Multiple-Representation Visual Compositional Dataflow Programming

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

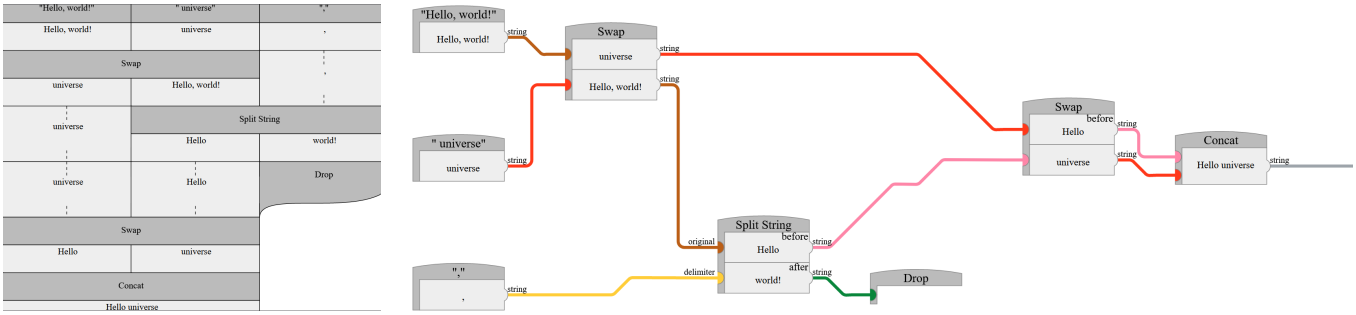


Figure 1: The same simple program in two of the three representations of our system. It can be converted to and from each view at will, as well as to textual code, with smooth animation between, and edited in all three modalities.

ABSTRACT

Many tasks that end users want to accomplish with a computer program are fundamentally data-flow transformations, and both visual and textual programming systems have been created to fill this need, but these are often inflexible, unapproachable, or cumbersome, satisfying a niche at one stage of the process but limited at others. An approach that suits one part of the program, or one time in its development, may be confounding at another, but the user is stuck with both the constructive and obstructive aspects of a tool's chosen paradigm throughout. Much of this difficulty can be removed by enabling the cohabitation of multiple editing paradigms in a single program for the user to choose how to tackle the current point in the process - and change their mind. We present a new data-flow programming environment where the same program, or parts of the same program, can be viewed and edited as linear text, a node-and-wire graph representation, or a two-dimensional grid layout, and the correspondence between these representations is made clear through a continuous visual identity for each part of the program.

CCS CONCEPTS

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

KEYWORDS

visual programming, dataflow programming, end-user programming, concatenative programming

ACM Reference Format:
Michael Homer. 2023. Multiple-Representation Visual Compositional Dataflow Programming. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '23), March 13–17, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3594671.3594681

1 INTRODUCTION

Most users of computer systems are not programmers, and do not want or need to be, but many would like the computer to do things for them that are not possible with the tools they have. They would like a system to be slightly more featureful, to have functionality for a slightly more specific use case, or to automate a part of a process they currently do by hand. Often these tasks are fundamentally data-flow transformations: taking some pieces of information and mechanically transforming them into a new structure or deriving new data from them.

A number of end-user programming tools have been created to fill these needs, most popularly the spreadsheet, but also numerous visual programming environments aiming to relieve the burden of syntax supposed to be a main obstacle for non-programmers. Many of these have had some success, but also well-travelled drawbacks: spreadsheets are error-prone to the point of having an entire conference dedicated to that (EuSpRIG), while visual languages are often seen as cumbersome to use and reflecting a narrow scope of use cases [13, 31], and both require a level of abstraction to use *well*

‹Programming› '23, March 13–17, 2023, Tokyo, Japan

that is often not compatible with how the user thinks about the problem at hand [11, 21, 25, 32].

A user often thinks in terms of values rather than processes, conceiving of the task in terms of the values in front of them and only generalising afterwards. They often find conforming to textual syntax challenging, with the computer rejecting their attempt on seemingly-arbitrary grounds. Much of programming training is about seeing through abstractions, precisely because intuiting parallels between similar — but not quite identical — operations is so alien. The user conceives of the task in terms of their own domain knowledge, which may not break down in the same places and ways that a programmer would define it.

Various models have addressed different of these issues:

- Visual programming environments have advantages in avoiding syntax, and depicting the dependencies between parts of the program, but are generally seen as cumbersome to edit and difficult to follow in even medium-sized programs [31], while nonetheless making very explicit the flow of data [4, 32].
- The textual paradigm that most directly expresses data flow is concatenative programming (where arguments and return values are passed implicitly), but these are often thought of as "write only", relying on hidden abstraction to connect points in the process, although they also have minimal syntax.
- More recently, the author proposed a grid-based representation for concatenative programming, and slightly more general data-flow, that depicts data connections and values through layout and is edited by selecting displayed values [16], but it still has inherited limitations on ordering, despite enabling slightly more expressive programs.

Each of these has advantages and disadvantages, but many of them are complementary: accessing the strengths of each paradigm can cover for weaknesses of another, if only it were possible to use them together or in succession when the occasion suited.

This paper presents a system that allows the user to **express data-flow programs in any of these formats**, with a value-first interaction model, and to pivot between using them at any time. To ensure that the user understands the connection, relationship, and isomorphism between these representations, changing view presents an **animated transition** where each element of the program has a continous visual identity throughout: the cells of the grid slide to form the words of the text or the nodes of the graph, with wires and values taking the same path. The visual representations display intermediate values, and these **values are tangible**: the program is edited by manipulating the concrete values, rather than beginning with abstract concepts of functions or transformations. They may also be rich, rendered in a native format for values such as colours or images so that the true data may be seen in-place. The system permits both "map"ping and "filter"ing operations to be integrated in one to match the user's or domain's mental model of the task, and both inspecting and evaluating the program at many input values innately. We argue that this multiple-representation values-primary approach allows end users to work at the correct abstraction level for them in the moment, without committing to it for all time, and even enables professionals building pipeline programs to focus on the abstractions they need to, rather than satisfying imperative bookkeeping requirements.

The contributions of this paper are:

- A design for a system permitting editing a program in three very different modalities and switching between them at will, aimed at value-focused end users.
- A semantic design fostering data-flow pipeline patterns that are not easily expressed in existing systems but correspond to typical domain needs.
- A prototype implementation that runs in a web browser.

The next section sets out some guiding principles of this system, and Section 3 describes the three representations and the transitions between them. Section 4 introduces some of the pipeline semantics supporting the intended programs, and Section 5 discusses the prototype. Section 6 discusses related work, and Section 7 reflects on experience using the system and concludes.

2 GOALS AND PRINCIPLES

There are a few guiding principles in play that we will set out briefly before describing the system itself. Some are obvious and others more speculative, but for the purposes of this experimentation we will assume that they are true.

First, that data and values are the most significant aspects of the program to the (intended) user, and should as much as possible be visible, tangible, and foregrounded: no work should be required to display the values and they should be the focus of interaction.

Second, that different perspectives on the same code are useful, either for a "change of pace" or because the specific task, domain, or author aligns more naturally with one viewpoint or another. Use of multiple views has shown value in block-based systems previously [6].

Third, that the way the user conceptualises and breaks down a task is often not aligned with the way trained programmers would, with tasks a programmer would break apart or coalesce seen as atomic or unrelated.

Finally, that reproducibility is aided by explicit dependencies in step-by-step development. A problem that arises for end-user programming is that it is often easy to obtain a result once, but in such a way that there is not a path to applying the same process to other data because the code is transient or hyperspecialised — or worse, it *looks* reproducible, but latent state in the development process made it not so, as is common in notebook-style environments.

The impact of these principles is seen through the next sections.

3 MULTIPLE REPRESENTATIONS

Our system allows viewing and editing programs in three different modalities, both for different parts of the program code, and for the same code at different times. Each is discussed in its own section below, but in short they are:

- Concatenative linear text, where the program or function body is a sequence of identifiers in the manner of a conventional concatenative language; discussed in Section 3.1.
- Interleaved 2D grid layout, where function calls and the values they operate on alternate vertically, with data dependencies identified through layout; discussed in Section 3.2.

- Node-and-wire graph format, where functions are represented as nodes that display their name and output values, and have connection ports that data dependencies run as wires between; discussed in Section 3.3.

Both first and last are fairly conventional, while the grid layout resembles previous work [16]; a key feature, however, is that both non-linear representations display concrete intermediate values and invite direct interaction with them. The user can switch between these views of their code at will, and an animated transition gives each element of the code a continuous visual identity throughout. It is always possible to switch to a later view in that list, while each gains slightly more expressivity and can express some programs that the previous cannot, so switching upwards is often, but not always, possible. These transitions are discussed in Section 3.4.

The user can define multiple functions, each given their own separate tab in the editing environment, and these can call each other, or be used as the entry point of a data-processing pipeline (see Section 4). It is possible to write functions in all three modes and use them all at once. This mutual compatibility implicates some design choices of each representation, discussed below.

3.1 Linear Concatenative

Concatenative programs pass arguments implicitly, and so the program is simply a sequence of words identifying which function is to be called next; commonly, functions take their arguments off a global stack, and push their return value(s) to the same stack for the rest of the code to use[1]. The term "concatenative" arises because two programs can be composed — using the output of one as the input to the other — by appending them together [37]. In this way they directly express data flow, and encourage breaking the program down into smaller pieces that can be composed together.

Programs in this form can be cryptic, though also largely "syntax-free", consisting only of a list of names, with no special characters or other constructs; Figure 2 illustrates a trivial example program. In our system, this modality provides quick entry and editing of programs, using conventional textual affordances. It is the most rudimentary of the representations used, and most useful for either very simple functions or long linear pipelines of one-to-one transformations. Concatenative languages are often seen as "write-only" and cryptic, but we combat this in our system by accepting it: to read the program, there are two other views that make the functions, their effects, and the connections between them very manifest, trading away some concision for much greater clarity.

[1]Non-stack-based approaches are also possible [20], but are not treated here.

```
"Hello" length 2 mul
```

Figure 2: A trivial program in the linear representation. Here, the literal "Hello" is a nullary function that pushes its value to the stack, where the length function will consume it and push 5 in its place. The literal 2 similarly pushes its own value, and mul takes both of these and pushes their product, leaving the stack with the value 10 at the end of the program. All of this is implicit, relying on the user knowing the arity and role of each function.

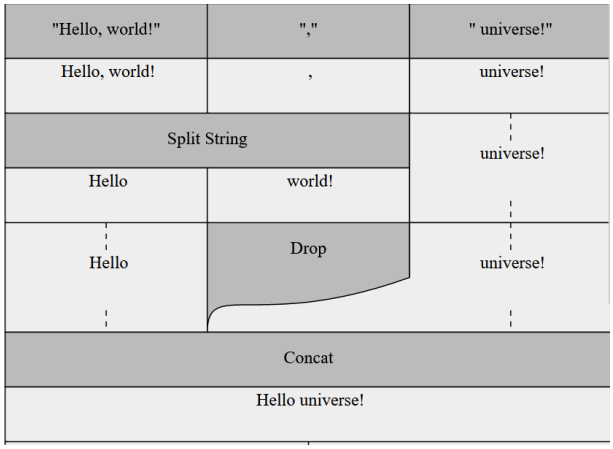


Figure 3: A simple program in the grid representation in our system. Evaluation runs vertically, with functions (dark grey) below argument and above return values (light grey). The "drop" zero-output function illustrates new capability in this version of the grid representation that the original work did not support. Its function cell tails away into the left of what would have been its output, while the value to its right expands to fill the space.

3.2 Grid

The grid representation interleaves functions with the values they operate on, with each function laid out below its arguments and above its output(s), building on previous work [16]; the core of this view aligns with that previous work where it is discussed in more depth. Alternating rows of functions and values are laid out in a grid, with functions spread out below the full width of their arguments, and their return values splitting the space below the function. An example of the representation is seen in Figure 3. The argument and return value cells render their values inside, so that the concrete values and the results of operations on them are directly visible whenever possible. To edit the program, the user can drag to select a horizontal range of value cells, and a menu will present a list of functions that can operate on those values to add to the program below them. The process is thus focussed on the values in use, rather than considering the operation to perform first, and an exploratory style of seeing what can be done with these available values is encouraged.

The current system includes some extensions to the grid representation in previous work, and in particular is able to represent zero-output functions, which are important for expressing the range of pipeline operations we wish to support in this work. A function with no outputs has no output cell in the row below, and instead the function cell extends into the next row, reducing to a point in the lower-left corner. Any value cell to the right will expand into the space vacated by the nullary function, and will be treated as occupying the full horizontal space of both its natural area and the area below the nullary function. An example of this in action is seen in Figure 3 with the "drop" cell in the middle.

This view can represent any program from the concatenative mode perfectly and provide an alternative view on it, which can

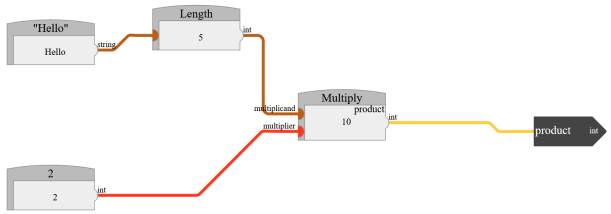


Figure 4: A trivial program in the graph representation. Here, evaluation occurs left-to-right: the "Length" function's input port on the left is connected to the "Hello" string literal's output port on its right with the brown edge, and the "Multiply" function node is connected to both the "Length" output and the "2" number literal's output with the red edge. The yellow edge runs to the depicted function's sole output on the right-hand side.

also be edited and switched back to the linear concatenative view. The additional dimension also permits expressing further programs, however, which are more difficult to express in the concatenative mode. For example, in a concatenative program only the values at the top of the stack can be operated on, but in the grid modality a function may be laid out below any values, without the fiddly stack-manipulation operations required for this in concatenative programs. This extra power is convenient, but a function doing so will *not* be able to be switched back to the linear view where it cannot be represented, though the function will be available to use from any view.

3.3 Graph

The graph representation is akin to a conventional node-and-wire visual programming environment in the dataflow vein (as opposed to control-flow languages), but with a strong focus on displaying the data values being operated on. A node represents a function, with slots for individual argument inputs on the far left edge. The values produced by the function are displayed within the node, in the same fashion as in the grid representation, in their native format. Output wires run rightward from these values, forming a complete data-flow graph. A trivial example is depicted in Figure 4.

In this system, edges always flow from left to right: the graph layout will ensure that all of a node's dependencies precede it and all of its dependants follow it, with those above and below having no direct relationship. This requirement has both a usability and a semantic basis: it ensures that the sequence of operations is always clear and minimises the convoluted wire-crossing that can occur in other graph-based systems, and it also provides manifest layering that is used for pipeline and flow diversion semantics discussed later in Section 4. The user will be unable to move a node to a position that would break this constraint.

The user can click on any output value to add a new edge to the graph, and choose any compatible input port further right in the graph to connect to. The prospective wire will be displayed at the mouse pointer, and an already-occupied target input port will be disconnected.

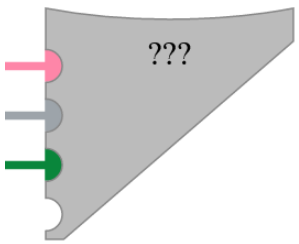
Instead, the wire can be dropped *mid-screen* to add a new function to the program. A new node for the pending function will appear, and further wires can be added to it as well, seen in Figure 5a. Once all the values to be used are connected to this pending node, the user can select a function able to process those inputs in place and it will substitute itself for the placeholder. They can also choose to create a new function of those inputs, which will spawn a new tab for editing that function.

An output can also be connected on the far right edge as an output of the entire function. An output can be connected to multiple inputs, but an input can only be connected to a single output.

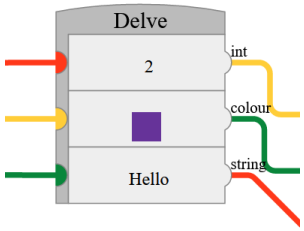
A few other design decisions were made to accommodate the other representations of programs. The wire colouring is not semantic, simply assigned to each new edge in turn to make them distinguishable. However, while programs created in the graph view may cross wires at will, those native to the concatenative or grid views will use explicit reordering operations such as "swap" or "delve" to get values into the right order for the next function. The graph renderer is aware of such functions (identified by the shape of type parameters used) and will preserve the wire colour in the output, as shown in Figure 5b. This permits following the colour along the graph to see the true connection, just as if the wire crossing had been drawn directly, while still representing the program faithfully. Ordinarily, an output edge will be a fresh colour from any of the inputs, unless the function node has exactly one input and output.

Again, the graph view is able to represent any program from the concatenative or grid modality with a direct data-flow view, and the program can be edited in this view and switched back to the original view. Again, however, there are also programs that can be expressed in the graph modality that do not have equivalents in the other representations without explicit reordering operations within the program. For example, any wire crossing has no parallel within the grid view, as all used values must be sequential and in order, while any output value is used exactly once and cannot be consumed by two different functions.

While editing programs in the grid view and maintaining linearisability is often done incidentally, compatibility is liable to be



(a) A "pending" node in the graph. Three input wires have been connected, with an additional slot always available. Clicking the node shows a menu of functions with those inputs to convert into.



(b) The "delve" function is used for stack reordering in concatenative mode, cycling the deepest value to the top. Here this is the red wire. The other input/output colours match up also.

Figure 5: Two graph nodes with multiple connections to them illustrating different features of the representation.

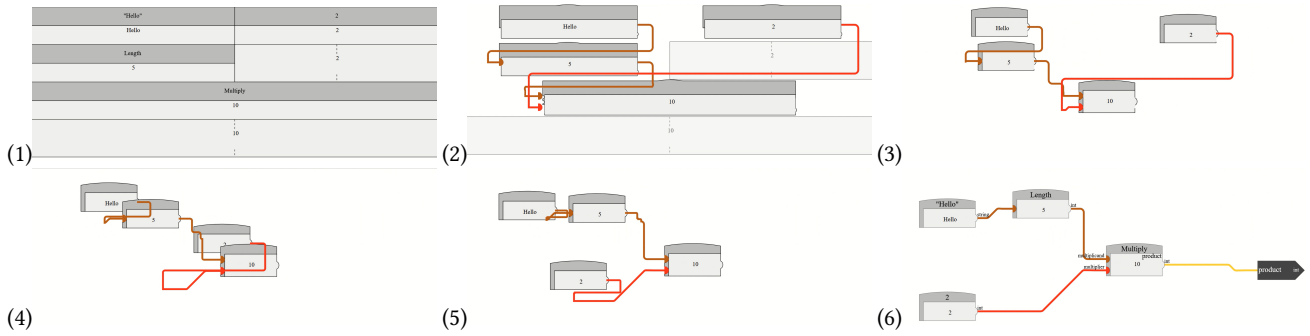


Figure 6: Snapshots of the transition between the grid and graph views. Images (1) and (6) correspond to the program from Figure 4, while the steps in between show the transition between them.

broken inadvertently in the graph view due to the substantially wider degrees of freedom provided. Any program in the graph modality can use functions defined in either of the other modalities, and the reverse is also true. In this way, a part of a program that requires additional flexibility can be implemented in the graph, while the bulk can use another approach, or the graph can be used for the high-level architecture while individual transformations are implemented using a different mode.

3.4 Transitions

A selector in the interface allows the user to choose "Text", "Grid", and "Graph" views of the function in the current tab, with the present one selected by default. When the user chooses another view, an animated transition occurs taking the corresponding elements of the present view to their equivalent in the new view. This animation takes approximately one second, so as to strike a balance between being fast enough to be unobtrusive and slow enough to be understandable.

Precisely how the transition occurs depends on the two views involved. The most substantive is between the grid and graph views, the two with the most components. Figure 6 depicts points in the transition: starting in the grid view (1), the function cells morph to resemble the shape and dimensions of the graph nodes, while the value cells shrink to match the output components of the graph nodes, and edges from the current values to their consuming functions appear as the value pass-through grid cells fade away (2). Reaching the final shape of the graph nodes at the half-way point (3), they are now only in the wrong location, and (4-5) begin to move towards their eventual positions, with the edges adjusting correspondingly to continue to connect the nodes. Finally, the transition is complete (6), and an additional edge for the function output appears, as this output is implicit and latent in the grid but explicit in the graph.

Switching in the other direction is similar, following the reverse order.

For transitions to and from the linear textual representation, as only the function names are used in this representation, the values are not represented, and the connections are implicit, only these are relevant and participate in the animation. However, the direct relationship between the grid and textual representations does allow for a smooth ordering of this movement at the function

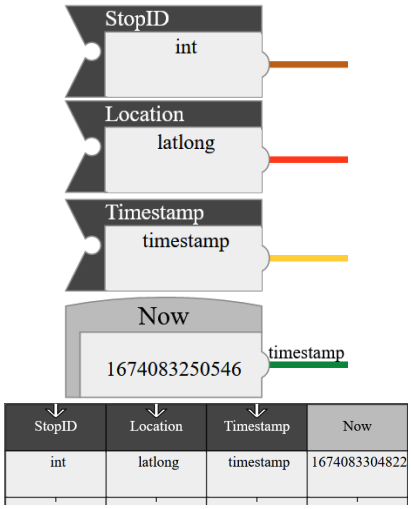


Figure 7: A function with parameters in both graph (top) and grid (bottom) view. A further zero-parameter function, "Now", is called in each and is displayed aligned with the parameter entries.

level, with function names moving to fill out the grid from top-left down and right, and vice-versa for the other direction, while to and from the graph they move simultaneously.

4 PIPELINES

Thus far we have looked at the system in terms of how programs are displayed and edited, but another goal is to allow the code to be reused to process many data items.

This section discusses the semantics of our system with respect to data pipelines. These semantics are to large extent severable from the program representations discussed in the previous section. However, both have also been co-designed to complement each other, so the behavioural effects align closely with the visual representations.

As far as possible, we do not want the user to need to think about the mechanics of the pipeline beyond the basic data flow that all our representations use. Any parameterised function within the

system accepts a sequence of (typed) arguments, analogous to a row of input data in a pipeline, regardless of the representation used to create that function. A function of three parameters is seen in both visual representations in Figure 7. The user can thus write their code to deal with a single set of values, and it can be slotted in as a pipeline component consuming such a (partial) row of data. That is, any function can in effect be "map"ped across a stream of incoming data.

Within the function, any desired computation can be performed, and another row of output data is produced. For example, a function may take two parameters: a string for a person's name and a date of birth, and output the name and the person's current age. This function would call another function internally to obtain the current date, perform the date arithmetic, and pass the name through unchanged. This function could be used within a pipeline with a stream of names and dates, but the user need only consider the one pair of inputs, and in our grid and graph views can even see them in situ.

4.1 Loading Data

The prototype implementation allows interactively importing a comma- or tab-separated value (CSV or TSV) file to provide input rows. Doing so will automatically create a function, in the grid view by default, that takes the same number of parameters as there are columns in the file, using column names and inferred types as available. The user can edit this function as they wish, and scrub through these rows of arguments to observe their effects on the output. This function can still be used directly from any other function by providing appropriate arguments, or all of the resulting outputs can be computed and displayed at once, or sent on for further processing.

Conceptually we imagine the system being able to be connected to external data streams, but the prototype runs entirely in a web browser and thus cannot do this. Both pre-determined batches of data and live streams would be possible, with no substantial difference to the user.

Because this system makes all data dependencies explicit, each evaluation of a function on a row of data is independent of all others. We view each original row of input data as conceptually a separate thread of execution, so loading a CSV file with 100 rows will create 100 threads whose results are collated at the end, or sent on for further processing later in the pipeline.

4.2 Stopping Threads

We have seen how projecting functions across a stream of data can be useful. Real-world datasets, however, are often incomplete, partially invalid, corrupted, or otherwise problematic, and data cleaning, wrangling, and imputation may be required. For example, one such issue in our name-birthdate-age example from above might be that some expected dates are missing or in an invalid format. A number of approaches are possible there, but a common one is to discard rows with null entries, potentially logging their occurrence. Another case where rows may be discarded is when the date is in the future, which is likely to be an error, or when performing analyses that require excluding minors from any further processing.

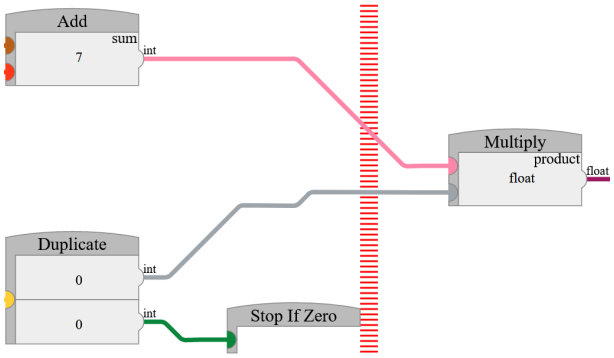


Figure 8: A thread-stopping function triggering in input values, in the graph view. The computation does not proceed to subsequent functions, seen where the "Multiply" function has no output value displayed, and the dashed line indicates the threshold beyond which no computation occurred for this set of inputs.

Any case where rows are not passed forward no longer fits the "map" model, and is a pattern that is challenging to express in typical data-flow environments. The conventional functional approach to such a situation is to compose multiple maps and filters, but these can be challenging for users who are not programmers to create[2]. The two or three parts created may not correspond to a clean transformation within the problem domain, or the user may need to repeat precursor steps in both places.

Our model instead allows the user to terminate processing of this row at any point: they do not need to think of two separate operations, but can instruct the system to stop based on whatever criteria they choose, and no further processing will occur (neither in later steps of a pipeline, nor within the same function). In this way, the desired domain semantics can be written as-is, and there is no need to repeat the same logic in a separate filter nor to break the transformation up.

This "stop" instruction is where the layering seen in the graph representation, and inherent in the grid representation, comes into play. All steps in the same layer are independent of one another, and conceptually executed in parallel (in our implementation, their substeps may be interleaved, but no code will actualy execute simultaneously). No step in a later layer is executed until all in the previous layer have completed, and if the evaluation is stopped in that previous layer, execution will not proceed. This ordering is core and directly visible to the user, avoiding a semantic ambiguity identified by Muhammad [25] in other visual languages where identical-looking programs can have a meaningfully different order of execution. This creates a clear point where the user can decide to stop processing, and a space where the visual representations can indicate the "high-water mark" of a given execution when illustrating concrete values. When rendering concrete data values, the point of stopping can be marked, as shown in Figure 8 for the graph view.

[2]Another typical functional solution here is to point at the list or option monad, but "abstract nonsense" may not be helpful to the user already struggling with abstractions.

The prototype includes a number of convenience functions for this purpose, such as "Stop If Null" and "Stop If True", but the user can also create their own. If one function calls another that stops processing, the first function will also stop, so user-defined domain-specific discard criteria can be abstracted into named operations as needed.

4.3 Stream Operations

There will be cases where the user needs to perform operations on the entire sequence of data, such as sorting. These operations do not work with individual data items or even rows, and so do not fit within what we have discussed so far. To support them, we allow functions to be used at both the ordinary "row" and stream level.

A user-defined function with data attached to it (e.g., from a CSV) can be called in two ways: providing arguments, to run it as written on a single row of data as we have seen so far, or with none, to produce a single output encapsulating the stream of that data as processed through the function. Stream-level operations such as sorting threads (by the top of their stack), finding the maximum-valued row, or aggregating the data can be performed on this value.

These are best used in a zero-parameter function acting as the entry point for the whole program, but the system does not enforce this. The user can edit with the same affordances as elsewhere in the system, and using any of the representations. The system will display the eventual rows as produced by that stream-level pipeline.

Any function with matching parameters can also be used at the stream level to project that stream through the function, so pipelines can be built up from smaller pieces. Different functions in the same program may operate at either level.

5 IMPLEMENTATION

The prototype implementation is available at http://ecs.vuw.ac.nz/~mwh/px23/, and works in a web browser. No installation is required, but source code is available from https://github.com/mwh/mrvcdp. The code is written in TypeScript and the rendered program representations are a dynamically-created SVG. Programs execute client-side, using promises for each component asynchronously. An eclectic array of built-in functions is included for experimenting, but the in-browser execution model is not suitable for handling large amounts of data. Some pieces of functionality have varied browser support, notably the animated transitions, which have almost no support in current versions of Safari due to the lack of SVG CSS Path support, while some other aspects have minor functional or aesthetic differences.

No data is transmitted or saved from the client, although some provided functions access third-party APIs (such as Wikipedia). The file chooser in the top right allows loading in data from a local CSV or TSV file, which is processed in the client, and populates a drop-down list of data rows to evaluate the program on.

6 RELATED AND FUTURE WORK

Animated transitions between block and text representations have some currency in block-based programming environments, originating independently in Tiled Grace [17] and Droplet/Pencil Code [1]. Simultaneous multiple-representation block environments such as Poliglot [22] also exist; both animated and simultaneous approaches have difficulties with temporarily-invalid intermediate source code states, which our system largely avoids, although there are some *permanently* incompatible programs with certain views. Projectional editors such as Cedalion [23], Gandalf [12], and Mentor [7] are an antecedent to multiple-representation environments such as here, presenting both usability challenges and enhanced affordances to programmers [36]; one has notably been built for the concatenative language Forth [14]. Generally these involve editing an abstract syntax tree or similar representation, and then projecting it into a concrete syntax, but they can also allow for other visualisations.

Hazel [29] is a live, functional, structural programming environment with data-flow elements. Code is evaluated around "holes" in the program, and the editor displays the results of these evaluations, but code presents as a conventional applicative language. Moore's colorForth [24] is arguably a visual AST editor, but has no other representations.

Several node-and-wire visual languages exist, including those using the graph edges for both control- and data-flow[19, 25]. Examples include LabVIEW [8, 28], Simulink [35], Pure Data [3], Yahoo! Pipes, Unreal Blueprints, and others [9, 32]. A number of musical tools including both physical and programmatic systems also follow such an approach [27], and make results *audible* live. Our system's focus on displaying values is distinct from most of these, and no system we are aware of supports multiple representations in the manner of this work.

Muhammad [25] analysed the semantics of widespread end-user dataflow languages, including both spreadsheets and visual systems. We draw on some insights from this analysis in the behavioural design of our modalities, notably in establishing a visible ordering of execution and maintaining display of values.

Spreadsheets are the most widely-deployed programming method showing intermediate values [21], and use both specialised textual syntax and spatial references. Several spreadsheet-derived systems extend this paradigm with additional representations, types, and visualisations [2, 10, 30], retaining the core spreadsheet editing and input cycle. Userland [26] is an integrated dataflow environment incorporating both spreadsheet cells and Unix shell pipelines, displaying intermediate states. Systems such as Natto [33] provide a cards-on-canvas aesthetic for working with principally conventional code, equipped with some data-flow connections between cards and convenient renderers. The Vivide system [34] is a live dataflow programming environment where interactive widgets are scripted with Smalltalk, composing programs out of their connections. All of these approaches provide some level of dataflow and reactive live programming, with (most) data values visible and available for further use, though many also have latent values that are never accessible (such as *within* a spreadsheet cell calculation).

This project is a continuation of the author's previous work on 2D representations of concatenative programs [16], value-driven visual programming [15], and transitioning multiple-representation environments [18], although no implementation is shared.

6.1 Future Work

In addition to the three representations described in this paper, there are other representations just beyond the edges of this system that

could be incorporated directly. The textual view is currently just text, but a block-based editor also makes sense and fits in between the plain text and grid models. This would allow for higher-level affordances of the sort provided to the more-visual representations, and could transition to standard text in exactly the manner of Pencil Code and Tiled Grace. On the other end of the scale, the graph view presently is quite constrained. A liberalised graph even without any semantic addition could permit more non-semantic layout use (albeit this is precisely the cost affiliated with general node-and-wire systems), or one that decoupled the values from their producing functions more could allow for a still-more-value-based system in the manner of our previous work Calling Cards, and user-defined dashboard layouts. The model also admits additional modalities that are not necessarily interconvertible with the existing ones, provided that they can slot into a pipeline with arguments and outputs. Bespoke interfaces for expressing functions with particular purposes, such as to perform filtering or type conversion [5], or to present visualisations of datasets, could be added and used from programs as others are. Further experimentation and user studies will help to refine the concepts presented here.

7 REFLECTIONS AND CONCLUSION

First, some assorted reflections and observations of the author from using the system during and after the development process:

- The grid representation was the most-used middle ground, though commonly all would be engaged in a single program, rather than the entirety converging on one approach.
- Switching to any other view became reflexive whenever there was a pause to consider the next step.
- A first draft or principal pipeline initially dashed off as text before being refined in another view was a common pattern.
- The main entry point very often settled in the graph view, even when not required.
- Most functions with substantial edits in the grid, but not graph, view remained compatible with other views.
- At times, easy graph editing was something of an attractive nuisance, inviting convenient but incautious changes that unnecessarily blocked use of other views for that function.
- Graph *viewing*, however, was helpful for concatenative and grid functions containing multiple stack-manipulation operations, where edge colouring could be followed through the program.
- Any function with a "stop" gravitated towards living in the graph view except utility "stop if [domain-specific condition]" helpers, which were rarely looked at again.
- Tabs remembered the mode their function used, but sometimes matching the previously-viewed tab would have been less jarring.

7.1 Conclusion

We have presented a design and prototype for a programming environment aimed at users who are not (intending to be) programmers supporting data-flow pipelines and focusing on the values under examination rather than functions or methods. It enables them to view and edit their program in three very contrasting forms to suit the particular task at hand and their current mental model. Understanding of the connection between these views is assured by giving each element a continuous visual identity and animating transition between representations, and the semantic model aims to allow component boundaries to match with the requirements of the domain. We hope that this concept illustrates the potential for a new approach to programming environments, and that the prototype will be useful for exploring the design space further.

REFERENCES

[1] 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

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

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

[4] 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

[5] Alexis De Meo and Michael Homer. 2022. Domain-Specific Visual Language for Data Engineering Quality. In *ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. https://doi.org/10.1145/3563836.3568727

[6] Wenbo Deng, Zhongling Pi, Weina Lei, Qingguo Zhou, and Wenlan Zhang. 2020. Pencil Code improves learners' computational thinking and computer learning attitude. *Computer applications in engineering education* 28, 1 (2020), 90–104. https://doi.org/10.1002/cae.22177

[7] 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.

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

[9] 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

[10] 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

[11] 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

[12] 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

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

[14] Ulrich Hoffmann. 2019. Forth Projectional Editing. In *EuroForth 2019*.

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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] Charles H. Moore. 2009. Chuck Moore's Wonderful colorForth Programming Language and OS. https://colorforth.github.io/.

[25] 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

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

[27] 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.

[28] 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.

[29] 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

[30] 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

[31] 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

[32] 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

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

[34] 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

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

[36] 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.

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

Received 2023-01-22; accepted 2023-02-06