Interleaved 2D Notation for Concatenative Programming

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

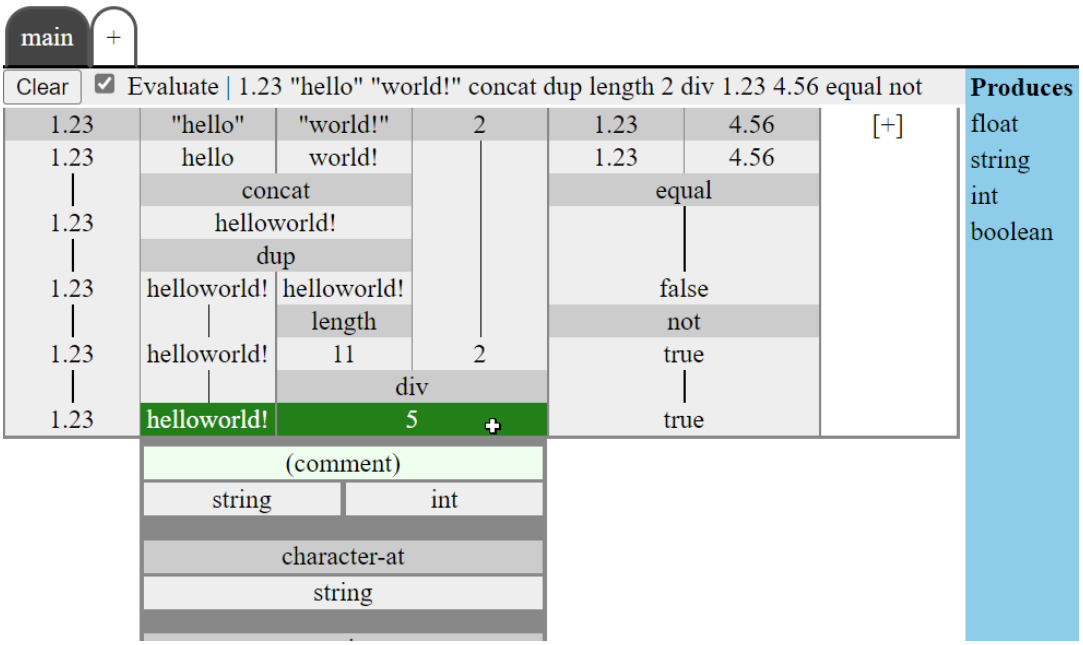


Figure 1. A small concatenative program in the midst of editing in the interactive two-dimensional notation prototype.

Abstract

Concatenative languages use implicit argument passing to provide a concise expression of programs comprising many composed transformation functions. However, they are sometimes regarded as "write-only" languages because understanding code requires mentally simulating the manipulations of the argument stack to identify where values are produced and consumed. All of this difficulty can be avoided with a notation that presents both the functions and their operands simultaneously, which can also ease editing by making available values and functions directly apparent. This paper presents a two-dimensional notation for these programs, comprising alternating rows of functions and operands with arguments and return values indicated by physical layout, and a tool for interactive live editing of programs in this notation.

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

Keywords: concatenative programming, visual programming, dataflow programming

ACM Reference Format:
Michael Homer. 2022. Interleaved 2D Notation for Concatenative Programming. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3563836.3568722

PAINT '22, December 05, 2022, Auckland, New Zealand

1 Introduction

Concatenative programming languages are those where two subprograms may be composed simply by concatenating their code: the output(s) of one program will automatically and implicitly be the input to the next [6]. A program is a sequence of named function calls, with no explicit arguments, named variables, or control structures. The most widely-known concatenative languages are Forth, PostScript, Factor, RPL, and Joy, all stack-based, but other approaches to concatenation that do not align with stack machines also exist, such as Kihi and Om [3, 12]. Most often, these follow a postfix notation, where each function called consumes its arguments from the stack put there by the previous functions, and pushes its outputs onto the same stack. In this way, concatenative languages are extremely concise for expressing pipeline transformations: appending a further transformation step is just appending another function name. They can be seen as imperative manipulation of a stack, composition of functions, or an expression of a data-flow graph, all true from a certain point of view, and foster a style of programming built around identifying and defining these transformational steps. However, concatenative languages are sometimes seen as "write-only" languages, as comprehending novel code requires knowing the behaviour and arity of the functions it uses.

In this paper, we present a new interactive representation of concatenative programs, where function inputs, outputs, and relationships are tangible and explicit. This representation is two-dimensional, displaying both the functions and the stack contents in order, and manipulated by drag selection. The values on the stack are foregrounded in editing operations, so it is always clear which are available and where they come from.

The contributions of this paper are:

- A notation for concatenative programs, where the functions and the stack contents are displayed in two dimensions.
- A model for interactive editing of programs in this notation.
- A prototype tool where the user can interactively edit a program, and see the effects of their changes in real time.

The next section gives background on concatenative programming for a general audience, and highlights some of the challenges this modality presents. Section 3 introduces our new two-dimensional notation for representing these programs. Section 4 describes the model for editing programs in the notation, while Section 5 illustrates a prototype tool putting that into practice. Finally, Section 6 positions this paper among related work, and Section 7 concludes.

2 Background

Languages like Forth aim to be close to a machine execution model, where the stack is a result of parameter-passing convention. Other concatenative languages, such as Joy, can be much higher-level, leaning more to the functional programming style [31]. Still others follow non-stack models entirely, such as Kihi [12]. In this paper we concentrate on the stack-based models, and will use a simple Forth- and Joy-like language to illustrate the concepts.

This section aims both to introduce an unfamiliar reader to the basic facets of concatenative programming, and to highlight the weak points that our interactive notation aims to address. A reader with familiarity with concatenative programming may wish to skip to Section 3.

2.1 A Toy Stack-Based Language

A program is a sequence of words. A word is the name of a defined function such as "add", or a literal value such as "15". A small program could thus be `15 7 add`. This program contains three words, each of which identifies a function.

The program is executed from left to right, processing each word one at a time. For each word, the corresponding function definition is found, and as many arguments as it requires are popped off the stack. The function is evaluated, and any outputs that it produces are pushed onto the stack. The next word is then processed in the same way, with the new stack values in place.

A literal identifies a nullary (zero-argument) function that pushes the value of the literal. The first step of executing this program is thus to push 15 onto the stack. Following that, the next function pushes a 7 onto the stack.

The next function, "add", pops two values off the stack, adds them, and pushes the result back onto the stack. The final result of 22 is the only contents of the stack at this point; we may want to further process the result, perhaps to halve it, or square it. We could append the program "2 div" on the end to halve the result: the "div" function takes two arguments, just like "add", but one will already be there. This is why these languages are known as *concatenative*. The steps of executing that program are as follows:

Remaining program	Stack (left is top)
15 7 add 2 div	
7 add 2 div	15
add 2 div	7 15
2 div	22
div	2 22
	11

Our program now is becoming unwieldy, however. We can shorten it by defining our own functions for some of these operations. Because all arguments are passed implicitly on the stack, we can simply splice part of our program out into a new function defined as exactly the words we have replaced, and put the new function words in their place:

```
halve : 2 div
square : dup mul
15 7 add halve square
```

In many respects, this is clearer than the long form, but we now have three functions that we must understand the input and output arity of in order to comprehend the program. There is more background knowledge required that is not apparent in the visible syntax of the program, in the way that more typical applicative and imperative languages clearly delineate function arguments and which values have been returned.

2.2 The Stack

A function always consumes its arguments from the top of the stack, expecting a number of values of particular types and meanings.

Sometimes the values on the stack are not in the desired order. For example, the parameter order of subtraction is semantically meaningful: it expects the subtrahend to be on top of the stack, and the minuend underneath it. A program may have produced the values in the opposite order, or code may be inside a function that uses a constant minuend and takes the subtrahend as an argument (e.g. distance-from-100). Stack languages will include stack-manipulation operations for these cases, or cases where a value is needed more than once. Common operations include "swap", which swaps the top two values on the stack, and "dup", which duplicates the top value on the stack.

These stack manipulations are sometimes necessary, but can also make the program execution more challenging to follow. In the worst case, reading the program turns into a game of Three-card Monte, where the reader must keep track of where the values on the stack came from.

3 A Two-Dimensional Notation

The linear expression of concatenative programs is concise, but not explicit about which arguments and outputs are in use or connected with which functions. It shows only the function calls, without any view of the expected stack contents or which values are being consumed as arguments, or where they came from. We can display our well-formed programs differently: as well as showing the functions in use, we can display their stack effects, and the dependencies between functions, in a two-dimensional fashion.

The representation we propose is a grid of cells, with each cell representing either a function or a value on the stack. Immediately below each function call, one cell will appear for each of its outputs, spaced across the width of the function. For each function that consumes arguments, it will stretch below the cells for all of those arguments. The rows thus alternate between functions and stack items, and data flows vertically downwards between functions.

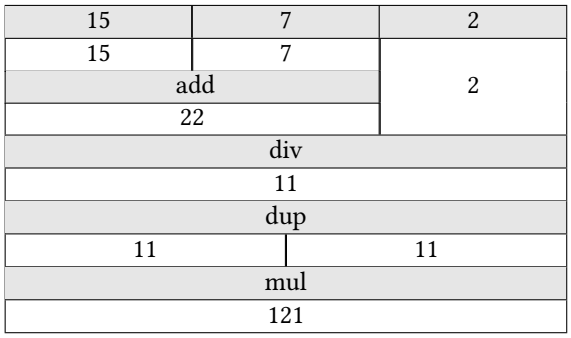


Figure 2. The program from Section 2.1, "15 7 add 2 div dup mul", in the two-dimensional notation. Functions are on a grey background and stack entries on white; execution flows top-to-bottom.

Revisiting our example program from Section 2.1, "15 7 add 2 div dup mul", we can see it in the two-dimensional notation in Figure 2. This illustrates several elements of this representation:

- The "add" function has two arguments and one output. It thus spans two cells above, but only one below.
- In contrast, "dup" takes *one* argument and has *two* outputs. The space below it is divided in two.
- All nullary (0-argument) functions are in the top row.
- The "div" function uses outputs of both "add" and the nullary "2". The stack item cell for the *value* "2" thus spans multiple rows.
- The available stack values are visible, in order horizontally, and data dependencies between functions are visible by following the table vertically.
- The "square" operation, "dup mul", is the last four rows from our table – it is still concatenative.

Within a single row of stack entries, all of the values are (or could be) available at once. Adjacent cells represent two values that could be given as arguments to a single function, regardless of where they come from.

Any program in the linear concatenative language can be transformed into the two-dimensional representation by simulating its stack and the data dependencies exposed between functions.

3.1 Stack Entries

Our examples so far have featured only integer values on the stack for concision and easy identity. However, the notation is generic in two ways: one is that it can show the *types* of argument and return values instead of concrete values. For many uses this may be the more common case, as the types are often more important than the values, or there are no known values at the time of writing. As each function has a known arity and type signature to the language implementation, these can be readily displayed at all times,

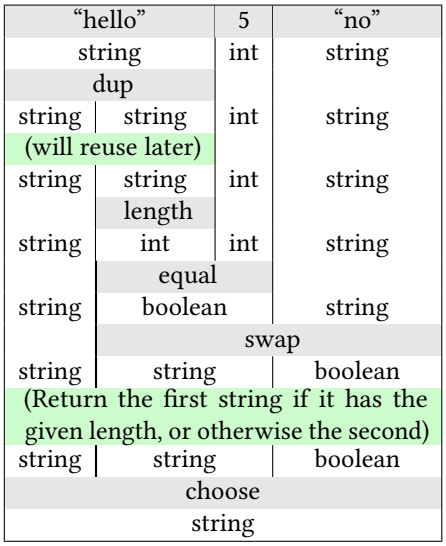


Figure 3. Representation of part of a program with types (white background) and comments (green background and parentheses) displayed.
The `choose` function returns its first argument if the boolean is true, and the second otherwise, so this program returns the string on the left if it has the given length, and the string on the right otherwise.

saving the user from needing to remember them or refer to the documentation. Figure 3 shows a program using various data types.

The other is that the notation can show *any* value — not only other typical base data types like strings and booleans, but more complex values like associative arrays, images, or graphs (of both kinds). These displayed values can be useful not just for debugging, but also for their own sake: the goal of a program may be to produce certain visual representations of a computation result, and they can be immediately and tangibly displayed complete with their provenance. This is a powerful way to communicate the results of a computation to a human audience, and to provide a visualisation of the computation itself. For space, we do not include an example of this here, but we discuss it further in Section 4

3.2 Comments

The two-dimensional representation is also useful for annotating parts of a program with comments. In this model a comment always applies to a range of stack entries, rather than a line of code, and this connection is made explicit. The comment acts as an identity function semantically and its text is displayed below the range of cells just like a function. Figure 3 shows a program with two comments (green background), one stretching across the entire row, and one applying to only two stack entries.

4 Interaction

With this representation, new editing interactions with concatenative programs become available. In particular, because the stack is visible, with all simultaneously-present items adjacent and in order, it is possible to identify and select a contiguous range of them in a single row. The system can then present the user with available options: functions able to consume these arguments, stack-manipulation operations, the ability to define a new function to process these values, and so on. A vertical or rectangular selection is also possible, to identify a composed chain of functional data dependencies that could be spliced out and abstracted into a new function.

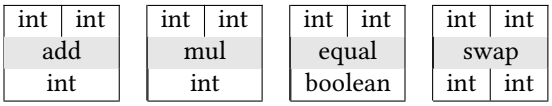
For editing, it may be useful both to display concrete values for stack items, and to display the types of values that are expected. Within a function, there may or may not be concrete values known, and in different circumstances either type or value may be more useful, so both should be available where possible.

4.1 Calling Functions

Each stack entry is in an individual cell of the grid. Contiguous ranges of cells represent adjacent stack items. A horizontal dragging operation can thus select a block of stack entries. If these items have not already been consumed by a function (that is, they do not have a function cell below them), and the rightmost argument is the rightmost output of another function (that is, it would be on top of the stack at this point in the linear program), then they are available to be transformed by a function.

Because the selection identifies exactly how many arguments there are, and what data type they are, the system can present the user with a list of functions that could make use of the selected items, as seen in Figure 1. This is different to the usual approach in both textual and visual editing paradigms of first specifying the operation, and then identifying the operands to apply it to.

For example, if the user selects a range of two integers, a menu of functions able to accept those arguments can appear:



Once the desired function is selected, the system can insert the new function call below the arguments and populate the grid with *its* outputs, ready for further selection.

4.2 Defining Functions

Alternatively, the user can select to create a new function to process these values: a separate editing view can then open up for this new subprogram, and a call to the newly-defined function be inserted. This subprogram has the same affordances as the parent program. The parameters are arrayed across the top, above the first functions of the body,

and whatever stack items remain at the bottom row are the return values of the function.

int	string
halve	length
int	int
equal	
boolean	

6	"hello"
halve	length
3	5
equal	
false	

If concrete values are available from the parent program at the point of use, they can be used to provide example values for the editing process, as in example-driven programming [2, 25], or the user can work with types alone. Additional nullary functions to produce values for use within the function can be added on the right-hand end of the top body row, producing their outputs after the parameter values.

int	string	
halve	length	false
int	int	boolean

The function cannot accidentally consume more arguments than intended, because the editor is presented only with the closed initial stack that was selected by the user originally: from the point of view of this function editor, no other stack items exist, and when adding a new function call the user can only select from available arguments. The return values of the function will be whatever is on the final row; these may be outputs of functions used above, or could be unused argument values remaining unchanged.

Because the editing operations change the function body in steps, the function may temporarily be in an inconsistent state. For example, the function may need to return a single value, but two are left on the stack at the bottom, because another function call combining the two values has not been added yet, or a previous one has been removed and is pending replacement. While the system can highlight this discrepancy for information, it is not a blocking error and the user can proceed to complete the function definition. Only when eventually saving the function does this inconsistency need to be resolved before continuing.

4.3 Replacing and Removing Functions

The cells for individual function calls can also be chosen, and there can be multiple options available depending on the circumstance.

Because both the input and output types are constrained at this point, other functions with the same signature can be presented to the user to choose from, and the original call replaced with the selection.

If the function outputs are not used anywhere, or are the same as its inputs, it can be removed entirely, and the argument values allowed to flow through.

If the function is user-defined, its body can be spliced into the program at this point in place of the function call. The body could then be edited to tailor it to the specific use case in play.

4.4 Stack Manipulation

The "swap" and "dup" operations, and other stack manipulation operations, can be made available in the same way as other functions. These will be available for *any* selection of stack items of the appropriate size.

The system can be slightly cleverer as well when the user selects a range of arguments. In addition to listing functions able to consume that sequence of values, it can offer multiple-function sequences involving stack manipulation. For example, if there is a "character-at" function wanting first a string and then an int, but the user selects a sequence of an int and then a string, the system can offer to swap the two items first:

int	string
swap	
string	int
character-at	
char	

Selecting this option would insert both functions into the program, adding multiple rows at once. This affordance can reduce the amount of manual stack manipulation required by the user, at the expense of listing more options to wade through in the situations where it is not desired.

4.5 Vertical Ranges

A vertical selection must mark out a pipeline of functions processing each other's results. There are subtle constraints on exactly what can be selected in this way. For example, it must not have a "ragged" edge on either side — it must consume all of the outputs of each function in the selection — and it must be as wide at the top and the bottom. However, a selection that does achieve this corresponds to a sequence of words in the linear representation of the program that could be spliced out and abstracted into a new named function.

4.6 Exploratory Programming

The affordances described above are designed to support exploratory programming, where the user is trying to find a solution to a problem by trying out different approaches [14, 26]. When concrete values are displayed, it is immediately apparent what data is available, and possible to try out different combinations of arguments to see which operations are possible with them.

These can be particularly useful when the data types in use are very rich. For example, if an item has a record type, the user can choose to display the evaluated program steps, and see the values of the individual fields of the record. They can then select one of those fields interactively, and have the system generate and insert a function that extracts that field.

Still-richer types, like "email" or "photo album" can also be exposed for direct programatic manipulation. Whatever operations are available on these, alone or in combination with other values, can be discovered, employed, and their

products observed and followed up on in the same way as any typical data type. *Displaying* the results of such exploration on such extremely rich values may be the point of the process, or they may be processed as steps of a larger task. Used in this way, there are parallels to the paths of Obenauer's "itemized operating system" [19].

Alternatively, the system could be configured to display data such as a sequence of numbers as a chart. Manipulating earlier functions in the pipeline (using replacement as in Section 4.3) or altering constant values could then change the rendered graph. Regardless, these rich values are integral parts of the program display; in this (sub-)approach, the program sits somewhere between a classical concatenative or dataflow program and a "notebook"-style system like Jupyter, or various literate programming environments.

4.7 Backtracking

Thus far, we have focused on the "forwards", data-driven direction: where the user has some values and wants to process them. The interactions we describe can also work with some modification in the reverse direction: rather than selecting a range of argument values, they could select a desired return type or types, and functions able to produce that type enumerated. In this case the program would grow upwards, choosing the final result stack type first, and then working backwards towards the original arguments.

For example, the user could start out desiring to output a string and a boolean. They could drag across both types to find any functions that produce those outputs, or select just the string to see what could create it, perhaps selecting a "repeat character n times" function. Now they would have a boolean, a character, and an int to satisfy, and could again select a range to see what available functions produce them, or to define a new function handling some part of the work.

This reverse direction follows exactly the same principles and affordances as the forwards direction, and potentially both could be in use at once (even simultaneously as a "meet in the middle" approach).

5 Tool Prototype

There is a prototype implementation of the system available online in a web browser at http://ecs.vuw.ac.nz/~mwh/demos/p22-2d-concat/. Figure 4 shows a screenshot of the prototype in action, editing the example program from Section 2.1. It includes full support for most of the affordances and features described in the paper for working with concatenative programs.[1] Areas of limitation include backtracking,

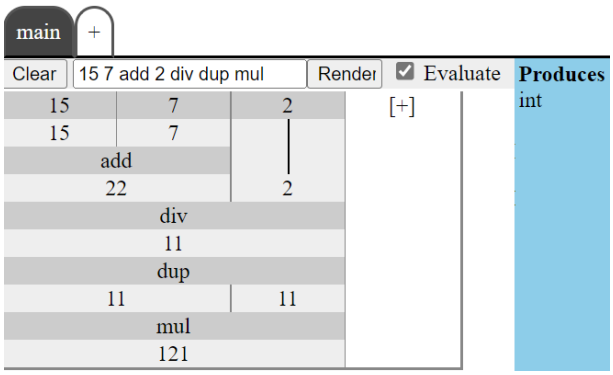


Figure 4. The prototype in action, editing the example program from Section 2.1. The lighter grey cells are values, selectable by dragging horizontally to choose a range of arguments. The darker cells are functions. At the top, tabs allow selecting different programs or functions to edit.

which is not supported, and vertical splicing, which works only in some circumstances.

The system permits entering a program in the traditional textual concatenative form and rendering it into the grid notation, using any of the pre-defined functions. It will animate the words moving from linear to grid notation and back in order to make clear the relationship between the two, in the Tiled Grace [10, 28] tradition of multiple-representation block-based programming environments. It also permits editing programs and functions within the grid format using the dragging affordances described, and adding nullary functions to the program with the plus button on the top row. Multiple programs and functions can be open at once, accessible via the tab bar at the top.

Dragging across type/value cells (that have not already been consumed by a function below) pops up a menu of available functions accepting those arguments, and selecting a function will add it to the program. Clicking on a *function* cell will produce a radial menu of options for that function, including replacing it with another similarly-typed function, removing it, or going to its definition as applicable.

The right-hand sidebar displays the types on the output stack, and the parameter types when editing a function. Red outline markings indicate a type error, most notably when editing a user-defined function that is in use elsewhere and has specific return types required. Elsewhere in the sidebar additional information relevant to system extensions under trial is displayed, but this is not relevant to the core described in this paper.

The grid display can be switched between displaying types and concrete values using the toggle below the tab bar. For complex types, this can make the per-cell display quite large.

A notable implementation limitation of the prototype appears in the case of extreme nesting, where a value is consumed by many layers of function having multiple outputs.

[1] It also supports other uses of the grid flow that are *not* compatible with conventional concatenative languages (and do not have a corresponding linear textual representation). We do not discuss these in this paper, but they result from relaxing some of the constraints described in previous sections, and type extensions, including some of the potential extensions from Section 6.1. Using the system with concatenative programs behaves as discussed.

For example, rendering the program "1 dup dup dup dup dup dup dup dup dup" results in the rendering of the first row failing to cover the full row. This is due to the use of HTML tables for the grid in this version, and the approach used for dividing cells when a function has multiple outputs: browsers do not allow a cell to span more than 1000 columns. With ten or more two-way splits below a single cell this limit is passed under the method in use. This is a limitation of the current implementation, and not a fundamental limitation of the approach. Practical programs rarely encounter this issue, but it is easy to produce pathological examples. We note it explicitly because it is otherwise mystifying to encounter.

6 Discussion and Related Work

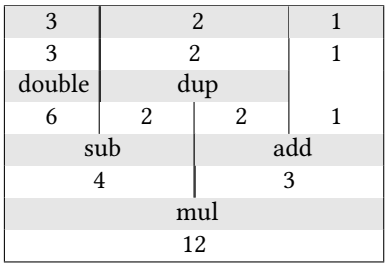
In this section we will discuss extensions and limitations of our approach, and position it amongst related work on both concatenative languages and visual programming.

6.1 Further Extensions

We have focused on support for traditional concatenative programs above, but there is some more generality in the grid approach. We note a couple of ways in which the system could be extended, but do not focus on them in this work.

While every linear concatenative program can be represented in the grid notation, there are grid programs with clear meaning that do not correspond to a linear program, because on the grid it is less crucial for values to be at the *top* of the stack to be operated upon.

For example, the following depiction has clear meaning:



However, the "sub" call is using two values that cannot be on top of the stack, as one of them came from "dup" and is below dup's other output. This program is *not* directly linearisable into a textual program the way that others depicted in the paper have been (without inserting stack-manipulation operations), but it is clear that it can be evaluated and expressed. The semantics of such programs are interesting future work, but for this paper we concentrate only on the classical concatenative programs, which correspond to those where where the rightmost *input* to each function is also the rightmost *output* of a function above.

Similarly, because the number of arguments consumed by a function is explicit, it is possible to overload functions by arity, avoiding the need for a series of distinct functions to be defined with different names to perform the same operation on different numbers of arguments, as in the "rot", "rot4", etc. functions in most implicit-argument concatenative languages. However, the utility of this is fairly limited.

Another extension is to lean further into the "pipeline" nature of these programs, and allow user-defined functions for data-parallel processing of collections, or of incoming events. While we have made initial explorations of what this can permit, it remains unclear just what it should look like or how it should work, and we do not discuss it further here.

Because programs are edited non-textually and with selections of contiguous elements, this approach may also have applications for programming on mobile devices, where the small screen size and touch-based input make text-based programming difficult. Preliminary experimentation with this possibility has been promising [9], but the space requirements are an obstacle. Hybrid or multi-device programming may be interesting avenues to investigate with this notation.

A practical system implementing this notation interactively need not limit "functions" to merely named references, but could have arbitrary configuration within the function cells. For example, a single function could include a drop-down list or text-entry field to genericise its operation, without needing to incorporate the selected parameter within the program at run time.

Because the grid inherently represents data flow, an alternative representation of the program as a graph is also possible, more akin to "node-and-wire" visual languages. Such a visualisation may be helpful for some users and use cases, and a multiple-representation approach could permit switching between the two, just as the grid and text representations are animated between in the prototype. A general graph permits a still more broad range of programs, beyond even the non-linear ones discussed above, so editing in the graph format will permit only a subset of programs to convert back to the grid format, but any program in this system could be transformed to a graph.

The final interactive extension is to be still further interactive, or in fact reactive, leveraging the data-flow nature to integrate external signals. Inputs to functions could arise from signals in the outside world, and outputs be wired directly to widgets or controllers without any impedance mismatch between the language behaviour and the reactive system.

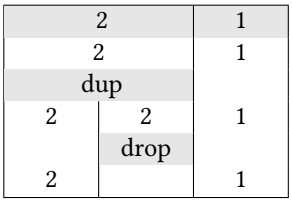
6.2 Limitations

There are a few limitations of this representation.

6.2.1 Nullary Outputs. A notable limitation of this representation is that it is not readily able to represent functions with nullary *output*, that is, functions which consume one or more values, but produce none. These functions do exist in concatenative languages: one use is functions called purely for side effects in languages which allow that, such as "print". Another is a very primitive operation often called "drop", which merely discards the item on top of the stack, and is

used when a function has produced a surplus output. The model does not accommodate these well. A nullary function would take up space, but leave a hole in the grid below itself, as there would be nothing to put in that space.

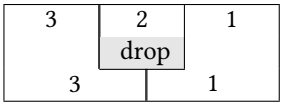
For example, take the trivial program "2 dup drop 1", which uselessly duplicates the value 2 and then discards it:



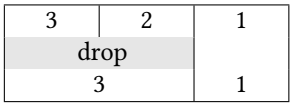
The values 2 and 1 are adjacent on the stack, and should be adjacent in the grid as well to be selected as argumetns to another function (such as "add"), but instead there is the space below the "drop" function where its output would be.

There are a few potential paths for dealing with this issue.

- One is simply to ignore it: while these operations do exist in some concatenative languages, the language of this model just does not include them.
- Another is to extend the stack item(s) on either side to fill the space, ensuring that the items that are consecutive on the stack remain consecutive in the display.

  This is inconsistent with other grid connections, but could be marked with diagonal shading or other means.
- A third is simply not to have such functions: instead of a "drop" operation that consumes one value and produces none, include only a function that consumes two arguments and produces the first as its sole output, having the same effect of discarding a single value (in effect, a Church Boolean).



All of these have reasons to follow them, and we will explore each in future. Our present prototype follows the third path, in part due to technical limitations of the rendering, but it has not thus far been an issue outside of theory.

6.2.2 Space. This layout requires a significant amount of horizontal space, sufficient to display at least all functions called on operands in the stack at the same time. With complex function dependency chains, the room required can even exceed this level.

For generic concatenative programs, this may be an issue. In particular, visualising execution by substituting function bodies for their calls (see Section 2.1) will rapidly require a very large amount of room. For the time being, we are focused on the *interactive* use of this representation for editing a program. Here, running out of room can be accommodated by abstracting part of the program as a new function, as encouraged in concatenative programming in general. We regard this nudge as, if not quite a feature, an acceptable cost for the enhanced clarity that this view provides.

For some uses, it may be sensible to transpose the grid layout presented here, so that functions are to the *right* of their operands, rather than below, and data flows horizontally. This would not just be a rotation of the required space because the minimum column width is determined by the size of the labels, which (presumably) continue to be horizontal for function names.

6.3 Related Work

Forth is the most widely-known concatenative language, albeit predating the term. There are many variants of Forth, often tied to specific machines and what the underlying hardware most directly supports. The simplistic nature of the language lends itself to experimentation, which has included projectional editors [7]. A notable variant in the context of this work is Moore's colorForth, which is a semi-visual language [16]: colour has semantic meaning in the language, so a word is defined by writing it in red, while writing it in yellow is a function call, for example. While this is a visual form of a concatenative language, it does not present the stack contents at all, and has little in common with our system or notation.

The term "concatenative" was brought to the fore by von Thun's Joy [31], which (unlike Forth) is focused on being a functional language, and introduced the term "concatenative" for describing this family. Joy includes an extensive set of combinators foregrounding this functional mindset, and employs the idea of function parameters and return values, but is still fundamentally represented as manipulating an invisible data stack. The toy textual language we created resembles Joy in spirit, but is not a direct translation of the original language.

Factor [23] is an object-oriented concatenative language, consequently supporting complex data types and values. Factor code is written textually, but edited and executed within a Smalltalk-style "image". It follows the same argument-passing style as our toy language and our notation is thus compatible with Factor, but rendering values in the interactive tool would be complicated by the presence of objects.

Stack-based concatenative programming is commonly used in genetic programming, due to its innate spliceability [8]. Languages have been designed specifically for this purpose [13, 22], but primarily for use by computational processes, which often produce large and inscrutable programs. A more visual notation such as ours may make these programs more comprehensible and assist with "explainable AI" goals, but we have not yet applied it to them.

While this work has focused on stack-based concatenative programming, there are other concatenative languages

that do not make similar use of stack arguments. For example, the Kihi language [12] is concatenative, composing programs by appending them, but evaluation splices return values into the program source in place of function calls and their arguments, so operands are always within the program rather than on an implicit stack. Our notation does *not* operate over these languages, where there is no delineation between functions and values and alternating rows of each are impossible.

Other visual programming environments for functional languages have been proposed, primarily for applicative languages such as Haskell [5, 24]. Restricting the user to well-typed programs is important in these systems as in ours. Applicative and compositional (concatenative) functional languages are loosely interconvertible, so many applicative programs could be represented in the notation of this paper.

Hazel [20, 21] is a live, functional, structural programming environment. The displayed syntax is similar to a typical applicative textual language, but Hazel programs may have "holes", incomplete portions of the program where the type is known, but the implementation has yet to be written. The editor evaluates around these holes to display calculation results during editing, and values that are present may be rendered in graphical ways. As well as these exploratory programming elements, the "holes" correspond in part to the "backtracking" direction of interactive editing described in Section 4.

The notation in this paper makes clear the data-flow graph latent in the concatenative program. Other data-flow languages are therefore relevant. The most widespread data-flow language is the spreadsheet [15], which also makes use of a grid structure, but does not operate in a similar fashion. Userland [17] presents a spreadsheet-like UI with an integrated dataflow environment, including side-by-side representation of stages of Unix shell pipelines, which are a form of compositional programming. This representation was an inspiration for this work, but the result and semantic use of layout are very different. Systems such as Natto [29] provide a cards-on-canvas aesthetic for working with principally conventional code, equipped with some inter-card data connections and convenient renderers including tables, images, and JSON values, which inspire some of the presentational options discussed. The earliest visual data-flow languages date to the 1960s [30], but there are now a number of nodes-and-wires data-flow programming systems [11], such as Pure Data [1], Yahoo! Pipes, and others [4, 27], while a number of musical tools also follow such an approach [18], including physical modular synthesisers. These systems could for the most part represent the same data dependencies as this work, but without addressing the adjacency of arguments required in editing a concatenative program.

7 Conclusion

Concatenative languages use an unusual programming paradigm that is frequently found difficult to read, but concisely expresses pipeline composition of functions. By showing arguments and functions together, and their connections through layout, we can create a concise representation of the program showing the structure and flow of the program. This paper has presented such a notation and a prototype interactive editor using it, and highlights the potential for alternative representations of this paradigm to simplify the use of concatenative programming in suitable contexts.

References

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

[2] Jonathan Edwards. 2004. Example Centric Programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 124. https://doi.org/10.1145/1028664.1028713

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

[4] 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

[5] Keith Hanna. 2002. Interactive Visual Functional Programming. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/581478.581493

[6] Dominikus Herzberg and Tim Reichert. 2009. Concatenative programming-an overlooked paradigm in functional programming. In *International Conference on Software and Data Technologies*, Vol. 1. SCITEPRESS, 257–262.

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

[8] Kenneth Holladay, Lee Spector, and Maarten Keijzer. 2013. Session Details: Stack-Based Genetic Programming Workshops. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation* (Amsterdam, The Netherlands) *(GECCO '13 Companion)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3253595

[9] Michael Homer. 2022. Swipe-and-Tap Functional Programming. In *ACM Interactive Surfaces and Spaces Conference (ISS)*.

[10] 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

[11] 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

[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] Maarten Keijzer. 2013. Push-Forth: A Light-Weight, Strongly-Typed, Stack-Based Genetic Programming Language. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation* (Amsterdam, The Netherlands) *(GECCO '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1635–1640.

https://doi.org/10.1145/2464576.2482742

[14] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[15] 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

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

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

[18] 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.

[19] Alexander Obenauer. 2021. The Future of the Operating System. https://alexanderobenauer.com/os/.

[20] 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

[21] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (jan 2019). https://doi.org/10.1145/3290327

[22] T. Perkis. 1994. Stack-based genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. 148–153 vol.1. https://doi.org/10.1109/ICEC.1994.350025

[23] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. 2010. Factor: A Dynamic Stack-Based Programming Language. In *Proceedings of the 6th Symposium on Dynamic Languages* (Reno/Tahoe, Nevada, USA) *(DLS '10)*. Association for Computing Machinery, New York, NY, USA, 43–58. https://doi.org/10.1145/1869631.1869637

[24] Matthew Poole. 2019. A block design for introductory functional programming in Haskell. In *Proceedings of the 2019 IEEE Blocks and Beyond Workshop (B&B)*, Mark Sherman and Franklyn Turbak (Eds.). Institute of Electrical and Electronics Engineers, 31–35. https://doi.org/10.1109/BB48857.2019.8941214

[25] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (2019), 9–1.

[26] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and live, programming and coding. *The Art, Science, and Engineering of Programming* 3, 1 (2018), 1–1.

[27] 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

[28] Ben Selwyn-Smith, Craig Anslow, Michael Homer, and James R. Wallace. 2019. Co-located Collaborative Block-Based Programming. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. https://doi.org/10.1109/VLHCC.2019.8818895

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

[30] William R. Sutherland. 1966. *The on-line graphical specification of computer procedures*. Ph. D. Dissertation. Massachussetts Institute of Technology.

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

Received 2022-09-01; accepted 2022-10-02