vertically-adjacent cells of concrete iterated values. This taxonomy is focused on editing and semantics of programs, rather than visibility of the values or dynamic structure of evaluation. In the terms of Hils [8], on which Muhammad builds, this work has a *data-driven execution mode*; *procedural abstraction* is a feature of baseline jq that our system does not expose. In the terms of Tanimoto [20], this system attempts to have level 3 liveness, but does not fully reach this level: only some parts of the program are editable in the "visual" (tabular) representation, and only in some ways, but modifications do trigger updates to program evaluation and there are no specifically separate editing and running modalities.

XQuery [17] is a system for querying and transforming XML documents, parts of which follow similar principles to jq. Some visualisation and exploration tools for XQuery exist, both commercial and in the literature [2]. The XQuery data model diverges from jq quite significantly, however, so it is not clear whether transfer in either direction is constructive.

"Big data" pipelines are another case of potentially-branching dataflow transformations, and tools for debugging systems like Apache Spark deal with some similar issues [7]. These tools must first and foremost address issues of scale, while also dealing with arbitrary directed graphs, but allow exploring derivations and intermediate values. Elements of this system may be applicable to them in some cases.

Numerous visual data-flow systems use a graph-based representation of the program, with nodes representing operations and edges representing data flow, such as in Pure Data [4]. In some cases these nodes are equipped with convenient renderers for the the data values produced by them, including Natto [19] and Calling Cards [9]. While showing

data values, the principal structural determiner is the source code structure, rather than the dynamic structure imposed by the data values in use. An earlier version of this system attempted to include a graph-based visualisation of evaluation, as either a substitute or alternative view, but this was found to be either trivial (for linear pipelines) or incomprehensible (with any level of branching), with little middle ground.

6 Conclusion

While widely-used, the jq language has uncommon semantics that are not always obvious, and elements of the standard tool make reasoning and assessing the evaluation of a program difficult. The syntax is quasi-concatenative, the semantics feature data-driven branching, and implicit flattening obscures the origins of produced values. We have shown a quasi-visual environment that compensates for many of these limitations, while covering a wide range of jq functionality. This environment permits both examining the derivation of jq outputs from programs and input, and making commonplace edits to such programs, with direct reference to the real data.

6.1 Future work

This interactive system offers some opportunities to communicate further aspects of the jq program. One in particular noted by industrial observers is performance. The evaluation strategy of jq is opaque, and some constructs may be much slower than others producing the same result on the expected input without this being obvious to the user. Highlighting these, or even providing a performance estimate alongside the visualisation, could be helpful for users in the interactive environment where they can explore changes speculatively. We are not aware of any systematic study of jq program performance characteristics currently to draw on for this.

Some other tools have similar execution models to jq, including both "jq-but-for" tools like fq, sq, xq, and yq, and elements of XQuery/XPath [17]. The visualisation techniques described here could be applied to these as well. Some further-afield systems, such as .NET's LINQ or the list monad in functional languages, could also have this approach applied.

References

[1] G. P. Anders. 2023. ijq: Interactive jq git repository. https://sr.ht/~gpanders/ijq/.

[2] Jihad Boulos, Marcel Karam, Zeina Koteiche, and Hala Ollaic. 2006. XQueryViz: An XQuery Visualization Tool. In *Proceedings of the 10th International Conference on Advances in Database Technology* (Munich, Germany) *(EDBT'06)*. Springer-Verlag, Berlin, Heidelberg, 1155–1158.

[3] Glen Chiacchieri. 2018. Flowsheets v2. https://github.com/Glench/Flowsheets-v2.

[4] Bryan W. C. Chung. 2013. *Multimedia Programming with Pure Data*. Packt Publishing.

[5] fiatjaf. 2021. jiq: jid with jq. https://github.com/fiatjaf/jiq.

[6] Monica Figuera. 2017. ZenSheet Studio: A Spreadsheet-inspired Environment for Reactive Computing. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. ACM, New York, NY, USA, 33–35. https://doi.org/10.1145/3135932.3135949

[7] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 784–795. https://doi.org/10.1145/2884781.2884813

[8] Daniel D Hils. 1992. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing* 3, 1 (1992), 69–101. https://doi.org/10.1016/1045-926X(92)90034-J

[9] Michael Homer. 2022. Calling Cards: Concrete Visual End-User Programming. In *Programming Experience Workshop*. https://doi.org/10.1145/3532512.3535221

[10] Michael Homer. 2022. Interleaved 2D Notation for Concatenative Programming. In *ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. https://doi.org/10.1145/3563836.3568722

[11] Timothy Jones and Michael Homer. 2018. The Practice of a Compositional Functional Programming Language. In *Asian Symposium on Programming Languages and Systems*. https://doi.org/10.1007/978-3-030-02768-1_10

[12] jq. 2023. jq is a lightweight and flexible command-line JSON processor. https://jqlang.github.io/jq/.

[13] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[14] John McCarthy. 1959. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 26. Elsevier, 33–70. https://doi.org/10.1016/S0049-237X(09)70099-0

[15] Hisham H. Muhammad. 2017. *Dataflow Semantics for End-User Programmable Applications*. Ph. D. Dissertation. Pontifícia Universidade Católica do Rio de Janeiro. https://hisham.hm/thesis/thesis-hisham.pdf

[16] Hisham H. Muhammad. 2019. Userland. http://www.userland.org/.

[17] Johnathan Robie, Michael Dyck, and Josh Spiegel. 2017. *XQuery 3.1: An XML Query Language*. Technical Report. W3C.

[18] Advait Sarkar, Andy Gordon, Simon Peyton Jones, and Neil Toronto. 2018. Calculation View: multiple-representation editing in spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 85–93. https://doi.org/10.1109/VLHCC.2018.8506584

[19] Paul Shen. 2021. natto website. https://natto.dev/.

[20] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

Received 2023-07-17; accepted 2023-08-07