Calling Cards: Concrete Visual End-User Programming

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

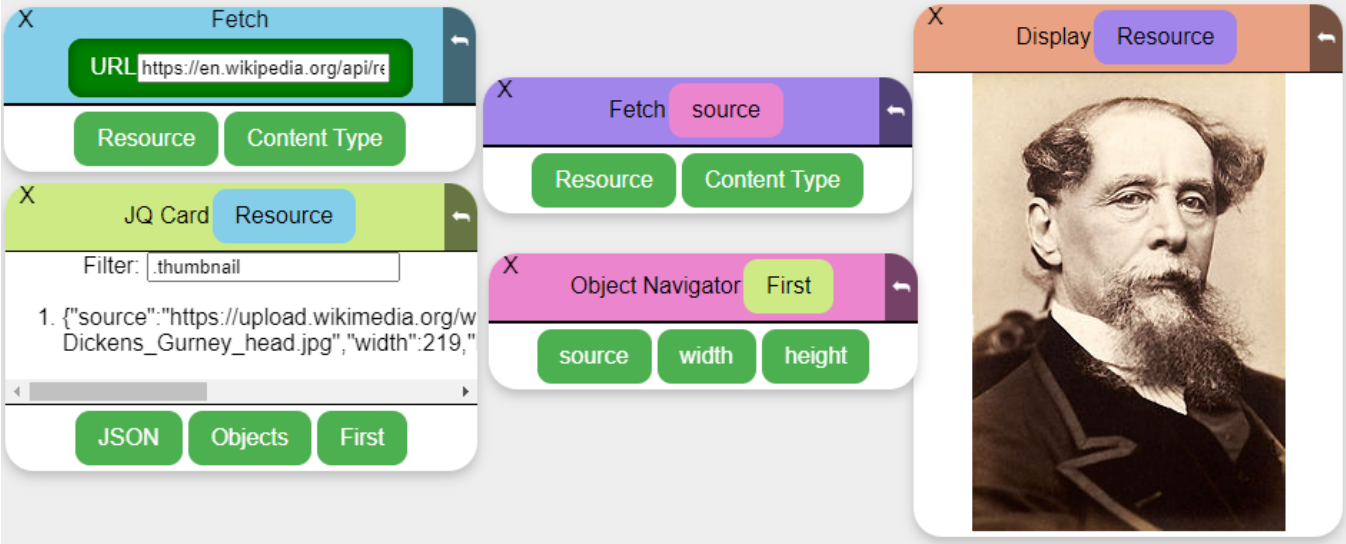

Figure 1: Program that retrieves random Wikipedia thumbnail.

ABSTRACT

Creating a program that performs even a simple task and shows the result is unapproachable to most people, and even trained programmers face a burden to create a new program. In this paper, we introduce a prototype system and model for live visual dataflow programming where intermediate steps are visible and all components are tangible and manifest. Our system aims to allow a user to dive in and immediately have a working program that can be incrementally extended.

CCS CONCEPTS

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

KEYWORDS

end-user programming, exploratory programming, visual programming, dataflow programming

<Programming> '22 Companion, March 21–25, 2022, Porto, Portugal

ACM Reference Format:
Michael Homer. 2022. Calling Cards: Concrete Visual End-User Programming. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (<Programming> '22 Companion), March 21–25, 2022, Porto, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3532512.3535221

1 INTRODUCTION

An end user wanting to instruct the computer to perform a simple task, or to retrieve, slightly process, and show some data, faces a daunting task. Even programmers have a significant burden in creating a new program, and debugging missed data-processing steps is a complex hassle. We have developed a visual programming approach that allows live coding, makes all intermediate steps visible and immediately available for further use, and lets the user stop at any time with the fruits of their labour on display.

A key goal of this system is to have data values in use be manifest, tangible, and visible. This has three purposes:

- to have the state be apparent so that the full computational flow can be inspected;
- to have potential intermediate results always available for incremental extension;
- and finally that simply displaying the data may *be* the point.

Our system is essentially a data-flow programming environment: individual displayed "cards" take in input at the top and emit output at the bottom, usually displaying their principal output in

between. Each card represents a transformation step, and the user can compose the transformations they want piece-by-piece, always incrementing from a known point they can see, and always able to reconfigure previous steps or remove later ones.

By keeping the intermediate values on display we hope to make thinking about the next and previous step easier, and by updating whole system live as it changes we hope to make the effects of modifications readily apparent. Displaying by default and movable cards also let users create a tailored space displaying exactly what they want, and exposing just the actions that they want, without needing a separate space. The boundary between "programming" and "customisation" is thus blurred, we argue for the better.

The prototype of the discussed system is available in any web browser at https://mwh.nz/demos/px2022. On first visit, it will load a demonstration program making use of some facets of the system. Subsequent page loads will restore the last state.

The next section discusses the computational model of the system, setting aside the visual and user-interaction elements for the moment. Section 3 discusses the practical realisation of the user interface in our prototype, then Section 4 elaborates on the mechanisms for creating new cards in the system. Section 5 contains discussion and positions this project among some related work, while Section 6 concludes and identifies future explorations.

2 THE MODEL

The operating environment is an open space that can contain any number of *card*s. A card has a title, zero or more *input*s, zero or more *output*s, and usually a *body*.

Both inputs and outputs have labels, to identify them to the user. An input also has a specified type or types of data it can accept. Similarly, an output has a type of data it emits. Any input can be *connected* to a compatible output, but an output can connect to multiple inputs.

The **card** has a dual role, depending on the perspective one takes:

(1) In one sense, the card "is" what is in the body: its outputs are renditions or transformations of that body, and a connection to the card is a connection to the data it holds.
(2) In another sense, a card represents a computational node: taking in data from its inputs, it produces some result, which is the body, and makes the result or components of it available through its outputs.

In the first sense, there is thus a concrete value currently associated with every card, which is expected to be relatively long-lasting — not a fast-moving stream pipeline, but specific pieces of data being manipulated.

In the second sense, cards and connections form a compositional functional programming language [16] with variable-arity functions joined together.

We lean more to the data-primary viewpoint in the design choices of the system, but both have validity, and both are simultaneously true.

2.1 Behaviour

Precisely what a card does with its inputs, or where any outputs come from, is not restricted by the model, and cards may work at very high or very low abstraction levels. Instead, for any specific purpose an integrator is expected to choose a coherent set of cards to make available for the intended users.

The cards serve the dual role of representing data values and transformation steps. A card has contents, which ambiently exist, and may be displayed (or not); a card also has outputs, which derive from its content, and may be routed for use elsewhere (or not). On a conceptual level, initial data values can be thought of as files stored on the system, a card displaying that value as the result of opening that file, and outputs as structured means of "zooming in" on facets of the data that were not originally clearly visible. On a practical level, cards do take inputs like function parameters and produce outputs like (multiple, named) function return values by executing arbitrary code in between.

Only explicit input–output connections communicate between cards, and that communication is routed via a "kernel" component that handles multiplexing, retaining current values, and tracking the global picture of what is happening within the program. This indirection serves to prevent any inadvertent latent dependency on the specific implementation of another card being introduced, so that new cards can always expect to interoperate with existing ones. It also provides metadata insight into what is happening in the system, which can be exposed through special cards.

In the current implementation, card behaviour is primarily implemented in JavaScript (matching the host system), but the model does not rely on any particular host language or environment.

2.2 Types

At the abstract level, the type of data being exchanged is opaque, but to interact and edit the system needs to understand the data in use. In the current prototype, supported types include text, pictures, numbers, tables, JSON objects, sequences, maps, and arbitrary binary data.

An input can be specified to accept multiple types, and the card can adjust its behaviour according to what it receives. An output always has a concrete type.

3 THE USER INTERFACE

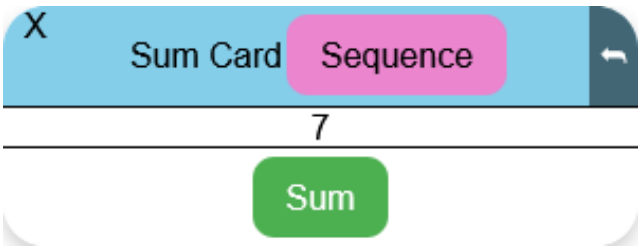

Figure 2: A simple "sum" card with one input for a sequence of numbers, and one output for the sum. The output value is displayed in between for the user.

As displayed on screen, shown in Figure 2, there are three sections of each card:

(1) a "title bar" of sorts with a unique background colour, displaying at least the title and some user-interface controls,

along with any inputs of the card displayed as individual buttons;
(2) the body, which usually displays a visible rendition of one of the outputs, and is generally the tallest area;
(3) the outputs, each displayed in the same fashion as inputs.

It is possible for there to be no inputs, no outputs, or no body. Each card in the system is identified by a unique colour in its title bar. When an input on a card is connected to an output, the input's colour changes to match the card the output belongs to.

Cards can be laid out on screen arbitrarily by drag-and-drop with pixel positioning, like windows in common GUI systems. The layout is not semantic and can be adapted to suit what the user wants to have visible, or to group related cards together. A cross in the upper left permits closing and removing a card.

The "body" for most cards will be its principal output rendered in a human-friendly form, in line with the goal of all data being manifest and visible. The default is that the first or only output is used directly for the body. The user interface understands how to render various types of value automatically, so for example outputting an image will result in the image being displayed, while outputting a "table" value (an ordered group of columns) will cause a standard table display.

Each card has a control in the top-right corner to "flip" the card and display the (notional) back. The back of the card contains meta-settings, including for every card a toggle for displaying the body or not (so that larger or repeated displays can be omitted if the user prefers). Some cards provide additional secondary settings that are also displayed on the back of the card.

3.1 Card Connections

There is no direct visible representation of the graph of card connections, because this becomes obstructive with even a small number of cards (see an example in Figure 4). However, input buttons do indicate where their input comes from by both colour and label. An input takes on the hue of the card that owns the output it is connected to, and also copies and displays the label of that output. This is visible at all times and makes it possible to follow the trail of connections passively, visible in Figure 3.

When more direct indication is required, hovering the mouse pointer over an input button will display an overlaid arrow from the corresponding output, and will highlight the output button itself. There is also a hidden feature to show all arrows at once "tube map"–style by holding "C"; this was used to produce Figure 4 and Figure 8, but is too unwieldy to be enabled routinely.

At present, there are three ways to form a connection between cards:

(1) by drag-and-drop from an output button to an input button;
(2) by point-and-click, on an output button and then an input button;
(3) by creating a new card with a connection already in place through double-clicking an output and choosing a suggested type-compatible card from the resulting menu.

In all cases, only type-compatible connections can be formed, and no connection that would create a cycle where a card depends on its own output is permitted. Both dragging and clicking cause the compatible inputs to be highlighted and incompatible inputs dimmed, and highlight a compatible input further when the pointer is over it. Point-and-click displays an overlay arrow from the output to the pointer so that it is clear which output is to be connected.

We expect that future experimentation may result in removing either drag-and-drop or point-and-click, as these are fully redundant with one another. Previous work has suggested drag-and-drop can be a problematic interaction paradigm [9, 13, 15], but also that it may be both expected and preferred by younger users [3].

3.1.1 Typeable Inputs. Some inputs are marked "typeable". These inputs present with a text-entry field within the input button as seen in Figure 5, and the contents of this field are used as the input. Such an input may still be connected in the ordinary way, and that connection will supplant the typeable field. These fields exist to simplify construction of programs with simple text configurations, like a textual filter or URL retrieval.

3.2 Instantiating Cards

In the present prototype, new cards can be created by double-clicking the background to open a menu of all available cards. Double-clicking on an output button also produces a menu of suggested compatible cards that could accept the type of that output. Selecting one of those cards will add a new card to the world with that output connected to one of the new card's inputs.

Each new card is allocated a unique hue, used for the title bar, connected inputs, and overlaid arrows. The card is initially placed in an available empty space on screen, starting from the top left. Any inputs not connected are initially green and labelled with their purpose. Depending on the card, it may require all inputs to be filled before its outputs are active, but it is possible to form connections from those outputs from the beginning.

3.3 Using Cards

In the prototype, we have implemented many cards for experimentation, across a broad range of abstraction levels. The selection to date has been demand-driven: what would be an interesting thing to try, or would help to investigate the behaviour of the system, and what cards would support that? Similarly, at times the system (particularly the UI) has been extended to facilitate behaviours that would be convenient for cards.

Existing cards include, among others: Text-entry and numeric-entry cards for single values; Low-level arithmetic with sum, product, difference, and quotient outputs; Statistical calculations on numeric sequences; Converting CSV input to tabular data, and tables to JSON text; Displaying any input value, with no outputs; Outputting the time every second, with no inputs; Fetching a web resource from a URL, and making HTTP posts of data to a URL; Constructing a sequence of values from multiple individual inputs, dynamically created; Exposing fields of a JSON object as individual outputs, dynamically created; Embedding another textual language (jq [17]) within the processing pipeline; File input, permitting a local file to be accessed in the system; An image input, permitting direct entry of image data as part of the program.

The live prototype includes almost all currently-developed cards at once, although we would expect real-world use to use tailored subsets. The nature of the programming model here permits elements like images, which are segregated second-class items in most

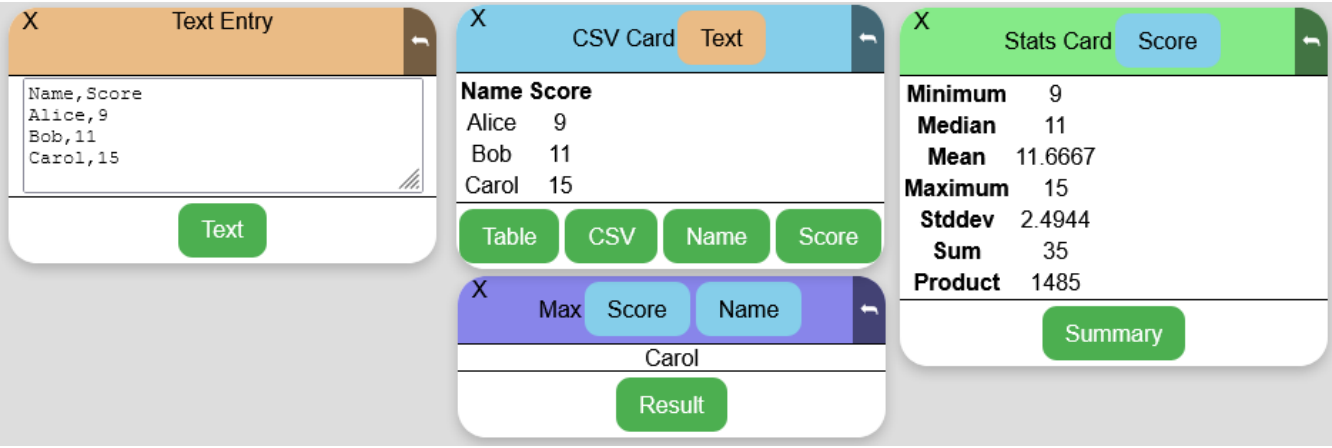

Figure 3: A program of four cards and four connections: one from the orange text entry card to the blue CSV card, two from the blue to the purple maximum card, and one from the blue to the green stats card, all indicated by the colour of the input button matching the source card's title bar.

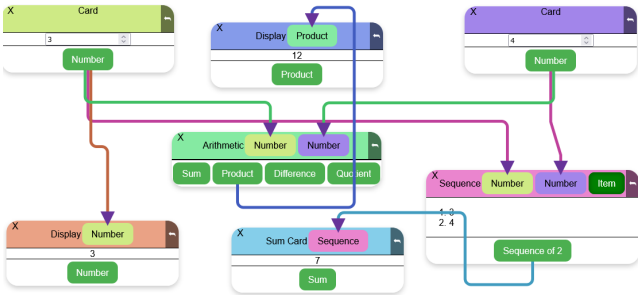

Figure 4: Connections between several simple cards displayed as overlaid arrows in a non-standard mode. The arrows would not ordinarily be visible all at once, but illustrate the range of interconnections.

Figure 5: A "URL fetch" card with a typeable URL input field.

languages, to be included within the program itself on equal terms with text and numbers.

4 CREATING CARDS

The current prototype operates as a web page, where executable code is in JavaScript. Cards can be implemented as JavaScript classes inheriting from a `Card` superclass, which provides useful default behaviours. For example, a trivial "add two numbers" card could be implemented as in Figure 6, setting up the properties of its inputs and output in the constructor by assignment, and overloading the `calculate` method to generate the result.

The `Card` class ensures that `calculate` is called each time the inputs are updated. JavaScript proxies [6] and properties provide "assignment" syntax for manipulating inputs and outputs. More advanced cards could directly subscribe to input changes or use arbitrarily-more-complex code to implement whatever behaviour is desired.

The superclass, and its `inputs` and `outputs` fields, have been built to support the possibility that the author of a card is not a confident programmer. Some behaviours are "magical" or automatic, such as creating an input when it is referenced, so that copy-and-paste amateur programming has a chance of producing something useful with minimal syntactic overload. The idea here is that the users of the visual system may span a range, with some using exclusively the visual system and others taking on a "local expert" role tweaking custom cards for their environment.

A card can create, remove, and alter inputs and outputs dynamically. For example, we have an "Object Navigator" card that accepts a single JavaScript/JSON object as input, and has an output for every key in that object. This card permits an object tree to be navigated and connected within the main system in the usual way, and can create potentially very many outputs according to the input value. An input connected to an output that is removed is disconnected, in the same way that it would be if the source card were removed.

4.1 JavaScript Card

There is also a "JavaScript Card" that permits creating a new arbitrary card from within the system. This card has space on the back to provide both "setup" code and "calculation" code, corresponding to code in the constructor and `calculate` method above. The card updates immediately when its code is modified, permitting live prototyping and experimentation. We use this card to explore and experiment with additional card ideas before forming them into permanent concrete cards, but it can be used for its own purposes as well to create bespoke cards and behaviour.

```
class AddCard extends Card {
  constructor(init) {
    super(init)
    this.inputs.number1.type = 'number'
    this.inputs.number1.label = 'Number 1
        '
    this.inputs.number2 = {
      type: 'number',
      label: 'Number 2',
    }
    this.outputs.sum = {type:'number',
        label:'Sum'}
  }
  calculate(n1, n2) {
    this.outputs.sum = n1 + n2
  }
}
```

Figure 6: A trivial "add two numbers" card implemented in JavaScript. The backend library of our system permits simple assignments to cause the inputs and outputs to be created and updated.

Once more, this card also anticipates use for small bespoke extensions by the end user, and exposes the same magic `inputs` and `outputs` functionality. It nonetheless exposes the standard JavaScript API, and so can fetch web resources, spawn worker threads, set timeout callbacks, and so forth, enabling advanced custom behaviours.

4.2 Nesting

Subprograms can be nested inside the "container card". This card provides a separate canvas that can be opened, edited, and closed, and can contain as many cards and connections as desired. Special "Input" and "Output" cards, available only within these nested subprograms, create connection points on the container itself, and can be connected to other cards within the container. The inner world is isolated from the outside otherwise, and direct connections cannot be made across the boundary. Only the explicit inputs and outputs are able to communicate.

From the "outside", the container card is an ordinary card, that can been moved around and have connections made to or from it. The innards are not visible in the outer world, though they are part of the same overarching program. A complex program may thus be created exploratorially and then abstracted into a single card, opaque to the outside, and tidied out of the way of the main operating space. Figure 7 shows such a subprogram as it appears from the outside. Clicking the button would zoom in on the pictured subprogram as shown in Figure 8, allowing it to be edited just as the main program is, and these edits would also take effect live.

Custom cards can thus be assembled using only the mechanics of the system itself, with whatever behaviour can be produced using the other available cards. These are not so easily converted into permanent cards of the system, and so suit more bespoke use cases.

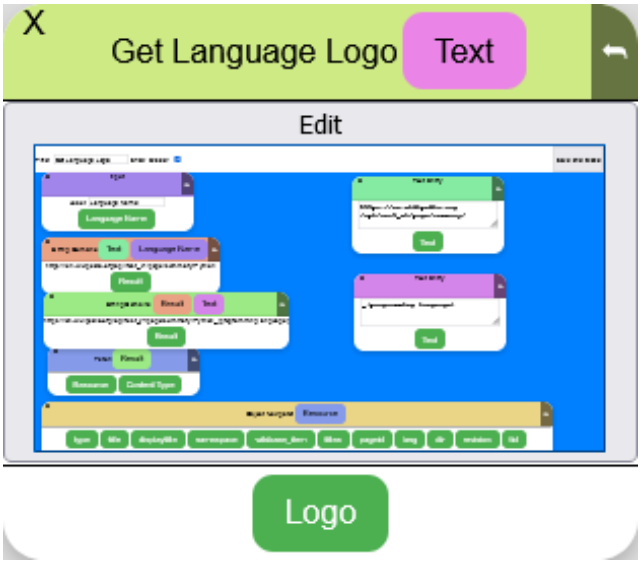

Figure 7: A submodule of a program isolated as a single card, here rendering the contained program within the card. This figure is approximately actual size: the scaled-down image of the subprogram is rendered within the button at quarter size, illustrating structure but not readable textual content.
Figure 8 shows the contained subprogram, rearranged to show the entire program and how the input and output are created.

They are able to compose the very high-level operations of other cards directly and so condense a large amount of backing source code into a single unit.

4.3 Remote Cards

The prototype also allows cards to be backed by a WebSocket connection, a two-way communication channel to a web server. This feature exists to support some actions that are not available within a web browser, but it also permits the remote implementation to use any technology, and simply speak the JSON-based protocol sent over the socket. We have not implemented any cards using other languages at this stage.

5 DISCUSSION AND RELATED WORK

Displaying all data by default places some constraints on the viable programs that can be built. Cards can simply be very large (our prototype caps how much of the screen can be occupied by a single card before scrolling is introduced, but it is still necessarily large). Substantial "batch" processing tasks, of the sort that many dataflow pipelines are used for, are not well-supported in this system. Its expectation of long-lived displayed data that is intrinsically "part" of the program works against this, while for most dataflow models at least some input data is provided from the outside and the program applied to it. The prohibition on cycles also prevents the present system from being Turing-complete (though a card may provide Turing-complete functionality in itself). Extending the system with

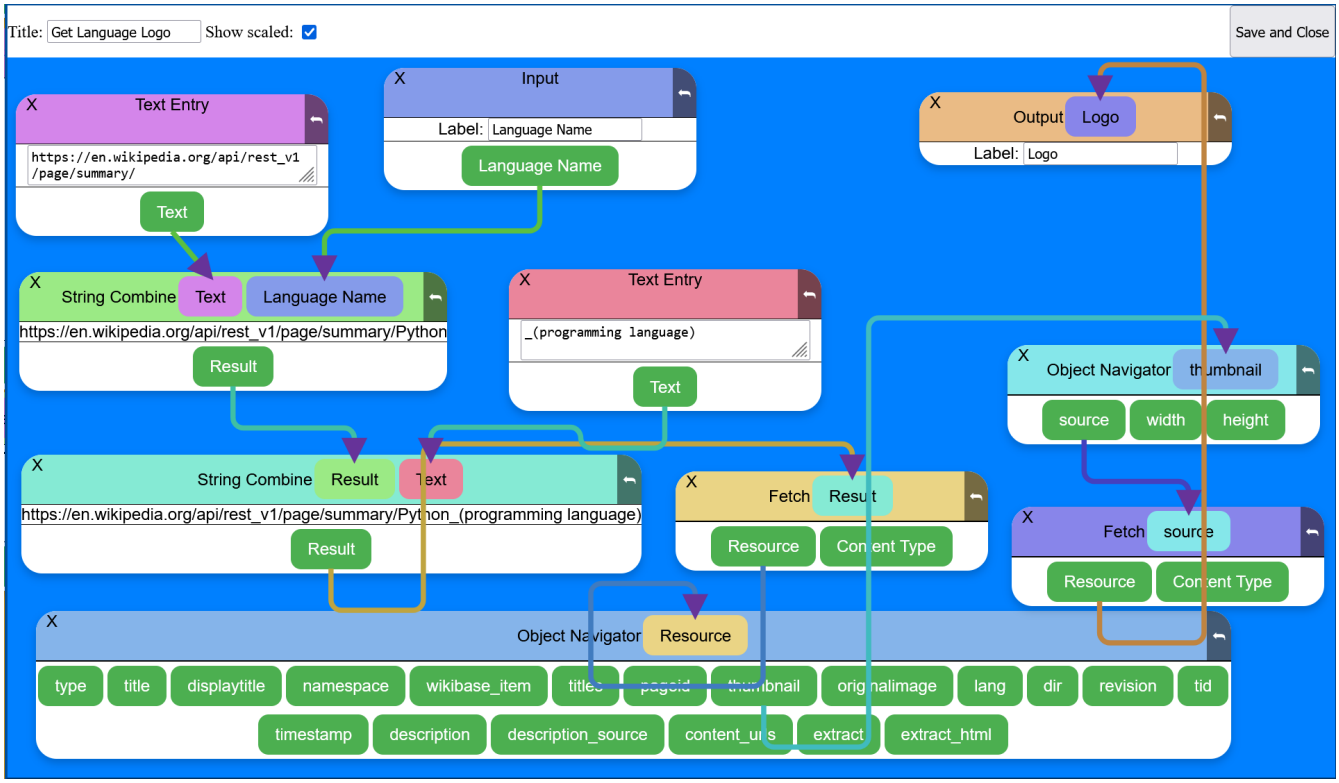

Figure 8: The submodule from Figure 7 in its full editable size, rearranged to include the full program and the optional overlaid connecting arrows. When this pop-up editor is closed, it scales down to appear in a card in the main program with a single input "Language Name" and a single output "Logo".
The two "Input" and "Output" cards at the top generate and label the connection points on the submodule card seen earlier. Connections from and to those cards communicate between the outside world and the subprogram here.

recursion, named subprograms, or alternative connection types remains future work.

We regard these limits as setting the scope of the tool and what is built within it. The option to hide the body of chosen cards allows shrinking redundant displays, but a program that is too large to fit on screen comfortably is also too large to be comprehensible in this model. Shifting part of the program into a nested container, isolated with well-defined boundaries, is encouraged implicitly by the need for on-screen space.

Similarly, seeking a well-matched set of available cards at the right abstraction level — whether creating those cards is something the user can do themeselves or not — is important to a productive experience in this system. These needs may change over time, and the integrator can make higher- or lower-level cards available as needed, allowing subtasks to move out of the card language and into a conventional programming language.

However, some tasks may simply not be suitable for this system, and that is not a problem either. We see two main purposes for this sort of environment: exploratory development, where the always-concrete data model facilitates very direct querying (an "exploration" or "programming" purpose), and building a bespoke "dashboard" space to show exactly the information or actions needed, without relying on precisely the right components having been built already (a "customisation" purpose). These are not cleanly distinguished and represent a continuum where the same act could be analysed as exploration, programming, or customisation; we would even argue that as far as possible this system *is not* for programming, despite producing programs. In both cases, a partial program is also a complete program with a partial result, and the self-limiting nature of the display highlights where further modularity or abstraction are needed, avoiding the Deutsch limit of *too many* items on screen by limiting the program to *too few*.

The most direct influence on this system was some previous work we had done using the Lively Kernel [22, 32]. The Lively Kernel provides a JavaScript-based "operating system" analogue with separate programs and a conventional windowed environment, largely unlike this system, but includes a live `that` pointer referring to the program the user most-recently interacted with [31]. In our work [10] we had utilised this pointer to create live data connections between separate applications within the shared JavaScript runtime environment through direct object manipulation. Unlike the present work, these connections were not intended or facilitated by the applications or host system and represented more of a hostile extension. Some subsequent work in Lively, such as WebWerkstatt [20], and the earlier and ongoing Fabrik [14], steer more in the direction of modelling data flow, continuing a focus

on the application programming side. This work also drew from experience building a parallel graphical language for cellular automata [21], which highlighted issues of scale and always-visible connections.

As part of research laboratory Ink & Switch's "Capstone" experiment one of a series of prototypes had "data pipelines" [18] resembling our system, including cards with "uses" (input) and "exposes" (output) ports. This was the most similar system to our work, and while they moved on from this prototype to more conventional programming swiftly, including for some of the spatial reasons above, we agree with the direction it was going. Our system facilitates the same kind of connections, albeit with a deeper investigation including nested subprograms and cards fully isolated from one another. Our view is that the spatial issues are addressed by modular nesting, further abstraction, and the self-limiting nature of available space.

A number of flow-chart-style box-and-line visual languages exist, including those using the graph edges for both control- and data-flow. Examples include LabVIEW [7, 26], Simulink [33], Pure Data [5], Yahoo! Pipes, and Blender's several node graphs. A number of musical tools also follow such an approach [25], including physical modular synthesisers. Our system's focus on exposing a single current value at each stage, and focusing the nodes on those values themselves, is distinct from most of these. In most cases, the goal of these systems is to express a repeated process, rather than to represent a current state as in our approach. In this work all editing is fully live, with no run-edit distinction.

Apple's Shortcuts language [1] presents a linear top-to-bottom flow where dynamic "variables" can be selected to use one of the outputs of a previous "action", and any action can have many possible outputs. Similarly to our system, no direct visual connection between the origin and its use is made ordinarily, but icons and colouring suggest likely links. Shortcuts by nature expects the program to run many times with unknown data and so does not have facility to display values or live-edit the running program.

The most widely-used programming system displaying intermediate values is the spreadsheet [19], which uses specialised textual syntax alongside spatial references. Several spreadsheet-derived systems extend this paradigm to provide multiple-representation reactive programming environments with additional data types, visualisations, and classical programming functionality [4, 8, 27], retaining the core spreadsheet editing and input cycle. Userland [24] provides an integrated dataflow environment incorporating both spreadsheet cells and Unix shell pipelines, with visual representations of intermediate states. 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. The Vivide system [30] is a live dataflow programming environment where interactive widgets are scripted with Smalltalk, and programs can be composed out of many arbitratily-powerful widgets outputting their chosen data to one another. This flow of data is arranged by the end user, but a widget is not identified with its present data value. All of these approaches provide some level of dataflow and reactive live programming, with (most) data values visible and available for further use. These all have a more explicit "programming" phase contrasting with our system and the Capstone prototype, but consequently expose more power directly within the system. Our present system focuses on avoiding even this level of traditional programming exposure, but more hybrid approaches seem to have merit.

Parts of the "operating system" side of this model are reminiscent of aspects of the Plan-B research system, notably the original "box" filesystem [2], where typed "boxes" (files) could be connected to one another such that accesses to one were relayed through a translation operation to the other. Our system currently focuses only on what is displayed on screen, rather than a persistent filesystem behind it, but on a conceptual level we do see the data values as primary and related in a similar way.

6 CONCLUSIONS AND FUTURE WORK

We have presented a prototype system and model for exploratory data-flow programming using a system of "cards", where intermediate values are visible, manifest, and tangible. Our prototype operates in a web browser at mwh.nz/demos/px2022 and supports a range of different cards providing both high- and low-level operations. The operating model anticipates users across multiple levels of skill and investment, including those approaching being conventional programmers, and envisages curated sets of operational cards made available according to the application domain of the user.

6.1 Future Work

We plan multiple user experiments with this system, both addressing the user interface itself (which has evident flaws at present) and the overall model of the system. In particular, the system currently supports both point-and-click and drag-and-drop connections between cards with entirely different appearance and interaction: we anticipate that one or other of these will be removed if the alternative proves more usable or clear. We also intend to explore user intentions and desires for what can be done in the system, with regard to the types of program/custom display that can be created.

The strong separation between cards via the intermediated kernel provides both positive and negative elements. We intend to explore both strengthening and weakening this quasi-"microkernel" approach to see what helpful programming and interaction models may arise from it.

We plan to investigate forming existing and additional cards into coherent subsets for specific users, establishing the level of abstraction and types of operation that are needed. It is unclear whether entirely-separate domain-specific languages or a language-level module or dialect system, [11], are the appropriate way to support the range of use cases, or whether in the operating-system vein this is more of a "package manager" problem which transcends layers [23] and may be either separated or integrated. Each of these approaches has points in favour and against, and further real-world experience is needed to determine which mechanism is suitable when.

The visual editor exposes a range of functionality to end users, novices, or non-programmers, but such interfaces are notoriously tiresome and frustrating for experts. A textual mode would allow these experts to construct pipelines in a more traditional "programming" manner while remaining entirely within the execution and data model. In representing a data-flow system, this textual

language would most naturally fit within the concatenative or compositional [16] mould, structured by chaining together transformative steps to realise a result. This textual representation could be dynamically convertible to and from the visual representation in the manner of Tiled Grace [12, 28] to permit different editing approaches for different tasks to those users comfortable with both, or for expert users developing cards or subprograms for others.

Finally, although the present prototype allows encapsulating subprograms within a single card, these only compact an explicit portion of the dataflow graph. Extending the system to allow reusable abstractions, and additional types of inter-card connection for operations in the manner of higher-order functions, loops, and parallelism is another active area of work in collaboration with some industrial applications.

We hope that such findings may have broader applicability to other end-user systems.

REFERENCES

[1] Apple Inc. 2022. The flow of content in Shortcuts on iPhone and iPad. https://support.apple.com/en-nz/guide/shortcuts/apd33c8d5c6d/ios.

[2] Francisco J. Ballesteros and Sergio Arevalo. 1999. The box: a replacement for files. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. 24–29. https://doi.org/10.1109/HOTOS.1999.798373

[3] Wolmet Barendregt and Mathilde M. Bekker. 2011. Children May Expect Drag-and-drop Instead of Point-and-click. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI EA '11)*. ACM, New York, NY, USA, 1297–1302. https://doi.org/10.1145/1979742.1979764

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

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

[6] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies - Virtualizing Objects with Invariants. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 154–178. https://doi.org/10.1007/978-3-642-39038-8_7

[7] Martin Erwig and Bertrand Meyer. 1995. Heterogeneous visual languages-integrating visual and textual programming. In *Proceedings of Symposium on Visual Languages*. IEEE, 318–325. https://doi.org/10.1109/VL.1995.520825

[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] Douglas J. Gillan, Kritina Holden, Susan Adam, Marianne Rudisill, and Laura Magee. 1990. How Does Fitts' Law Fit Pointing and Dragging?. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. ACM, New York, NY, USA, 227–234. https://doi.org/10.1145/97243.97278

[10] Michael Homer. 2010. Lively JavaScript. Honours report, Victoria University of Wellington.

[11] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. Graceful Dialects. In *ECOOP 2014 — Object-Oriented Programming*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer Berlin Heidelberg, 131–156. https://doi.org/10.1007/978-3-662-44202-9_6

[12] Michael Homer and James Noble. 2014. Combining Tiled and Textual Views of Code. In *IEEE Working Conference on Software Visualisation*. https://doi.org/10.1109/VISSOFT.2014.11

[13] 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

[14] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. 1988. Fabrik: A Visual Programming Environment. *SIGPLAN Not.* 23, 11 (jan 1988), 176–190. https://doi.org/10.1145/62084.62100

[15] Kori M. Inkpen. 2001. Drag-and-drop Versus Point-and-click Mouse Interaction Styles for Children. *ACM Transactions on Computer-Human Interaction* 8, 1 (March 2001), 1–33. https://doi.org/10.1145/371127.371146

[16] 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

[17] jq. 2018. jq is a lightweight and flexible command-line JSON processor. https://stedolan.github.io/jq/.

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

[19] Andrew 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

[20] Jens Lincke and Robert Hirschfeld. 2013. User-Evolvable Tools in the Web. In *Proceedings of the 9th International Symposium on Open Collaboration* (Hong Kong, China) *(WikiSym '13)*. Association for Computing Machinery, New York, NY, USA, Article 19, 8 pages. https://doi.org/10.1145/2491055.2491074

[21] Deacon McIntyre and Michael Homer. 2020. Poster: A Visual Programming Language for Cellular Automata. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. https://doi.org/10.1109/VL/HCC50065.2020.9127283

[22] Tommi Mikkonen, Antero Taivalsaari, and Mikko Terho. 2009. Lively for Qt: A Platform for Mobile Web Applications. In *Proceedings of the 6th ACM Mobility Conference*. https://doi.org/10.1145/1710035.1710059

[23] Hisham Muhammad, Lucas C. Villa Real, and Michael Homer. 2019. Taxonomy of Package Management in Programming Languages and Operating Systems. In *Workshop on Programming Languages and Operating Systems*. https://doi.org/10.1145/3365137.3365402

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

[25] 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, 29–38.

[26] Mark Noone and Aidan Mooney. 2018. Visual and Textual Programming Languages: A Systematic Review of the Literature. *Journal of Computers in Education* 5, 2 (2018), 149–174. https://doi.org/10.1007/s40692-018-0101-5

[27] 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

[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 - write JavaScript on a 2D canvas. https://natto.dev/.

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

[31] Antero Taivalsaari and Tommi Mikkonen. 2010. Simplifying Interactive Programming with Keywords "that' and "those'. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. 393–397. https://doi.org/10.1109/SEAA.2010.16

[32] Antero Taivalsaari, Tommi Mikkonen, Dan Ingalls, and Krzysztof Palacz. 2008. *Web Browser as an Application Platform: The Lively Kernel Experience*. Technical Report 2008-175. Sun Microsystems Laboratories.

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