In-line Compositional Visual Programming

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

ABSTRACT

Concatenative programming inherently expresses composition of sub-tasks of a pipeline, but one uncommonly seen model of this paradigm includes all data values inline within the program. A visual environment for editing and evaluating programs in this model would inherently display state in place, and allow for easy tracing of data flow through the program by watching the values literally move as evaluation steps took place. We propose a visual approach for programming in this style, with function calls and data values interleaved on a single "track", with specific concrete arguments always adjacent when a function term is evaluated and various affordances for editing, evaluating, and debugging. We then show how extensions to this model to multiple tracks can ease programming in the model and even make available some more inscrutable programming-language features, such as concurrency and effect systems, in a more accessible way.

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. 2024. In-line Compositional Visual Programming. In *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/yey-to-be-provided

1 INTRODUCTION

Both concatenative and visual programming languages naturally express data-flow pipelines, but are often seen as cumbersome to use or difficult to understand and debug. However, these paradigms have complementary strengths despite their contrasting styles, and a fusion can gain the benefits of both. This fusion will make some uncommon choices on both sides. In particular, here we propose following what has been called the "prefix-concatenative" model: the program consists of a sequence of mixed function calls and data values, and evaluation replaces a function call whose arguments are all available with its result. This approach naturally leads to a visual model where the view of the program during editing, and of intermediate states during evaluation or debugging, is identical; evaluation can be single-stepped, with partial results taken or preserved as their own editable functions; and bespoke visual representations of the transformation steps included to aid understanding.

The contributions of this paper are:

- A visual programming environment built around an inline-compositional concatenative language.
- Extensions to this model involving parallel tracks of execution in various ways.
- A prototype implementation that runs in a web browser.

The next section sets out background on concatenative languages, visual programming environments, and data-flow pipelines.

Section 6 discusses related work, and Section 7 reflects on experience using the system and concludes.

2 BACKGROUND

The "concatenative" paradigm refers to the ability to concatenate the source of two subprograms and produce a program representing the composition of the operations of the two. Slightly more generally, the term *compositional* is sometimes used to refer only to the semantics [12].

The most common form of concatenative language is the *stack-based* language, where functions pop their arguments, and push their return values, on a latent stack, and are evaluated left-to-right in a postfix notation. This is the model used by Forth [18]. Any function consumes zero or more stack values and produces zero or more stack values, and two functions used consecutively will then have some or all of the results of one be consumed by the next; all functions can have any arity in both directions, and the next function need not match that. While conventionally analysed as imperative stack manipulation, these are equivalently seen as a functional model, as in Joy [30]. A concatenative, or compositional, language thus uses juxtaposition to indicate composition of functions, where typical "applicative" functional languages use it for function application.

A useful property of any concatenative language is free abstraction: by nature, any linear subsequence of terms in the program can be spliced out and replaced with a call to a new function whose body is exactly those terms, with no further modification needed on either side. In this way, they encourage factoring out not just common subexpressions, but any subsequence of terms that is useful to name, and foster a style of programming that favours breaking apart the problem into very small, composable pieces whose functionality is clear from their name. They also by nature express

‹Programming› '24, March 11–15, 2024, Lund, Sweden

chains of operations applied sequentially to inputs simply, and so can make a good fit for data-flow pipelines, which are a significant area for end-user programming.

Although concatenative and stack-based languages are often identified with one another, there are both non-concatenative stack-based models and non-stack-based concatenative models, and this paper is interested in the latter, in particular an approach we will call "inline" concatenative programming. In an inline concatenative language, concatenating subprograms still composes them together, but the data values are not hidden on a stack, and instead included inline within the program source, interwoven with function calls. A program is thus a sequence of terms, each of which is either an operation (function call) or a value. When evaluating a program, a single step either either

- identifies some operation term whose arguments are all present and adjacent as values, and replaces that subsequence with the results of executing that function, or
- expands a function call to the sequence of terms that was defined as its body verbatim, which in a concatenative model gives an identically-behaved program.

For primitive operations, only the first of these is possible, while for user-defined functions either is available. Unlike an applicative system, such as the lambda calculus, but like all other concatenative systems, the operations in this model do not form a tree structure. Any operation may have multiple return values, and another operation may consume only some of those, automatically leaving the remainder for other operations; this "arity-neutral" composition is a distinguishing feature of concatenative programming, but is most visible in an inline approach. An existing realisation of this model is found in the Om language [6], which describes itself as "prefix concatenative" (as it does not use the typical postfix ordering of common stack-based languages), but more significantly it follows an inline-compositional approach. A formal treatment of a minimalistic calculus language with this model has been given for the Kihi language [12].

There are a number of interesting properties that arise out of this *inline concatenative* paradigm: there is no hidden state, all values are visible in the program, any evaluation step produces a valid program, and input values (or values deriving from them) move steadily from right to left. These properties make it an interesting candidate for a visual programming environment, which can make use of those visual traits and expose affordances for manipulating and debugging the program that are not available in a conventional text editor, and which can expose the strengths of the programming model for some tasks to users more easily than either expressing those tasks in more common programming paradigms, or using one of the notoriously-abstruse concatenative languages for their problem.

3 DESIGN

At a very high level, the system we propose is a visual programming environment for data-flow programs, structured as follows. Following the inline-compositional model, this system will display a program or function definition as a row of interleaved operation (function call) and data (constant value) cells. The data cells display the value they contain, with suitable affordances for manipulating them and suitable visualisations. When editing, sequential subsequences of cells can be selected and abstracted into new named functions spliced in their place. Operation cells can be expanded inline to their definitions, and new operations or values inserted between any two cells. The operations are typed and the system will reject invalid compositions, while allowing invalid states to exist while editing, with placeholders where further changes need to be made. Programs or functions can be evaluated in a single-step mode to debug or understand them (or for some other purposes discussed later), which will make a single reduction of an operation whose arguments are all present, either expanding to the function's definition, or resolving it to the return values of the function. Figure 1 shows the steps of a trivial example of this evaluation process. Because these partially-evaluated programs are self-contained free-standing programs with all their data dependencies included, they can also be saved, edited, and refined. In this way, a user who is uncertain about the behaviour of a program, or the implications of a change, can see it in operation with concrete values in situ, revise it speculatively, and continue.

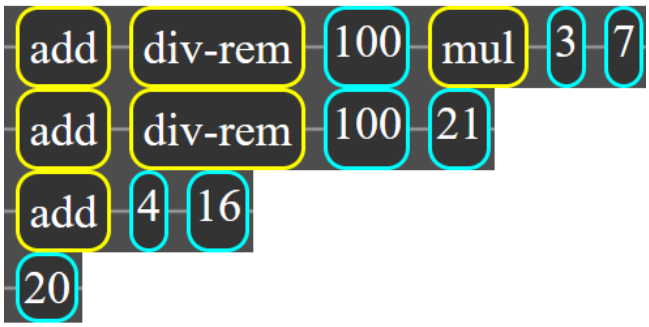

Figure 1: Evaluation steps of a simple mathematical expression as seen in the system. Each row is the successor of the previous; the div-rem function is returning two outputs.

3.1 Editing the Program

The user can drag to select any subsequence of terms — including the empty subsequence between terms or at either end — and splice it out for a replacement chosen from a menu of options:

- A newly-defined named function with that subsequence as its body.
- A chosen operation or value with a matching type for that location.
- Nothing, removing the subsequence entirely.
- The sequence quoted as a literal value.
- The result of evaluating the subsequence as far as possible internally.
- One single step of evaluation, most useful for expanding a function call to its definition.
- Another operation that **is not** type-compatible. In this case, an error marker will be displayed on the side or sides that requires additional terms, and the user can insert them as desired.

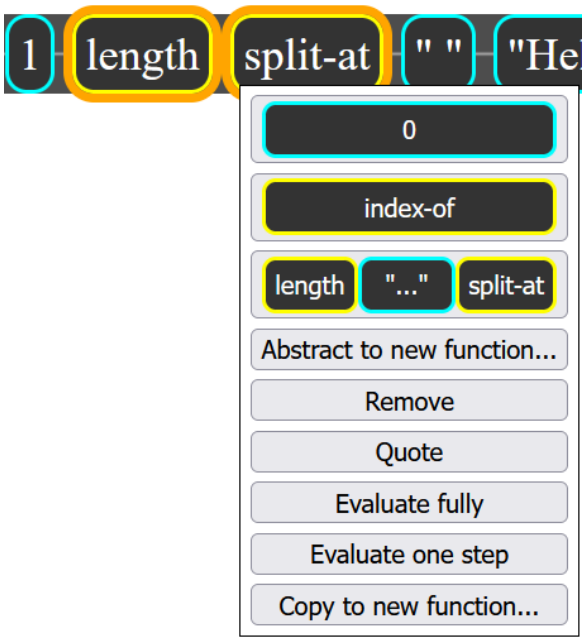

Figure 2: A (trimmed) menu of splice options for a selected subsequence. The selected terms have a thick orange highlight, around the yellow border of the two operation terms. The value terms, with blue border, are not selected and will not be replaced by the selected menu operation.

- Leaving the sequence as-is, but extracting it into a new function definition; this can be for generalisation or specialisation, or to allow interactive debugging.

Figure 2 shows a version of such a menu, trimmed for space. It uses type information about the selected subsequence to determine which options are available to replace it, in order to avoid overwhelming the list with irrelevant options. Additional options are plausible, such as pre-generated combinations of operations including argument-reordering or other standard transformations that allow inserting operations that are almost type-compatible but for ordering. Similarly, chaining multiple operations on the same sequence is feasible and potentially useful. Suitable interactions to surface these other options without overwhelming the user remain future work.

3.2 Evaluating

Conceptually, these programs could be deployed as standalone pipelines, or embedded in another system, but for the moment we only consider evaluation within the visual environment itself. Any function definition can be selected for evaluation, including the "whole program". On being evaluated, the function body is copied into another area, with the original definition left intact. The user can select to evaluate fully immediately, producing the resulting values, to take a single step and pause, or to evaluate fully but pause briefly at each step before continuing automatically. When there are no evaluable terms left, the evaluation is complete; this could be because there are only values left, or because all values have moved to the left edge of the program and any remaining operation terms do not have all their arguments available.

Whether stopped or paused, the same edits can be made as to a static function definition and evaluation continued. The current program state can be saved into a new function definition, or even be set to replace the original. In this way it is also possible for the programmer to create functions designed to produce *other* function definitions through partial evaluation, specialised with certain starter values.

Regardless of the stepping mode chosen, the program evaluates by choosing an operation term that is followed by as many value terms as needed to satisfy its arguments, and replacing that term with the result of evaluating the operation with those values. For non-primitive functions, these evaluate by replacing the operation term with the body of the function, with the arguments left in place. For irreducible primitive operations, such as numeric addition, the operation and its arguments are replaced with the result immediately.

The selection of which operation to evaluate is to some extent arbitrary: for non-side-effecting code, the outcome is the same for any order of evaluation, as in the Kihi model [12]. The current prototype always selects the leftmost evaluable operation term, but both choosing the rightmost term and more sophisticated models are possible. For example, given the expected role of interactive stepping and debugging, it may be best in practice to choose the operation with the largest or smallest number of arguments available, or to prioritise primitive operations over function expansions. It is not obvious what the best approach is, so the current prototype makes a simple choice and leaves the question open for further exploration and user studies.

3.3 Visual Representation

A single function is a list of terms. In display, this is a single horizontal track of cells. Operation terms are displayed as a box with the operation name yellow border, while value terms are displayed as a box with the value and a blue border. Figures 1 and 2 show both types of term. Compound values, such as lists and dictionaries, are stacked vertically within the box. Some values may have a more visual visual representation naturally, such as images, or may have a suitable visualisation available. These visualisations can be decoupled from the rest of the system, as they are encapsulated within the value cell, but in some cases they could offer interactive affordances for direct manipulation of data as well. However, interactions could interfere with selection and editing operations at the program level, so require caution, and the present prototype does not include them.

When a term is to be evaluated, it is highlighted along with its argument terms first. The new terms are then displayed in its place, with a short animated transition to make the relationship clear.

The origin of expanded terms is recorded and can be displayed as a highlighted bar below the terms, with multiple overlapping bars as applicable. This gives a tree (or really, a directed acyclic graph) representation of the origins of each term that was not in the original definition, but it takes up significant space and is not displayed by default. These highlights have some interactivity, allowing to select the full subsequence of remaining terms or to

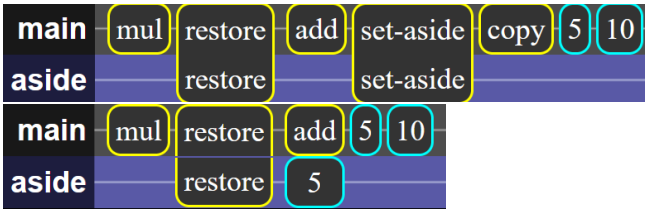

Figure 3: A simple example of a bypassed operation, where "set-aside" will move its argument to the second track below, and "restore" will bring it back, allowing `add` to use the two arguments below and `mul` to access the set-aside value along with the result of the addition. Top: the initial program. Bottom: the program state after `set-aside` has executed.

point at the location within the original definition, but no further functionality is available at present.

4 MULTIPLE TRACKS

So far, the system has been described as evaluating one pipeline, with a single "track" of terms. However, the visual layout of the program is not constrained to a single row, and the model is not constrained to a single track. In this section, we explore the implications of multiple tracks, and how they could relate to one another, or help to make otherwise-obtuse or difficult-to-understand concepts or program structures more accessible. We will consider a few different mutually-incompatible extensions and their potential uses and trade-offs; it is not clear which if any of these candidates is worth the complexity, but all have proven viable in test implementations.

4.1 Concurrent Tracks

The simplest extension to the model is to allow multiple tracks to be evaluated in parallel. These would operate independently, with their own scheduling, reduction, and data, but be displayed as literal parallel tracks.

4.2 Higher-Order Functions

Higher-order functions like `map` and `filter` apply some operation to each of many values in a collection. A `map` has an obvious visual representation with multiple tracks: for each item in the collection, a new track is created containing a copy of the expression to be mapped, and the single data value at the end. These tracks can then proceed as usual in parallel, with the same single-stepping and inspection capabilities in the visual environment. There is a direct visual indication of the processing of each item, including the intermediate steps of the transformation.

4.3 Bypassing Operations

A typical annoyance in a concatenative language is Byzantine stack manipulation sequences to process a value that is "below" another before it. This generally involves some sequence of rotation operations, sometimes with complex quoting and unquoting as well, or in simple cases a primitive "dip" operation that evaluates a quotation with the top value temporarily held off-stack. A secondary

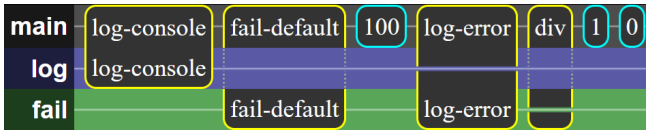

Figure 4: Three tracks, with every possible combination of tracks appearing in at least one function. `div` may produce an error on the failure track, but is not on the log track and is seen "going behind" it; `log-error` produces a log message on the log track for every error it sees, and passes on the error, relaying any unrelated log messages it sees transparently. `fail-default` uses the default value of 100 when an error has been produced in evaluating its arguments; `log-console` handles log messages. Functions that are not on a given track are transparent to values passing by them there, but the horizontal space is left empty to keep a clear alignment of terms.

track could allow for values to be "set aside", moved to that track to expose the values below, and then restored at another point.

A function on the principal track could send its argument into the other track, and multiple calls would queue up multiple values there. Another function "restore" would take the next value from the other track and put it back on the principal track. Figure 3 depicts a simple example of this, where "set-aside" appears across both tracks: when executed, it will produce nothing on the top track, but output 5 on the bottom track. Conversely, "restore" will receive that 5 and produce it on the top track. The purpose, order, and progress of this operation would be explicit and visible to the user in a way that paired rotation operations are not, and it is particularly suitable for animating or single-stepping to understand the program.

4.4 Error Track

Some operations may fail at run time based on their input values or an external problem; a trivial example is division by zero. These errors should be reported by the operation, but a conventional pipeline or functional program does not offer a clean way to do so: either the error must be reported as a value, meaning that the rest of the system must be prepared to unpack and relay a variant type, or the reporting must jump outside of the ordinary execution flow. Under this model, there can be a separate "error track" in parallel with the main body: a function that fails emits an error value onto that track instead of its normal output. That track can have its own code to process the error case, and will also evaluate in the same way as the rest of the system.

4.5 Effects

The preceding section described a way to address failed function executions, but it approaches a more general solution as well. We can consider a failable operation (such as `div`) as being a cross-track function, existing on both the principal track and the error track. When it evaluates, it is able to produce an output on either track.

We can consider other cross-track functions that do not necessarily produce errors themselves: for example, a function `decorate-errors` could have two definitions:

- On the principal track, `decorate-errors <note> <arg>` simply evaluates to its second argument.
- On the error track, when an error argument is available, it augments that error with the note provided on the principal track and produces the augmented error.

This allows for code to specify how errors should be handled within the evaluation of its arguments, and we could similarly have functions for logging the errors to different destinations and other purposes.

This can push one step further: when producing an error, a reference to the producing function can also be put onto the error track. That reference can be used for reporting purposes, but it is also plausible to allow the error-handling code to direct the system to substitute some other function or values in its place. In this way, a "default" value could be inserted in place of an anticipated error: `default-on-error 99 div 1 0` would produce an error, have it trapped by the handler, and replace the `div` call and its arguments with `99`. Given the "splicing" evaluation model of the whole language, enabling this is fairly trivial. In effect, this is a sort of limited one-shot delimited continuation.

This all need not be restricted solely to errors, however: we can have arbitrary tracks, and functions can be permitted to cross any of those they choose, returning values on any or all of those tracks. Figure 4 shows a constructed system with three tracks "main", "log", and "fail". Log messages emitted on the log track will be consumed by the nearest log-handling function, errors sent to the failure track by the nearest error-handling function, and one function crosses all three to produce a log message out of an error. This is one concrete example, but the approach is general and any number of arbitrary tracks and track-crossing functions are possible.

What we have is the beginnings of an effect system, where effect-handling functions cross tracks to consume the side-channel communications from effectful functions, and potentially to return values back to them. The evaluation of these effects will follow the same rules as the rest of the system, and the visual representation will be the same as well. This may allow for easier comprehension of the sorts of "action at a distance" that effects can have. It is somewhat serendipitous more than an original design goal, but merits further exploration.

5 IMPLEMENTATION

The prototype implementation is available at https://mwh.nz/demos/px2024, and works in a web browser. It is written in TypeScript, with a simple set of built-in operations to explore the space, but it is not currently capable of sophisticated programs.

6 RELATED AND FUTURE WORK

Visual data-flow programming systems are relatively common models for domain-specific tools for end-user tasks; most often, these are graph-based. Examples have existed for decades, and include Pure Data [2], LabView [7], ProGraph [29], Yahoo! Pipes, and others [3]. These allow complex branching and merging, but usually focus entirely on displaying the computation structure clearly, without including the data values being operated on. Testing and debugging in these systems have been identified as difficulties [13, 16, 26], with changes to the nature of the input data leading to hard-to-analyse behavioural problems. The system proposed here ought to do better in that respect, but a trade-off made is that it is less expressive when computations must split into separate branches, whether or not recombined later, as the concatenative system will require "jumping" some values over other terms to reach the operations in one branch.

Subtext [4] is a quasi-visual programming environment representing data values inline with operations, and explicit links between different points of the program. As in this system, it is possible to inspect the execution of the program to see how it arrived at a particular state, parameters and variables are tacit and unnamed (semantically), a function call essentially inlines the function's definition, and manipulation is principally through pointer manipulation rather than textual input. However, Subtext does not present a concatenative model but something more akin to an applicative–graph hybrid, and its core model is of replication and aliasing; it is conceivable that programs in each model could be projected or converted the other.

Thyrd [15] is a purely-visual concatenative programming language and environment, where cells of a spreadsheet-like grid represent operations and values of the program, including recursive grids, all edited through direct manipulation. Thyrd uses multiple stacks to contain data values during execution, but values can be directed into visible cells also, which can in turn be used as the starting points of other operations; while intermediate results are not displayed by default, it is possible to chain subprograms together such that all values are shown (or to display the various stacks during evaluation). The evaluation and data models of Thyrd are unique, but aspects of the visual display and manipulation have similarities to other systems. Spreadsheet 2000 [**?**] was another spreadsheet-like system with graph-based connections and direct-manipulation programming, but not concatenative in approach.

Parès et al. [24] discuss constructing data-science workflow pipelines through composing functions, and particularly addressing effects within those functions. Their system is purely textual, but is semantically compositional and has some similarities to the multi-track extensions discussed in Section 4 in particular. Both static checking and optimisation are important facets of this "Kernmantle" system and are elements we do not address in the present work.

Moore's colorForth [19] is a limited visual structured editor for a concatenative language, and includes inline literal data values and some static evaluation options. However, it is fundamentally a syntax-tree editor for standard Forth, where word colour indicates the token class instead of symbols or keywords, and has no visual or interactive evaluation features.

Hazel [22] is a live, functional, structural programming environment with data-flow elements. Interactive editing widgets are available for inline data values and the program can have "holes" that are evaluated around, similar to the type-mismatch placeholders in the model of this paper, but it is intended to permit evaluating "around" holes in partially-complete programs, and holes represent only a single value or expression rather than unspecified computational steps. Conceptual visual interfaces for program terms date back at least as far as Pygmalion [28]. Elliot [5] introduced "tangible functional programming" in which values can be displayed as editable GUI elements, including executing functions. The user

can construct higher-level values by composing elements in these GUIs, using them as the principal editing interface.

Muhammad [20] analysed the semantics of widespread end-user dataflow languages, and this analysis influences some of the design choices in this system.

Spreadsheets are a common end-user dataflow programming environment, and have been the subject of much research into their use and misuse [9, 14]. They follow a rows-and-cells approach, display (some) intermediate values, and often have visual means for showing data dependencies (although it is not necessary to the model). Many extensions to spreadsheets with additional kinds of data, visual representations, and data dependencies have been proposed [1, 8, 25] and sometimes incorporated into commercial products. A common spreadsheet pattern is to have sequential steps of a calculation laid out horizontally, each depending on the previous cell and sometimes earlier ones as well, which gives a similar layout of calculation steps as here, but with each cell being responsible for both displaying and calculating its value. This pattern is a common source of spreadsheet errors, as some calculation steps can easily be missed, or unused, and this passes silently [17, 23]; the approach in this system ensures that the sequence is explicit and inspectable.

Userland [21] is a data-flow programming environment using both spreadsheet-style formulas and Unix shell commands, and including both spreadsheet-style cell dependencies and Unix pipeline compositions. The pipeline modality has implicit composition and displays the operation and its result in a cell, as here, but always tied together and displaying both. Systems such as Natto [27] provide a cards-on-canvas aesthetic for working with principally conventional code, equipped with some data-flow connections between cards and convenient renderers.

The limited previous work on visual environments for concatenative languages has focused on the latent stack model, rather than the inline model [10]. Follow-on to that work incorporating multiple representations [11] of the program, particularly a graph-structured editor, suggest a future direction for this system. Some of the multiple-track extensions would seem at home in a graph view, although it is not clear exactly how that would operate.

6.1 Future Work

The multi-track extensions are preliminary and raise questions about how they can be typed, how they can be optimised, and how they can be used in practice. In particular, there is not an obvious formal model that accommodates the range of multi-track functions given in the examples to ensure that each track is type-correct and that the system is sound: some evaluate to different numbers of values, or pass through values not their own, and it is only by the cooperation of all involved functions that this is not a problem.

The select-to-splice interface is seemingly critical for efficient use of the system, but no user studies have been conducted to determine how well it works in practice. While it is hoped that in-situ presentation of relevant options ameliorates the common complaint of visual programming environments being cumbersome to edit, it's also possible that other approaches would give better results, such as a drag-and-drop "toolbox" modality or keyboard-based editor.

7 CONCLUSION

Under-explored even within the under-explored concatenative programming paradigm is the inline style of structuring programs, but it offers intriguing possibilities for visual programming. The system proposed here is a first step in exploring that space, and has some interesting properties that are not available in conventional visual programming environments. The multi-track extensions are particularly novel, and offer a way to address some of the limitations of the single-track model, and to make some otherwise-difficult concepts more accessible. It is not clear that these are all practically useful concepts, but we have outlined the space to highlight that they are worth exploring further.

REFERENCES

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

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

[3] 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

[4] 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

[5] 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

[6] Jason Erb. 2021. Om website. https://www.om-language.org/.

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

[8] 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

[9] 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

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

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

[12] 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

[13] 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

[14] 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

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

[16] 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.

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

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

[19] Charles H. Moore. 2009. Chuck Moore's Wonderful colorForth Programming Language and OS. https://colorforth.github.io/.
[20] 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
[21] Hisham H. Muhammad. 2019. Userland. http://www.userland.org/.
[22] 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
[23] Raymond R Panko. 2000. Spreadsheet Errors: What We Know. What We Think We Can Do. In *EUSPRIG*.
[24] Yves Parès, Jean-Philippe Bernardy, and Richard A. Eisenberg. 2020. Composing effects into tasks and workflows. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 80–94. https://doi.org/10.1145/3406088.3409023
[25] 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
[26] 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
[27] Paul Shen. 2021. natto website. https://natto.dev/.
[28] David Canfield Smith. 1975. *Pygmalion: a creative programming environment.* Stanford University.
[29] Scott B. Steinman and Kevin G. Carver. 1995. *Visual Programming with Prograph CPX* (1st ed.). Prentice Hall PTR, USA.
[30] Manfred von Thun and Reuben Thomas. 2001. Joy: Forth's Functional Cousin. In *Proceedings of the 17th EuroForth Conference.*