Swipe-and-Tap Functional Programming

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

Craig Anslow
craig.anslow@ecs.vuw.ac.nz
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand

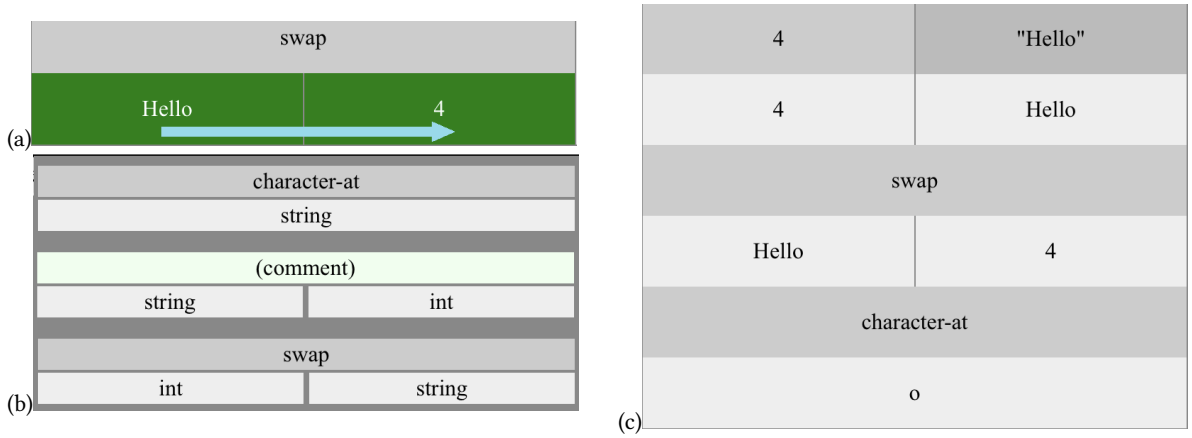

Figure 1: The major editing process in the tool, with the user selecting values to work with, and functions to apply to them, then obtaining a larger program to continue with.
(a) A horizontal swipe, represented by the blue arrow, selects a range of values highlighted in green, here a string "Hello" and number "4". (b) A menu of available functions compatible with the values selected in (a) is displayed. (c) The resulting program after choosing "character-at" from the menu in (b), with the new cell spread below both the "Hello" and "4" cells.

ABSTRACT

Programming on touch-screen devices is notoriously difficult, with conventional programming affordances typically unavailable or unhelpful. Here we present a novel touch-screen programming environment for a style of functional programming that more closely matches typical touch-screen needs, where all editing operations are driven by concrete data values and selected by swipe and tap gestures. The environment provides live editing and supports exploratory programming, with direct display of all calculation values and earlier phases of development always available to edit in-place.

CCS CONCEPTS

• **Human-centered computing** → Ubiquitous and mobile computing systems and tools; • **Software and its engineering** → *Functional languages*; *Development frameworks and environments.*

KEYWORDS

touch-screen devices, functional programming, visual programming

ACM Reference Format:
Michael Homer and Craig Anslow. 2022. Swipe-and-Tap Functional Programming. In *Companion Proceedings of the 2022 Conference on Interactive Surfaces and Spaces (ISS '22 Companion), November 20–23, 2022, Wellington, New Zealand*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3532104.3571459

ISS '22 Companion, November 20–23, 2022, Wellington, New Zealand

1 INTRODUCTION

Programming on touch-screen devices is notoriously difficult, and typical affordances of desktop programming environments are not helpful or even possible on such devices as they rely on keyboard chording, unoccluded pointing, and multiple interaction buttons associated with that pointer. At the same time, touch-screen tablet or phone devices are increasingly the primary or sole device for many users, and it is worth exploring whether different approaches can open up programming, or a form of programming, to these users, who may not desire the full scope of desktop programming environments but wish to express customisations, logic, or "end-user programming" tasks. Different programming paradigms lend themselves to different interaction modalities, and investigating some that are further off the beaten path may expose new affordances. We present a novel touch-screen programming environment for a style

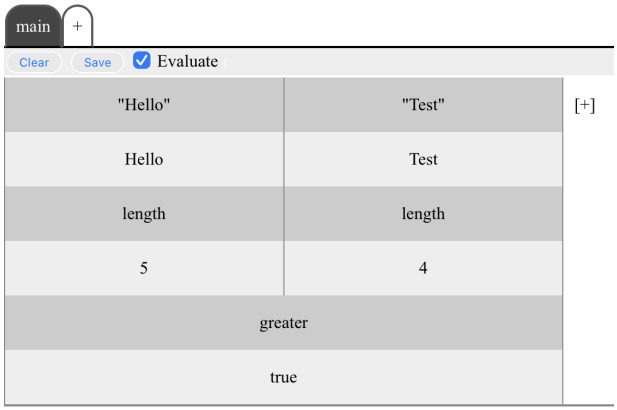

Figure 2: A small program being edited in the system.

of functional programming, where the user interacts using typical touch gestures that map naturally onto the model of computation.

Previous work on touch-screen programming, such as TouchDevelop [4, 8], Pocket Code [7], and Multi-Device Grace [6], has focused on imperative languages, where programs are a sequence of instructions. Under these programming models there is a wide range of possible actions at any given point that the user must wade through to select what they want, or rely on text entry that remains awkward on such devices. In this work, we consider a functional paradigm, and in particular begin with the *concatenative* or *compositional* approach. This approach operates in a more data-driven way and so new affordances for selecting the next step in a program can be explored.

This environment is built around a compact two-dimensional notation for composing functions with many arguments and return values. The program can be edited live with direct feedback, and in an exploratory fashion if desired, with direct support for a range of high-level data types. The principal editing affordance is a horizontal swipe selection, with all other interactions being simple taps, and the available options and their effects are immediately visible.

Throughout the following sections, images of the working prototype system are used to illustrate, and the prototype itself is discussed in Section 4.

2 FUNCTIONS, COMPOSITION, AND CONCATENATION

Functional languages are (in coarse summary) those where programs are constructed out of functions that take certain arguments and return a result or results. A functional language focuses more on transformation between inputs and outputs than on changes of state. While these languages are often statically typed, often side-effect-free, and often lazy, these are not essential features, and not critical to what we want to discuss here.

A key operation in functional programming is the composition of functions. This is the process of taking the output of one function and using it as the input to another, and producing a new function that performs the combined operation. *Compositional* functional languages foreground this operation, while applicative languages such as Haskell foreground applying functions instead [5]. These languages are often called *concatenative* [1, 9] when focusing on another aspect, that appending two subprograms together composes their operations. Composing functions builds up a computation step-by-step, without needing to explicitly name intermediate results or direct where they should go. There is thus less explicit bookkeeping to get in the way, but simultaneously the execution of the program is more opaque. Perhaps surprisingly, these languages make a good fit for touch-screen devices: the locality of the operands allows for compact and contiguous layouts that both make it easier to follow the program logic and lend themselves to interaction with single touch gestures.

We will not discuss textual languages any further here, but instead focus on the visual editing of composed function pipelines. Some of the motivations for pieces of functionality were to support these existing languages, but the system emerged as more general and so we present it in that form. In particular, we will look at the exploratory/live programming aspects of the system that are a natural match for touch-screen interactions.

Figure 2 shows a small program being edited in our prototype, consisting of only a few functions and operating only on trivial data types. The program is laid out in two dimensions: functions are spread out below their arguments, and above their return values (for example, the `greater` function in Figure 2 ranges below the `5` and `4` cells, while its return value `true` spans its full width below). Each function can span multiple cells above and below, and need not have the same number of inputs and outputs. The return value of one function can be used as the argument to another; in this case, the second function will be on a different row to the first (for example, `length` is used twice and both are used as arguments to `greater`).

3 INTEGRATING TOUCH TO PROGRAMMING

The main editing operation is a horizontal swipe, depicted in Figure 3: dragging across a contiguous span of values, which present large touch areas, selects them, and releasing the swipe will bring up a menu of all available functions that can consume those values. The user can then select one of these functions to add to the program, which will be inserted below spanning the full width of the selected values, on a new row if necessary. The outputs of that function will become available below it in turn. Figure 1 depicts this full process: swipe, menu, and new function row.

Because the selection process begins with the arguments, the list of functions to choose from will be tractable and only include items that are usable there. This is a key difference from most other touch-screen (quasi-)visual programming environments, where the

Figure 3: The primary editing operation is a horizontal swipe across cells displaying values, which selects them (highlighted in green). Upon release, a menu offering functions that can operate on those values will appear and selecting one will add it to the program below the selected values.

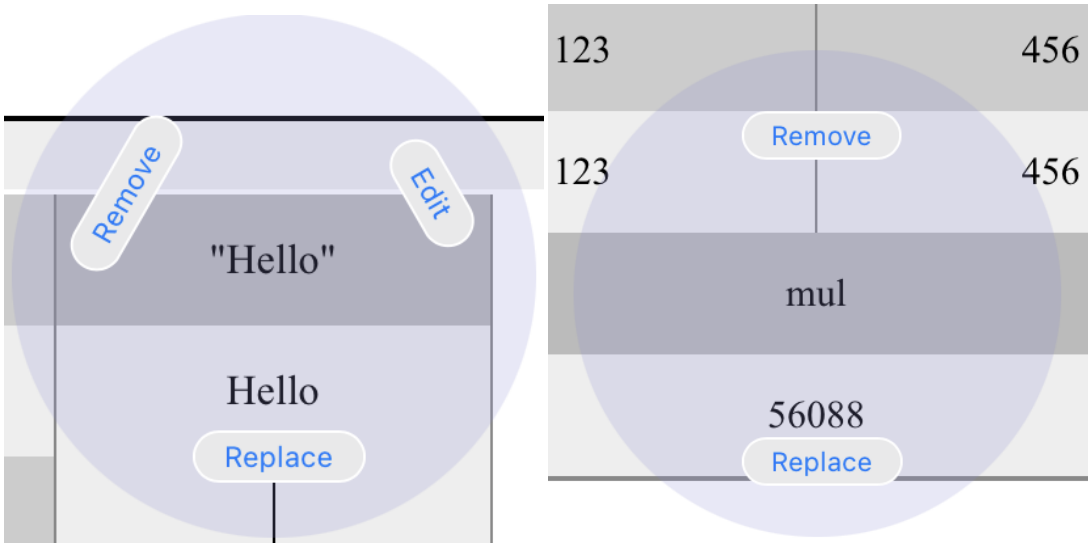

Figure 4: Two radial function menus with different options.

user must generally work in the opposite direction, deciding what to do and then choosing what to do it with. Here, the user can explore the space of suitable functions and their effects based on the data currently available in their program.

A button allows adding new uses of nullary functions, which correspond to literals or constants. For example, a numeric literal "123" is a nullary function that takes no arguments and produces a single numeric output. These functions always appear in the top row of the program. Their outputs will be available on each row below until they are used. Supported nullary functions in the prototype are integer, float, string, and boolean literals, as well as colours and empty collections.

A single tap on a function cell will bring up a radial menu of options for that function, as seen in Figure 4. These options will vary depending on the function and where it is in the program, and can include:

- Replacing the function with another with the same type, selected from a pop-up menu in the same style as for adding a function, but also limited to those with compatible *return* types.
- Removing the function from the program. This is available as long as none of the function's outputs are in use.
- Editing the value of a literal/nullary function, such as a numeric literal "123", a string, boolean, or colour. This is only available for the "top-row" functions that only produce outputs with no inputs.
- Going to the definition of a user-defined function.

The radial menu is intended to keep options close to the user's hand/finger. The only option that is always available is "Replace", which is applicable in all circumstances.

3.1 User-Defined Functions

The menu for adding a function includes a button to create a new user-defined function that takes the selected values as inputs. Each function exists in a separate sheet of the user interface. When editing a function, its inputs are arranged across the top for use like other values. For example, in the following (miniaturised) function definition there are two parameters, an int and a string, in the very top row. The second row now contains the functions that directly consume the arguments, and also a nullary function on the far right.

int	string	
halve	length	false
int	int	boolean

In the current system, user functions cannot be recursive or mutually recursive. They will not be made available in the menu for adding function uses if they would cause a cycle in the program. This is not a fundamental limitation, but is currently a pragmatic one. There are also UX concerns about how best to display these.

3.2 Value Types

This hybrid system is able to display concrete values the program operates over within the program itself. This is useful for debugging, but also for exploratory programming. The display of these values can provide a visual representation of the data being operated on, and tailored editing operations to suit the interactive paradigm.

For example, in our prototype the "colour" data type has a colour literal, chosen through a native colour picker, and colour values are displayed inline as a colour swatch. Operating at this level permits direct manipulation with native affordances for more accessible program creation, and immediate feedback on the concrete result.

This direct display allows creation of a program **to** display the resulting values easily, with the provenance of the calculation visible above. Because the environment is live, edits can be made to the precursor values and the entire display will update automatically, including these high-level data types. Figure 5 shows an example program with a colour and image value, each displayed inline within the program. Editing the string, or colour, literal at the top would cause the images below to update accordingly.

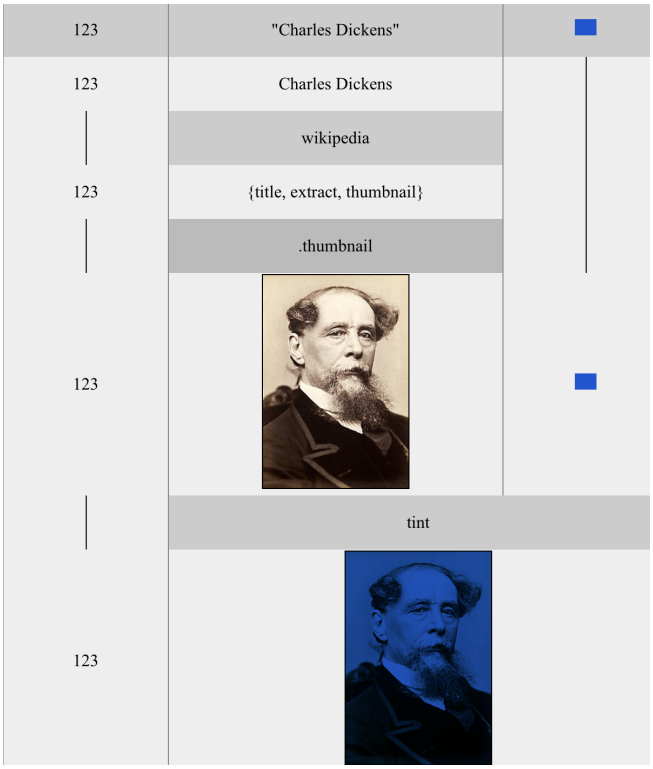

Figure 5: An example program using colour and image values, manipulating an image obtained from the Wikipedia API. Each darker cell represents a function processing the lighter value(s) above it, so data flow runs downwards (e.g. "tint" is a function consuming the image and the blue colour value above, producing the blue-tinted image below).

4 PROTOTYPE

The prototype implementation of this system runs in commodity web browsers, using single-pointer touch interactions, and available at http://ecs.vuw.ac.nz/~mwh/demos/iss2022/. All code executes on the client side, and is saved to local storage in the browser. While it will run on any device with a touch screen, the nature of the system creates relatively wide layouts and so it is more suited for tablet than phone form factors.

This touch-based interface is a derivative of a system originally developed for mouse-and-keyboard interaction [2]. That system was initially designed to work specifically for linear textual concatenative programs, and to convert between the textual and visual representations. The touch-screen version omits all of this text-focused functionality, along with the restrictions that ensure that programs remain linearisable, in favour of the broader exploratory programming style. It uses large touch-friendly action areas and text entry only for literals, with native widgets where possible.

It is possible to switch between displaying the dynamic values themselves and the expected types in the grid. Each of these may be useful, and in particular for programming the types are (approximately) fixed-size and so provide a consistent layout and drag size, while the concrete values can be varied in scale.

5 FUTURE WORK

As well as user studies, some extensions may be worth exploring.

The current prototype uses the horizontal and vertical directions for pragmatic implementation reasons[1], but it is possible that swapping the two would be more convenient at times: laying values and functions out in alternating *columns* instead of rows. Because most values and function labels are wider than they are tall, this would reduce the distance of a selection swipe, and may be more accessible on narrower screens that could not otherwise display the required arguments at once. However, the program would also tend to become extremely wide for even moderately complex programs; it may be most suitable for landscape orientation of a tablet device. Experimentation is required to determine whether both orientations are worthwhile, and when.

As the 2D grid represents a data-flow graph, a graph-based representation of the same pipelines could be as or more usable than the current approach, although node-and-wire systems are generally not well-suited for this interaction. Multiple-representation environments have been found helpful for learner programmers [3] and potentially an animated transition between grid and graph representations would assist the user's understanding.

"Function" cells need not be merely named references, but could have more complex configuration within them. 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. Currently, such affordances only exist for the nullary functions where specifying their value is the entire point, but tighter configuration of later functions may be convenient, particularly on smaller screens.

REFERENCES

[1] 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.
[2] 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
[3] 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
[4] R Nigel Horspool, Judith Bishop, Arjmand Samuel, Nikolai Tillmann, Michał Moskal, Jonathan de Halleux, and Manuel Fähndrich. 2013. *TouchDevelop: Programming on the Go*. Microsoft Research.
[5] 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
[6] 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
[7] Wolfgang Slany. 2014. Tinkering with Pocket Code, a Scratch-like programming app for your smartphone. *Proceedings of Constructionism* (2014).
[8] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: Programming Cloud-Connected Mobile Devices via Touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. Association for Computing Machinery, 12 pages. https://doi.org/10.1145/2048237.2048245
[9] Manfred von Thun and Reuben Thomas. 2001. Joy: Forth's Functional Cousin. In *Proceedings of the 17th EuroForth Conference*.

Received 2022-09-30; accepted 2022-10-07

[1] It uses an HTML table for layout, where cells spanning multiple *columns* are significantly more tractable than those spanning multiple *rows*. This is an artifact of the implementation strategy and not fundamental to the approach.