Reclaiming the Unexplored in Hybrid Visual Programming

Michael Homer
mwh@ecs.vuw.ac.nz
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand

Abstract

Programming languages have been trapped in a world of linear textual representations fundamentally unchanged for half a century. Even systems pushing beyond these forms — visual languages, projectional language workbenches, and end-user programming tools — largely ape the strictures of stream-of-bytes compilers and confine themselves to the popular paradigms of conventional textual systems.

Instead of recreating what succeeded in textual paradigms, new programming systems should also be exploring what did not — the confounding, confusing, convoluted approaches that fell by the wayside — with the sorts of direct manipulation, spatial connection, and change over time that textual languages could never match; and they should use their control of presentation to let the user choose the right representation for a piece of code in the moment — and change it. We argue that these two points unlock new frontiers for programming systems, and present preliminary explorations to highlight how multiple-representation environments can lower the pressure on more speculative visual paradigms, to encourage more investigation of this underexamined space.

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

Keywords: visual programming, dataflow, concatenative

ACM Reference Format:
Michael Homer. 2024. Reclaiming the Unexplored in Hybrid Visual Programming. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3689492.3690045

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

1 Introduction

Most programming languages are text-based, and over time, familiar patterns have stabilised in the design of these languages. Non-text-based "visual" programming languages also exist, and fall principally into two camps: those that are essentially graphical versions of successful approaches to textual languages, and those that directly manipulate data- or control-flow graphs.

This paper makes two arguments:

- first, that new visual programming systems should explore a wider range of approaches, including areas that have been *un*succesful in textual form;
- second, that the all-encompassing nature of both visual and textual programming systems is fundamentally limitating and has not been well-addressed in either domain.

Visual programming languages have the opportunity to explore aspects of programming that text-based languages have not been able to capture, but to do so effectively they must break out of the mould of existing languages. At the same time, they typically embody a single visual paradigm that the program must conform to, often with compromises in editing, comprehension, or expressiveness that are not ideal for all parts of the program or all audiences. Choosing a different approach means choosing an entirely different environment, and the programmer is stuck with that choice for the life of the program; faced with these trade-offs, it is no wonder that purely-textual systems, for all their limits, tend to win out.

Instead, a system that allows the code to be presented in multiple ways lets the right view of this piece of code be used in the moment, without forcing it to be seen that way forever. In so doing, representations that have more limited uses can still be valuable, not having to be general-purpose paradigms.

We argue for more exploration of the potential range of non-textual programming models, including pathways that seem to have failed in the past or have limited application, and to work to allow these to coexist. After this, we will present some preliminary explorations in this space in the form of Djel, a visual, spatial, interactive programming environment including several contrasting representations of

program code that can be switched between or used in different parts of the program. In this way we can explore varied possibilities without needing any one of them to carry the full weight of the system. Section 5 then positions this argument and prototype in the context of existing work, outlining the gaps where we encourage more speculative explorations.

2 Motivation

Previous work [30, 41, 64, 65] has found that simply having the ability to *look* at a program's code in a different modality confers benefits to the programmer, even when there is no editing in the alternative view, while different modifications will be preferred in one view over another when the option is available. At the same time, many programming tasks are performed by people who do not consider themselves programmers, or in concert with non-technical stakeholders who are not interested or able to do programming or understand program code. Both of these groups can then be helped by exposing multiple different views of a program, with different emphasis and different affordances. Having these views be of the program directly, rather than a separate diagram or model, ensures that they are always in sync, and lets the program itself be seen in the most useful way for the task at hand.

Visual programming systems that exist often do not serve the *programmer* well. Their high viscosity [19, 55, 61], combined with often-poor modularity, reusability, and other issues [14, 22, 54], have been raised as serious problems for professional programmers. For smaller end-user tasks, these costs can be immaterial, but the opportunity for extension or integration with larger systems is often lost in a closed world. Often, these limitations result from trade-offs made to support different uses, optimising for legibility over editability, for example, but the result is that the programmer's work is more awkward than needed much of the time.

The issue here is that the same tool — whether typical program code or a visual environment — needs to be squeezed into disparate uses [32]. A system that allows the representation to vary, not just in the large through different major components using different languages, but in the small and in the moment, can obtain the benefits of all of these elements, and relieve the pressure on any one of them to be perfect.

2.1 Exploring the Unsuccessful

There have been very many programming paradigms put forward, almost all of which have seen little success or limited areas of application. The elements that led to their being proposed in the first place are still likely to have *some* value, however, and if their drawbacks can be reduced then more of their strengths can be exposed. Visual programming environments provide additional dimensions to work with, but most often just explore the same approaches as successful textual languages. Here we are highlighting a market inefficiency, so to speak, where investigating those less successful paradigms may find disproportionate benefits.

For example, block-based systems present, in effect, a direct-manipulation rendition of the abstract syntax tree of a textual language. In (only) doing so, they inherit the limitations of those languages, and add the costs of visual editing too. Virtually all of them also "trap" the programmer: there is no way out, or the way out means leaving behind what they are familiar with permanently. Some systems offer an export to a textual model, and a few allow both views to continue being used, including from Leber et al. [39] and Bau et al. [4] — these provide a glimpse of the multi-representation future we are advocating for, but are still deliberately conventional languages.

It is understandable that educational environments prioritise similarity to successful reference languages, but the same is not necessarily true for practical uses. For example, the concatenative language family (Factor, Joy, Forth, PostScript) gives a different perspective but is often regarded as "write-only languages" because of the difficulty of reading an unfamiliar program; on the other hand, they have been feted as expressing composition in a direct, concise way, and offering trivial abstraction of any part of code into a reusable function [52]. Could a visual system reduce the difficulties and expose the compositional abstraction more clearly? Our argument is that at the least it is worth trying, and other lesser-used paradigms will have similar trade-offs to explore. In multiple-representation environments, such a paradigm can be used where it is effective, with other representations available to carry the weight of the overall program when it is not, and so these explorations can have much lower costs than otherwise.

2.2 A Call to Action

Our programming systems can expose the code in different ways, can let the code be manipulated in the manner most helpful *right now*, can expose the way of thinking about the task at hand that is most useful to *this* audience, *this* user, *this* moment — even if that way has drawbacks in other contexts.

Doing so enables new users to participate in the programming process, whether they are interested in learning about it or not. It enables the programmer to use the most suitable, convenient, efficient modality for the task at hand, and without being stuck with it for all tasks and all hands. Possibly this will not work out — likely, in most cases, it won't — but currently there is too little activity in this vein to know.

The next section introduces one attempt to provide this combination for a range of dataflow tasks, exposing several different program representations tied together in a single system, to illustrate the potential (and some limits) of this approach.

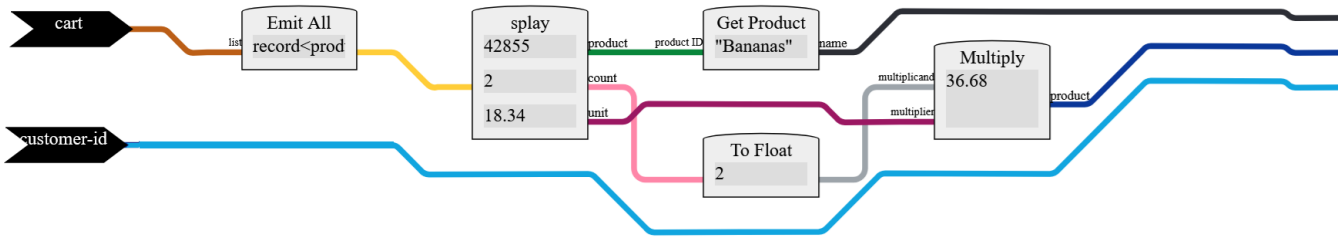


Figure 1. A simple program in the graphical dataflow view, transforming an input of a customer ID and list of shopping cart entries into a sequence of individual customer ID, product name, total price entries. This program will evaluate to many rows of output data for each input row, one for each item in the shopping cart. A user could scrub through each result or input to trace the behaviour of their subprogram, and the output sequence may be consumed and processed further by another part of the program, potentially in a different modality.

3 Djel

To support the motivations set out in the previous section we present several different program representations within an overarching environment. For this paper, we focus on the use case of processing essentially tabular data, and offer a number of speculative program representations suited for that task, with the idea that multiple may be used within the same program, and even on the same piece of code at different times. We call this overall system Djel, but many of its parts are severable, and what we want to highlight is the *breadth* of what is possible from pursuing even a single paradigm in multiple ways. Not all of the elements we present will be successful — at least one of them is almost certain to be a dead end — and that is fine. Exploring the space of possibilities to find out where there may be hidden value is the more important goal.

The underlying model in a Djel program is a directed, layered graph, where vertices represent operations with inputs and outputs, with inputs always from a node in an earlier layer. and operations in the same layer known to be independent of one another. These simple constraints facilitate several very different approaches to displaying and manipulating the program. We will show text-based modalities with more efficient editing, visual modalities that communicate the structure of the program, and modalities that foreground the effects on the data being processed. All of these expose *the same* program, but in different ways. The system permits switching between these representations freely, and communicates the correspondence between them through animating the transition, giving each fine-grained component a continuous visual identity throughout.

Evaluation can be conceptualised either as a flow of data through the graph, or as treating each layer as consuming and producing a tuple of values, with all layers composed together. This duality is what enables some of the variety of views possible, and is mostly transparent to the programmer.

To permit more complex tabular processing, we also permit an operation to have multiple *results*: it may produce multiple values for each of its outputs, in effect causing the layer it is in to produce multiple "rows" of output. In this case, multiple strands of evaluation proceed from this point, with each later layer essentially "map"ped across all the rows. Similarly, an operation could have *zero* results, terminating this line of evaluation. These sorts of tabular operations, including filtering out some rows, or inserting multiple rows in place of one, are common end-user processing goals, but here the complexities are abstracted behind the run-time model. Not all programs (perhaps not even most) will need this feature, but it aligns with one of the strengths of the concatenative model we are exploring so we include it.

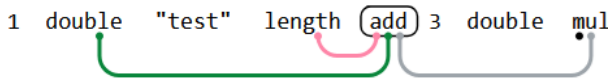
4 Representations

The following subsections each describe one core representations of a subprogram. All are interactive and can be edited. While very contrasting, they all share the same underlying model, and so the user can switch between them at any time. However, some representations are more restricted and can only display a subset of programs. The intention is that different functions the user defines will use a variety of these representations, and they may move between them during development.

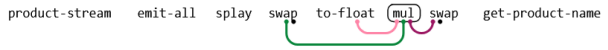
The first three representations and the model of transitioning between them we extend from earlier work [27]: a stack-based concatenative language, a spatial grid layout, and a graphical dataflow language, although the implementations of these views are not the same. We focus on the novel design improvements in these sections.

4.1 Graphical Dataflow

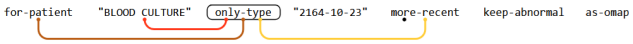
The graphical view is an essentially conventional visual dataflow language, where functions are nodes and data flows along edges. The concrete values being operated on may be displayed within the function nodes that produce them. In this model, the graph explicitly has layers, and all connections go from left to right. Connections can be formed by



(a) A trivial program in the stack-based concatenative representation, with a single function selected and all of its inputs and outputs indicated by lines to and from the respective other functions. Data dependencies are shown this way, and the user can interact with the display to trace the program.



(b) A similar program to Figure 1, but showing the need for stack-manipulation operations in a concatenative language and the limited information that would be provided without the visual overlay.



(c) A more friendly top-level program, composing well-designed helper functions to filter medical test results and produce the standard OHDSI OMAP Common Data Model-format mixed tabular output, without requiring stack-manipulation operations. These helper functions are in mixed representations themselves, with keep-abnormal using the spatial grid and as-omap a graph view.

Figure 2. Some examples of programs in the stack-based concatenative representation.

drag and drop, and functions replaced with type-consistent alternatives from a menu.

This is a direct representation of the underlying model, and we present no particular novel features in this view. It serves as an "executable diagram" depicting the overall structure of the transformation the program performs, particularly suitable for mixed-technical audiences where other sorts of diagram may be used, such as UML or flow charts.

4.2 Stack-Based Concatenative

The stack-based concatenative representation renders the dataflow graph as in a stack-based textual language like Factor [2], Forth [45], or Joy [63]. Programs in these languages are sequences of operations that manipulate a stack of data values, with each operation implicitly consuming some values from the stack and producing others into it. From an alternative perspective, they are functional languages where juxtaposition means function composition, rather than application. The implicit connections between operations via that stack define a dataflow graph, and so any program in this model can be transformed into the graphical model behind Djel. However, not every dataflow graph can be represented in this model, as the stack is a linear structure and only the operands on top can be accessed.

This model of language can be very efficient for a programmer to construct new pipelines in [16, 52]: composing long pipelines requires little bookkeeping work, simply naming the composed operations, and abstracting out subfunctions is trivial. However, they are completely inscrutable to non-programmer audiences, and often even to programmers unfamiliar with the specific set of functions in use, to the point they are sometimes called "write-only languages". Thus, while they can be highly productive for producing new code, they are less suited for maintenance or communicating to other stakeholders.

The motivating issue for this view in Djel is an inherent challenge of the concatenative model, namely that understanding the functionality of unfamiliar code is difficult: it requires mentally simulating the stack state at each point in order to know which function's output is being used as which other function's input. A visual environment offers opportunities to counter these difficulties while retaining or reinforcing the positive elements. These data dependencies can be displayed explicitly, significantly aiding the maintenance programmer, while retaining the quick, concise expression that is the strength when writing new code, including new code sited within an existing program.

In our earlier work [27], a stack-based representation was exposed purely as plain text. In this work it is primarily a structured representation instead, although the user can still edit the program as ordinary text if desired. Figure 2 shows a simple program in this view. Individual functions can be selected or hovered over to overlay connectors to the sources and destinations of that function's inputs and outputs, as shown in the figure. In this way, one of the primary difficulties of stack-based languages — needing to understand and track the arities of all nearby functions — is ameliorated by the visual display. This allows the strengths of this style of concatenative language to be exposed without the primary drawback, using the additional dimensions that a visual environment has available. A program is a concise description of simple compositional pipelines of functions that is easy to write.

There are additional structural constraints for using this representation, as in a stack language only the top elements can be accessed. These are the same as in prior work, and essentially require no wire crossing and that the last input to each operation is also the last output of another. Programs that do not meet the requirements cannot be displayed in this view, but suffer no other ill effects, and can be edited into a compatible form if desired. The system will not offer this view as an option when it cannot be used.

This view in particular reflects where less-successful approaches to programming can be redeemed by a visual environment: the established weaknesses of comprehensibility are mitigated, while the strengths are untouched. Only relatively small enhancements were needed to make the resulting programs more accessible and understandable, but these were simple and obvious within a visual-programming

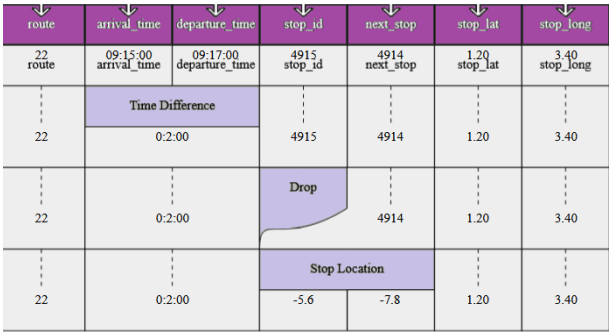


Figure 3. A simple program in the grid view operating on public transport route information, showing functions spread below the data value(s) they consume and above the values they produce.
This program cannot be expressed in the stack-based view because it uses values that would be below the top of the stack; for example, the "Time Difference" operation uses the two adjacent times without repositioning them first. The flow of data is from top to bottom, with each operation laid out below the values it consumes and above the values it produces.

paradigm. However, there will still be times when other approaches are more suitable, and the Djel approach of allowing different representations to coexist lets the concatenative style be used when it is helpful: a relatively linear pipeline will go well there, while a more complex branching structure may go less well. Instead of throwing away the whole approach, and its strengths, when it is an awkward fit for some part of the program, it can be used *for* its strengths when that helps, and mixed with other approaches when it does not.

4.3 Spatial Grid Layout

The spatial grid notation also draws from previous work [25]. In this view, the program and concrete data values are interleaved and able to be manipulated to edit the program.

The key motivation of this representation is twofold:

- to show the derivation of results, including the functions and values contributing to them; and
- to allow easy exploratory program editing, even by non-technical users.

Direct interactions allow investigating what else could have been done at any point, or picking a data value or values and seeing what could be done with them, without committing to any significant restructuring.

There is a grid of cells representing functions, where layout indicates the input and output relationships between them. Each function stretches below all of its arguments and above all its return values, and those values may be displayed within the grid.

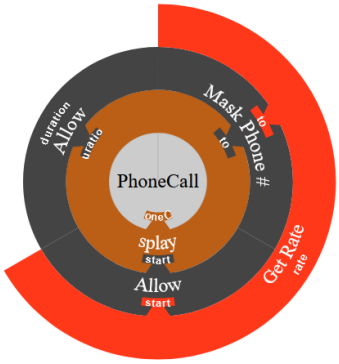


Figure 4. A program in a radial view, representing one of the suggested use cases for the view, filtering and masking information for data governance purposes. Values move outwards from the centre, being processed at each layer.

Figure 3 shows a program in this grid view. The flow of data is from top to bottom: at the very top, subprogram arguments and literal values appear, and below them are the operations that consume them, and so on down the grid. In between each row of functions is a row of values passing between the functions above and below. To edit the program, the user selects a consecutive sequence of values by dragging across adjacent cells. The system will then present a menu of operations capable of consuming all those values, and choosing one will add it to the program at that point.

4.4 Radial Graph

Not all visual representations of a program will turn out useful. Consider, for example, the view in Figure 4. Here the program is displayed as concentric circles of operations: arguments in the centre, with each outer layer being divided into annular sectors representing operations consuming the arguments to their inside. This "layering" corresponds to an intuition expressed by users first encountering the semantics of this system, so we explored some variations to find a constructive model in this vein. However, it occupies a lot of space, while not having room to display any values, and does not present easy editing affordances; no variants of it were significantly better. Despite this, it is possible that some user will find this view, or an improved version of it, to be useful for some purpose, and in a multi-representation system modalities of more speculative value can be included without needing to support use as a primary interaction. While we cannot vouch for the utility of this view in specific, we do value the exploration of possibilities that it represents.

4.5 Tracing Grid

This view is distinguished somewhat from the others as it is not primarily about representing the program, but illustrating the data-flow trace producing the outputs of the program. We previously proposed a version of this view for

search-category	customer-id	cart	Each	Splay			Multiply	10.0	Keep >	Prod. Name	Categories	Matches
				prodID	count	unit	product			name	category	
fruit	9354	[3 items]	prodID; < count; > unit	42855	2	18.34	36.68	10.0	36.68	Banana	fruit	✓
											imported	●
			prodID; < count; > unit	2174	8	0.75	6.0	10.0	●			
			prodID; < count; > unit	68713	3	7.68	23.04	10.0	23.04	Bread	bakery	●
fruit	4153	[1 item]	prodID; < count; > unit	2174	18	0.75	13.5	10.0	13.5	Apple	fruit	✓
fruit	6631	[0 items]	●									
fruit	6631	[1 item]	prodID; < count; > unit	2174	2	0.75	1.5	10.0	●			

Figure 5. This program in the tracing view is searching transaction records for a shop for any purchases where a customer spent more than 10.00 on any imported product. Three inputs were given: a category to search for, a customer ID, and a shopping cart of purchases.
The evaluation thread splits after the fourth column, with each product from the cart taking up a row of its own from then on, laid out beside the earlier, shared items. Reading a row from right to left shows the sequence of precursor values leading to that row's output.
Some threads of evaluation stop early, marked with a red hexagon; no further processing occurs after a dead end, so no further cells are filled to the right. This subprogram produces two results, and the user can trace why further paths were dropped, and how the final products came about.

the jq language [26], but in this system it can be applied to more general data-flow programs.

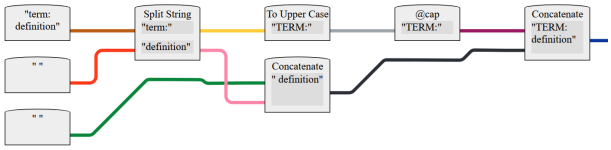
The principal role of this view is debugging or understanding complex derived sequences of data values. This serves both programmer and non-programmer users, as complex pipelines that can drop values, or produce additional values, can easily go awry, and identifying *at which point* this occurs is challenging with both conventional programming and other visual representations. The tracing view aims to let a programmer see when their expectations are upended, and to let a domain expert see how the program has treated the full set of data points collectively to observe patterns that need correction.

In this view, seen in Figure 5, the functions of the program are displayed along the top of a grid, with the labels of their outputs arrayed below (if any). Each row of the grid represents one pass through the pipeline: a cell displays an output value of the function above, given all the outputs to its left. When a function has multiple outputs, such as the `toRGB` function that produces the red, green, and blue coordinates of an input colour as separate outputs, multiple subcolumns are arrayed below the function header.

A function that produces multiple *results* (that is, multiple parallel sets of values for its outputs) will split the row at this point, producing one row for each result. The cell to the left will vertically span all of these rows, and to the right the system continues as before. In the case of a function

```
before, after = split "term: definition" " "
cap = to-upper before
result = concat cap (concat " " after)
produce result
```

(a) A very simple program in the applicative view, resembling the example in Figure 3. The "before" and "after" names derive automatically from the function output labels, but "cap" is a user-defined name.



(b) The graph corresponding to the applicative program in (a).

Figure 6. A program in the applicative representation and its corresponding graph.

producing *no* results, that row will terminate early, with no further cells filled to the right.

This view shows the concrete flow of actual data values through the functions of the program, but does not indicate their structure or dependency explicitly. When hovering the pointer over a column, the system will highlight the columns used as input for that function, and thus the values used to produce any given result in it. In common cases it resembles a typical pattern in spreadsheets, where incremental calculations are built up across the row, but here with more inherent structure behind the computations. This view is available for any program in the system, but is most useful for programs that are closer to linear pipelines, especially those that fork into multiple paths.

Some other indications of dependencies may be useful in more complex programs, such as the overlaid connection "hops" used in the stack-based concatenative representation from Section 4.2. The present prototype does not provide any such overlay, but of course the user can switch to more explicit views at any time, and back to continue tracing the program.

4.6 Applicative

The applicative view is a textual representation of the program in an ML-like structure. Each line is a function call, with its results assigned to a sequence of named variables. The names are generated from the output labels, if any, or otherwise from the types, but can be edited by the user. Function arguments are literals, named variables from an earlier line, or nested function calls in the case of linear compositions of functions. This view is a direct representation of the underlying dependency graph, and so can represent any subprogram in the system.

The value of this representation is familiarity for any programmer experienced in functional languages, with swift

editing and the traditional affordances, while retaining compatibility with all the other views. Because it is a direct representation of the underlying semantic graph, a user may make edits in here even if showing a diagrammatic view to other users is intended in future, and subprograms written here can be used from any other view with no friction. This representation offers the most efficient editing to a programmer, while less accessible to non-technical audiences. However, as plain text it is the most accessible to assistive technologies, which are extremely ill-served by most visual programming environments. The current prototype implementation only presents a standard text box when in this form, but more advanced IDE features could be added. The initial view after transitioning into this form is a structured block-based one, which does display some minor overlays for connections and types, but it converts to plain text when the user begins editing.

When transitioning to this format, function names, labels, and types move to the corresponding points in the text. Because (most) variable names appear more than once, as many duplicates as are required will also animate to their respective destinations.

4.7 Inline Concatenative

Inline concatenative languages are a rare form of the already-rare concatenative languages, but do offer some advantages for tracing and understanding. Initial visual representation of these languages was proposed previously [28], but here we do not take on the complexities of the prior work's multiple-track evaluation. In these languages, there are no variables or hidden stack of data: all data values are directly embedded within the program source, and execution (conceptually) updates that source code.

A visual system can show each phase of single-step evaluation and the resulting program in sequence at once, so that domain expert users can see where a computation goes off track. The unique nature of this representation is that these intermediate states are themselves valid, self-contained programs: they can be promoted to editable, explorable functions that can be refined, investigated, and incorporated into the main program. Only the top row is editable, although the later rows can be interacted with to display information about the functions or values.

This view works for only a limited subset of programs, and the present prototype has some difficulty determining which those are. While it shows some promise, further work is required to determine whether this offers benefits above any other representation of the program.

4.8 Notebook

"Notebook" systems, such as Jupyter, are a popular way to mix code and text in a single document, often used in data science and other exploratory data analysis tasks. Typically, there are interleaved cells of ordinary document content and code blocks, often in languages like Python, R, or Julia. The code blocks can be executed in place, and the results displayed below the block; all blocks share a common execution context, so variables set in one block are available in later blocks. Code blocks can be edited in-place and re-executed when the user has adjustments, or to refine the functioning of the code. This paradigm is extremely popular, particularly for users for whom programming is not the goal but a tool for some other purpose. However, the notebook paradigm has some significant limitations as well, in particular that is is possible for the state of the notebook to become out-of-sync with the code blocks, or for accidental dependencies on later or removed code to be introduced. Some "reactive" notebooks, such as Observable, have addressed this by laying a dataflow graph over the code blocks, still implemented in conventional languages avoiding some of these issues while creating a discontinuity between components.

We can recreate this model in Djel, with a single continuous flow through the notebook preventing out-of-order dependencies arising. The text blocks of a notebook are simply distinguished forms of comments, and code blocks subprograms that can be displayed in any available format. Figure 7 shows a trivial notebook using two different representations

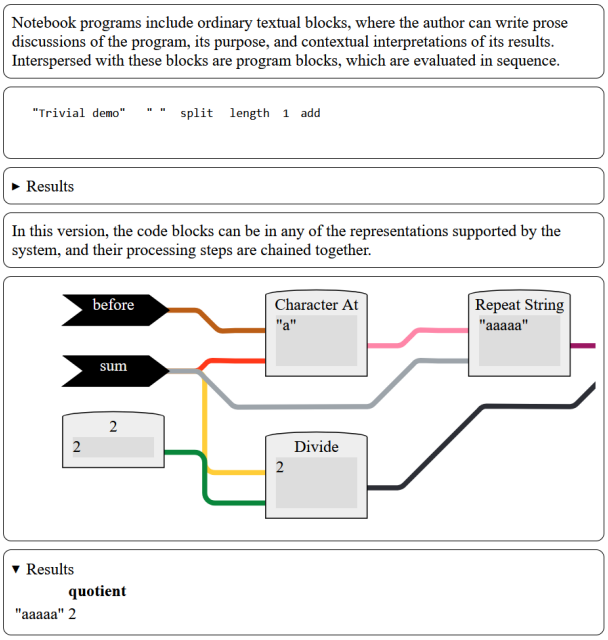


Figure 7. A very trivial program in the notebook view, showing a combination of the stack-based concatenative view and the graphical dataflow view in the two code blocks. The single result is seen below the final code block; a larger program could produce multiple rows there. Clearly, programs in this view can take up a lot of space very quickly, but in the context of a practical notebook a more realistic-sized program would not be an issue.

in its code blocks at once. These notebooks may sometimes be switched into other views, although most are not well-suited for doing so.

The ability to do so, however, highlights the essential sameness of the semantic model, and further extensions may allow more advanced connections.

4.9 Embedding Other Models

Djel imposes structure at the boundaries of a subprogram, but does not strictly require that the implementation within that subprogram follow the whole model. Provided that it is able to accept inputs and produce outputs in a compatible way, it would be possible to embed other languages inside, allowing an escape hatch from the model when necessary or desirable.

To explore this possibility, we implemented an embedding of the jq JSON processing language, which has a reminiscent, but not identical, execution model. Ultimately, this embedding was not as successful as hoped, and "similar" was not quite similar enough — jq has some "capturing" semantics that are critical to common patterns, but which Djel's model has no equivalent of — although a follow-up embedding of the textual jq language as a self-contained function was more effective. We discuss some of the discoveries from this experiment in Section 5, including some possible extensions to the Djel model that would allow the jq semantics to work.

4.10 Implementation

The implemented prototype system is a web-based environment accessible through a browser at https://djel.org/.

5 Discussion and Related Work

Our prototype exposes different views of code, and allows them to be used in different parts of the program. This allows some degree of freedom in choosing the right representation for the task at hand, but not unlimited. Jakubovic [32, 33] argues that "self-sustainable" programming systems are required, which will allow their users to choose, and build, the right notations for the task. Particularly relevant is the argument that

> Instead of seeking the right notation, interface or representation for the job, we might seek the right textual syntax for the job. If we cannot find one, the real reason may simply be that text is not well-suited to the job (*[figure of a spreadsheet]*). Yet if text is all we know, we will be under the false impression that it is an intrinsically hard job.

and the concept of "notational freedom", the ability to choose any desired notation for a part of the program with no cost. Djel from the previous section is a *step* in this direction, providing a variety of notations and allowing a degree of run-time mutation, but does not meet the other criterion of a fully open system with unrestricted mutation. It has a selection of modalities, and code in the same program can use a mixture, but there is always a single mode in use at one time for one piece of code, and new modes are not creatable within the system. In the cases where one of the available modalities suits the task at hand, this works out as well, but while there is a broader range than most systems it is not unlimited. Future and alternative work could expose more of the underlying structure to the user, allowing them to build their own representations, and a more advanced editing environment could permit better integration or nesting of modes. At times it would be useful to have components *within* a single displayed subprogram shown using different modalities than the surrounding code that suit the moment, which this system does not provide, but has been touched on in other work.

5.1 Embedded Visual Representations of Code

Conceptually, visual interfaces for terms of a program originate no later than Smith's Pygmalion [57], which had limited practical implementation. There are both total visual languages, such as LabVIEW [12], and primarily-textual systems including embedded interfaces for select portions of the code. Our model combines multiple representations in one system, and there are two broad ways such a hybrid can work: embedded visual representations within other code, and alternative visual representations of the code. Our proposed model is primarily aimed at the latter, to allow lower-stakes exploration of different visual modalities, but there is value in both. This section discusses existing work on visual representations as part of other code, and Section 5.2 considers systems that instead present multiple representations of the same code.

Lorgnette [18] is a framework for incorporating domain-specific editors for *part* of a program into conventional textual code editors. For example, colour pickers, tabular interfaces for editing tabular data, and run-time variable traces could all be exposed within the editor in-place where the relevant part of the code is, and the framework defines a general approach to creating and integrating bespoke visualisations or editors on top of any language. These views or editors can be displayed alongside the code temporarily, and parallel Hazel's LiveLits [50], the hints from Tiled Grace, interactive syntax in Racket [1], or Envision's [3] editor projections, among others.

Visual Replacements [5] is another model for embedding domain-specific visual representations within a textual code editor, glossing certain code fragments and potentially including concrete run-time data. Elliot [10] introduced "tangible functional programming" in which values can be displayed as editable GUI elements and higher-level values constructed out of them. Polytope [13] similarly combines text and visual programming within one another, a hybrid system with mixed structured and unstructured parts. Other than

in the notebook view, Djel does not focus on the sorts of composed embeddings that these systems do, but the direct-manipulation interfaces are a potentially useful extension.

Mage [37] is a system for interwoven notebooks with visual, textual, and interactive components. Code is generated *in response to* interactions, allowing most operations to be performed by direct manipulation, with more typical code as the backend result. Engraft [31] is a system for rich live editing of data to produce persisting transformations, incorporated within broader programs. Its rich environments can be nested inside one another, contribute data to one another, and be slotted within external programs generically. Both of these systems present a more direct mechanism for transforming data than our prototype achieves, but they present valuable pathways for building programs in the less-common paradigms we suggest exploring.

5.2 Multiple-Representation Programming Environments

A number of different approaches to wholesale multiple-representation programming environments have been proposed. In these systems, the same piece of code can be seen and edited using different modalities, rather than contrasting representations for different segments of the code. In most cases, there is a core textual language that is "ground truth" for the program, and visual representations are derived from that, while systems supporting multiple visual editing modalities as we have proposed are much rarer.

Within these systems, there are both switchable- and simultaneous-display techniques, either allowing the user to switch to and from the textual view, or displaying two versions of the code alongside one another. Both switchable and simultaneous displays have had benefits attributed to them, but both have also been suggested to make tracking the effect of changes harder. Our present prototype includes only the switchable approach.

Tiled Grace [29] is a block-based editor for the pre-existing textual Grace programming language, and introduced animated transitions between Scratch-like [53] block and traditional text representations. This system also includes some "overlays" and "hints" that depict some additional information on top of the main display, such as data dependencies, previews of colour values, or selectable menus of known accessory values like images. Droplet/Pencil Code [4] is a switchable dual-mode editor with fixed configurations, rather than freeform placement, which similarly includes animated interstitial steps. Poliglot [39] provides simultaneous multiple-representation editing, with a block and text display of the same program side-by-side. Edits on either side are immediately reflected on the other, but some intermediate states are invalid in one or the other language and delay the update. All three of these fundamentally manipulate a conventional text-based language, with the visual representation as a secondary view of the "ground truth" of the text. Our Djel prototype has multiple textual modalities, but they are not the canonical representation, and in our view depending on a (particularly pre-existing) textual language for that imposes more constraints on the visual language than is desirable; these systems also support only a single non-textual format, rather than many. Simultaneous display of multiple visual representations seems valuable at least as an option, and future work should explore this.

Sketch-n-Sketch [23], a simultaneous dual-representation editor for *diagrams*, allows both the diagram and textual code that produces to be altered to affect the other live. It allows abstraction of sub-diagrams into reusable components, but the fundamental output is always a fixed SVG image.

Programming environments supporting alternative *textual* representations also have some history. Projectional editors such as Cedalion [40], Gandalf [21], Mentor [8], and Isomorf enable primarily textual representations, including familiar-looking syntaxes for different language families, and have been noted as providing enhanced interactions combined with usability challenges for programmers [62]. Alternative textual representations like these are likely to have value, but are not a focus of this work.

5.3 Graphical Dataflow

Data-flow programming in particular lends itself to graphical representations [17, 47]. A number of node-and-wire visual languages, like our raw graph representation, exist following both data- and control-flow paradigms [34, 46]. These systems include Pure Data [6], Simulink [60], LabVIEW [12, 49], ProGraph [58], Yahoo! Pipes, Unreal Blueprints, and others [7, 14, 55]. Sometimes these are for specific purposes, such as music production [48], while others aim for general-purpose usage. These systems rarely display or focus on concrete values being processed, and alternative visual representations largely relate to graph layout algorithms only, while maintenance tasks such as testing and debugging are often reported to be challenges for these systems [36, 43, 55], particularly when input data changes. Multiple-representation environments, especially where views foregrounding concrete values exist, counteract some of these issues: rather than tracing a hypothetical through a graph, some of our views (e.g. both grid representations) display the data values and the operations affecting them in situ. Other tasks, such as testing, are not impacted as much by the visual representation but by the closed-world nature of the system, which our prototype continues, but which is not intrinsically part of the path we propose.

Enso [11], previously known as Luna, is a hybrid dataflow programming environment where the principal mode is connecting nodes in a graph on an infinite canvas, but nodes often contain relatively complex textual code configuration internally. Over time, the focus of the language shifted more towards orchestration of components implemented in other

languages, but there is also a textual format of the overall graph. Constructing complex data structures, which we noted as a weak point in Djel, was a claimed strength of Luna through specific visual nodes that 'look like' the final shape (e.g. a table, a tree), although it is not entirely clear how this panned out in practice.

Subtext [9] is a quasi-visual programming environment representing data values inline with operations, and explicit links between different points of the program. The core structure is a tree in the manner of an outliner [15], and evaluation essentially by cloning and aliasing subtrees recursively. This model is another possible representation that is unlike any in our prototype, but it has some analogies with the inline concatenative view; we speculate that a tree-based representation may be superior through having more well-defined constraints.

Spreadsheets themselves are the most widely-deployed data-flow programming method [20, 38]. They typically use both specialised textual syntax and spatial references, and more recently have included array and higher-order features [66]. Common spreadsheet patterns operate on rows of data, with cells making use of other cells to their left, perhaps among others. Errors in spreadsheets arising from calculation steps missed, or subtly altered, are a common problem [44, 51], and one that more explicit definitions of dependencies, and more exposure of intermediate calculations, can help to avoid.

Thyrd [42] is a purely-visual concatenative programming language and environment, where cells of a spreadsheet-like grid represent operations and values of the program processed in sequence. Intermediate results are not displayed by default, but the program can be structured to do so (as in a spreadsheet). Thyrd is one of very few explicitly concatenative visual systems, and sits somewhere between an ordinary spreadsheet and the inline-concatenative view of our prototype. While it is not clear that it overcomes comprehension challenges of the paradigm, its editing affordances may be efficient and worth investigating as another view.

Natto [56] is a cards-on-canvas environment where each card is a small program, often displaying its output, and connections between cards indicate data flow. Other work in this vein has used the aesthetic while focusing the choice of operations in the cards themselves, including Capstone [35] and Calling Cards [24], or within interactive widgets [59]. These are by and large operating at a higher level of abstraction than our prototype, but presenting individual functions as "cards" would be a viable alternative representation.

5.4 Evaluation Model

A central element of the Djel prototype is the unified layered dataflow model that underlies all the representations. The layering enables the stopping and splitting operations, and the lockstep functions-on-arguments presentations, while the dataflow structure means there is conceptually a traceable path from input to any given output, continuous through and inside user-defined functions it uses. These restrictions and abilities all enable some of the representatations — e.g. the layering is very directly employed by the spatial grid view — but using this model everywhere also imposes limitations on what can be represented and the breadth of representations that are available. For example, choosing between evaluating two segments of the graph is not possible, only discarding one of the results at the end, and feeding back the result of a (sub)program to the next execution can only happen through a higher-order function defined in the runtime. A different representation cannot alter these limits, although one could wrap some decorator function around the outside of what it displays (but none do currently). The explicit layering reflects the findings of Muhammad [46] that a visible, manifest ordering of execution is valuable in end-user dataflow systems.

Our explorations with jq are illustrative: it has a fairly similar model to Djel in most respects, but the differences in detail were enough to make embedding it directly problematic, and much more so than anticipated. In particular, it is very common in jq to capture all of the results of split threads into an array value (to the extent that an array "literal" `[1,2,3]` is capturing three outputs from three execution threads!), but in Djel these splits cover the whole layer, not just the values that directly caused the split; unifying just the correct values back to a single array value is semantically problematic as there may be parallel splits. These obstacles of small differences have been a recurring issue for multi-language projectional editors for textual languages as well. On the other hand, translating at only the input and output stage between the Djel and jq data models is relatively straightforward, and then the interior semantics are not important — indeed, at that point the inner code could just as well be Python, Haskell, or Prolog, simply presenting a black box to the outside. In some respects, this is where the Enso/Luna [11] system settled, after facing similar complications with only two representations. We believe that offering more representations will reduce the need for this, but semantic extensions to support greater variety are likely to be helpful in future work.

The current prototype has uncovered some real strengths of its model, but also limitations, some of which seem to be caused by the model itself. High-level pipelines are easily expressed and understood in the concatenative view, joining together multiple other functions that "do the work", while the spatial grid is most effective for local transformations where the concrete data is important, and has been the most common mode used for the data-massaging use cases we have been experimenting with. Scaling, computing differences of two columns, concatenating or formatting, or eliminating a column are all quite clear in that view, very

close to direct manipulation, and the result is seen immediately. Switching views — sometimes for no more than a single change, or with no specific modification in mind — soon becomes second nature once it is available. The tracing grid is extremely helpful for debugging pipelines that make use of the multiplicity feature, but also scales poorly to large datasets, or when there is heavy branching, something that the model encourages by making it superficially free in all views except this one.

6 Conclusion

Multiple-representation programming environments can explore a wider range of programming modalities, structures, and affordances than any single representation can provide. Supporting this allows the inclusion of representations that have different strengths and weaknesses, including those that have had limited uptake in the past, so that their benefits can be obtained despite whatever drawbacks they may provide in other circumstances. We have presented a prototype system incorporating several different editable program representations, able to be switched between at will, to highlight the breadth of the design space available following these approaches.

References

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding interactive visual syntax to textual code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (nov 2020), 28 pages. https://doi.org/10.1145/3428290

[2] Zackery L. Arnold and Saverio Perugini. 2019. An Introduction to Concatenative Programming in Factor. *J. Comput. Sci. Coll.* 35, 5 (oct 2019), 70–77.

[3] Dimitar Asenov. 2017. *Envision: Reinventing the Integrated Development Environment.* Ph. D. Dissertation. ETH Zurich. https://doi.org/10.3929/ETHZ-A-010863881

[4] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil Code: Block Code for a Text World. In *Proceedings of the 14th International Conference on Interaction Design and Children* (Boston, Massachusetts) *(IDC '15)*. Association for Computing Machinery, New York, NY, USA, 445–448. https://doi.org/10.1145/2771839.2771875

[5] Tom Beckmann, Daniel Stachnik, Jens Lincke, and Robert Hirschfeld. 2023. Visual Replacements: Cross-Language Domain-Specific Representations in Structured Editors. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Cascais, Portugal) *(PAINT 2023)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/3623504.3623569

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

[7] Philip T. Cox and Simon Gauvin. 2011. Controlled Dataflow Visual Programming Languages. In *Proceedings of the 2011 Visual Information Communication - International Symposium* (Hong Kong, China) *(VINCI '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/2016656.2016665

[8] Véronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, Bertrand Melese, and Elham Morcos. 1983. Outline of a Tool for Document Manipulation. In *IFIP Congress*. 615–620.

[9] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 505–518. https://doi.org/10.1145/1094811.1094851

[10] Conal M. Elliott. 2007. Tangible functional programming. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/1291151.1291163

[11] Enso International Inc. [n. d.]. Enso. https://ensoanalytics.com/.

[12] M. Erwig and Bertrand Meyer. 1995. Heterogeneous Visual Languages—Integrating Visual and Textual Programming. In *Proceedings of Symposium on Visual Languages*. 318–325.

[13] Elliot Evans. 2023. Polytope editor. https://elliot.website/editor/.

[14] Riley Evans, Samantha Frohlich, and Meng Wang. 2022. CircuitFlow: A Domain Specific Language for Dataflow Programming. In *Practical Aspects of Declarative Languages: 24th International Symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings* (Philadelphia, PA, USA). Springer-Verlag, Berlin, Heidelberg, 79–98. https://doi.org/10.1007/978-3-030-94479-7_6

[15] Edward Foster. 1985. Outliners: A new way of thinking. *Personal Computing* May (1985).

[16] Paul Frenger. 2003. The JOY of Forth. *SIGPLAN Not.* 38, 8 (Aug. 2003), 15–17. https://doi.org/10.1145/944579.944583

[17] Alex Fukunaga, Wolfgang Pree, and Takayuki Dan Kimura. 1993. Functions as Objects in a Data Flow Based Visual Language. In *Proceedings of the 1993 ACM Conference on Computer Science* (Indianapolis, Indiana, USA) *(CSC '93)*. Association for Computing Machinery, New York, NY, USA, 215–220. https://doi.org/10.1145/170791.170832

[18] Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 71, 16 pages. https://doi.org/10.1145/3586183.3606817

[19] Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *J. Vis. Lang. Comput.* 7 (1996), 131–174. https://api.semanticscholar.org/CorpusID:11750514

[20] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. 2012. End-user Debugging Strategies: A Sensemaking Perspective. *ACM Transactions on Computer-Human Interaction* 19, 1, Article 5 (May 2012), 28 pages. https://doi.org/10.1145/2147783.2147788

[21] A N Habermann and D Notkin. 1986. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering* 12, 12 (Dec. 1986), 1117–1127. http://dl.acm.org/citation.cfm?id=15550.15552

[22] Ian Hellström. 2016. The problems with visual programming languages in data engineering. https://databaseline.tech/the-problems-with-visual-programming-languages-in-data-engineering/.

[23] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

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

[25] 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

[26] Michael Homer. 2023. Branching Compositional Data Transformations in jq, Visually. In *ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*.

https://doi.org/10.1145/3623504.3623567

[27] Michael Homer. 2023. Multiple-Representation Visual Compositional Dataflow Programming. In *Programming Experience Workshop*. https://doi.org/10.1145/3594671.3594681

[28] Michael Homer. 2024. In-line Compositional Visual Programming. In *Programming Experience Workshop*. https://michael.homer.nz/Publications/PX2024/InlineCompositional-Homer2024.pdf

[29] Michael Homer and James Noble. 2013. A tile-based editor for a textual programming language. In *Proceedings of IEEE Working Conference on Software Visualization (VISSOFT'13)*. 1–4. https://doi.org/10.1109/VISSOFT.2013.6650546

[30] Michael Homer and James Noble. 2017. Lessons in Combining Block-Based and Textual Programming. *Journal of Visual Languages and Sentient Systems* Volume 3 (2017). https://doi.org/10.18293/VLSS2017-007

[31] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 72, 18 pages. https://doi.org/10.1145/3586183.3606733

[32] Joel Jakubovic. 2024. *Achieving Self-Sustainability in Interactive Graphical Programming Systems*. Ph. D. Dissertation. University of Kent.

[33] Joel Jakubovic and Tomas Petricek. 2022. Ascending the Ladder to Self-Sustainability: Achieving Open Evolution in an Interactive Graphical System. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Auckland, New Zealand) *(Onward! 2022)*. Association for Computing Machinery, New York, NY, USA, 240–258. https://doi.org/10.1145/3563835.3568736

[34] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (mar 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[35] Szymon Kaliski, Adam Wiggins, and James Lindenbaum. 2019. End-user programming: Data pipelines. https://www.inkandswitch.com/end-user-programming/#data-pipelines.

[36] Marcel R. Karam, Trevor J. Smedley, and Sergiu M. Dascalu. 2008. Unit-level test adequacy criteria for visual dataflow languages and a testing methodology. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 1 (oct 2008), 40 pages. https://doi.org/10.1145/1391984.1391985

[37] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

[38] 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

[39] Žiga Leber, Matej Črepinek, and Tomaž Kosar. 2019. Simultaneous multiple representation editing environment for primary school education. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 175–179. https://doi.org/10.1109/VLHCC.2019.8818927

[40] David H. Lorenz and Boaz Rosenan. 2011. Cedalion: A Language for Language Oriented Programming. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 733–752. https://doi.org/10.1145/2048066.2048123

[41] Yoshiaki Matsuzawa, Takashi Ohata, Manabu Sugiura, and Sanshiro Sakai. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 185–190. https://doi.org/10.1145/2676723.2677230

[42] Philip J. Mercurio. 2009. Thyrd: An Experimental Reflective Visual Programming Language. https://thyrd.org/thyrd/paper/.

[43] R. Mark Meyer and Tim Masterson. 2000. Towards a better visual programming language: critiquing Prograph's control structures. *J. Comput. Sci. Coll.* 15, 5 (apr 2000), 181–193.

[44] Roland T Mittermeir, Markus Clermont, and Karen Hodnigg. 2005. Protecting Spreadsheets Against Fraud. In *EUSPRIG*.

[45] Charles Moore. 1999. *1x Forth*.

[46] 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

[47] Brad A Myers. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing* 1, 1 (1990), 97–123.

[48] James Noble and Robert Biddle. 2002. Program Visualisation for Visual Programs. In *Proceedings of the Third Australasian Conference on User Interfaces - Volume 7* (Melbourne, Victoria, Australia) *(AUIC '02)*. Australian Computer Society, Inc., AUS, 29–38.

[49] Mark Noone and Aidan Mooney. 2018. Visual and Textual Programming Languages: A Systematic Review of the Literature. *Journal of Computers in Education* 5, 2 (2018), 149–174.

[50] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

[51] Raymond R Panko. 2000. Spreadsheet Errors: What We Know. What We Think We Can Do. In *EUSPRIG*.

[52] Jon Purdy. 2012. Why Concatenative Programming Matters. https://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html.

[53] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67.

[54] Robert Schaefer. 2011. On the Limits of Visual Programming Languages. *SIGSOFT Softw. Eng. Notes* 36, 2 (mar 2011), 7–8. https://doi.org/10.1145/1943371.1943373

[55] Marc Schmidt. 2021. Patterns for Visual Programming: With a Focus on Flow-Based Programming Inspired Systems. In *26th European Conference on Pattern Languages of Programs* (Graz, Austria) *(EuroPLoP'21)*. Association for Computing Machinery, New York, NY, USA, Article 6, 7 pages. https://doi.org/10.1145/3489449.3489977

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

[57] David Canfield Smith. 1975. *Pygmalion: a creative programming environment*. Stanford University.

[58] Scott B. Steinman and Kevin G. Carver. 1995. *Visual Programming with Prograph CPX* (1st ed.). Prentice Hall PTR, USA.

[59] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. 2014. Interleaving of Modification and Use in Data-driven Tool Development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/2661136.2661150

[60] The MathWorks, Inc. 2022. Simulink. https://www.mathworks.com/products/simulink.html.

[61] Franklyn Turbak, David Wolber, and Paul Medlock-Walton. 2014. The design of naming features in App Inventor 2. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 129–132. https://doi.org/10.1109/VLHCC.2014.6883034

[62] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 41–61.

[63] Manfred von Thun and Reuben Thomas. 2001. Joy: Forth's Functional Cousin. In *Proceedings of the 17th EuroForth Conference*.

[64] David Weintrop. 2016. *Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms*. Ph. D. Dissertation. Northwestern University.

[65] David Weintrop and Nathan Holbert. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 633–638. https://doi.org/10.1145/3017680.3017707

[66] Jack Williams, Nima Joharizadeh, Andy Gordon, and Advait Sarkar. 2020. Higher-Order Spreadsheets with Spilled Arrays. In *European Symposium on Programming*. https://www.microsoft.com/en-us/research/publication/higher-order-spreadsheets-with-spilled-arrays/

Received 2024-04-25; accepted 2024-08-08