Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 21 (2013) 68 – 74

Procedia Computer Science

The 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2013)

Data Programming for Non-Programmers

Dave Mason[*]

Ryerson University, Toronto, Canada

Abstract

“Big Data” is becoming a standard catch phrase that often means more like “Ubiquitous Data” - it’s not necessarily all that big, but it’s everywhere, and its collection rate is accelerating. Citizens may only have a need to spend a few hours per week attempting to make sense of that data. The concomitant worry is that citizens will be left behind as the IT priesthood leverages that data to run our world. But the skills to program in conventional languages are not universal; so, what to do?

We are creating a visual programming environment for user-programmers (who are not trained programmers). Because of the sporadic usage pattern of such users, traditional languages and code structuring approaches are both overkill and difficult to learn. Therefore, the environment is designed to support very fluid relationships among code chunks, in terms of their relationship and evolution, as well as very fluid interactions among code chunks when processing data.

Keywords:
End-User Programming, Ubiquitous Programming, Citizen Programmer, Ubiquitous Data, Visual Programming

1. Introduction

“Big Data” is becoming a standard catch phrase that often means more like “Ubiquitous Data” - it’s not necessarily all that big, but it’s everywhere, often dynamic, and its collection rate is accelerating. The concomitant worry is that citizens will be left behind as the IT priesthood leverages that data to run our world. But the skills to program in conventional languages are not universal; so, what to do? Citizens may only have a need to spend a few hours per week attempting to make sense of that data.

We are creating a visual programming environment for sporadic users/programmers - not necessarily novice, but not classically trained. It is loosely based on Scratch(1), but rather than multi-media, sprite-based exploratory programming, we are interested in democratizing the vast swaths of (open and proprietary) data becoming available with a programming-for-the-rest-of-us environment to access, analyse, and extract meaning from that data. The environment is implemented in Javascript, so gains the ubiquity of that

[*]Corresponding Author
Email address: `dmason@ryerson.ca` (Dave Mason)
URL: `programmingfortherestofus.com` (Dave Mason)

language. Versions will be available for the web as well as iOS and Android devices, and in the longer term we plan to also run it in a back-end mode on servers.

We aim to facilitate ordinary citizens posing problems like: "I wonder if there is a correlation between street lighting and the occurence of potholes."; discovering a city database of potholes and a power company listing of streetlight locations and writing a short program to extract and correlate the data and produce a nice little infographic that answers the question. In this way data becomes information and can then address personal goals or move into the citizen's political realm. In the coming data-heavy world, we believe such programming will become significantly more ubiquitous than experimenting with spreadsheets is today, and we believe it is important to have a fairly powerful programming environment to allow citizens to go beyond canned answers to questions that someone else has determined are important.

Our assumption is that, like spreadsheets, while there will be "power"-users who spend much of their time using the environment, most users will use it on the order of an hour a day. As such there are two particular challenges that affect our design:

1. The sources of data will be highly varied; they may include traditional spreadsheets, Twitter or Facebook APIs, real-time monitored data, or other code the user (or others) have written, and programs need to easily and seamlessly access that data. Similarly, the sinks for the data may include visualization graphics like tables, graphs, maps, data streams for other programs, web API updates, or dynamic web pages.
2. A person may pick up code they haven't looked at for a week - or code written by somebody else - and we want them to be able to make sense of it quickly. This will be exacerbated by one of our design principles: we plan to do ongoing A/B usability testing of the user experience, interface and language, and use the results to drive change, so that program might not look exactly like it did last week!

Because of this sporadic usage pattern, traditional programming languages and code structuring approaches are both overkill and difficult for our target users to learn. Therefore, the environment is designed to support very fluid relationships among chunks of code, in terms of their relationship and evolution, as well as very fluid interactions among code chunks when processing data.

The nature of our user/programmer base provides opportunities and challenges. Opportunities because, if we can make it transparent enough, evolving the language and the environment is possible. Challenges because, if we don't make the environment seamless and transparent enough, nobody will ever use it!

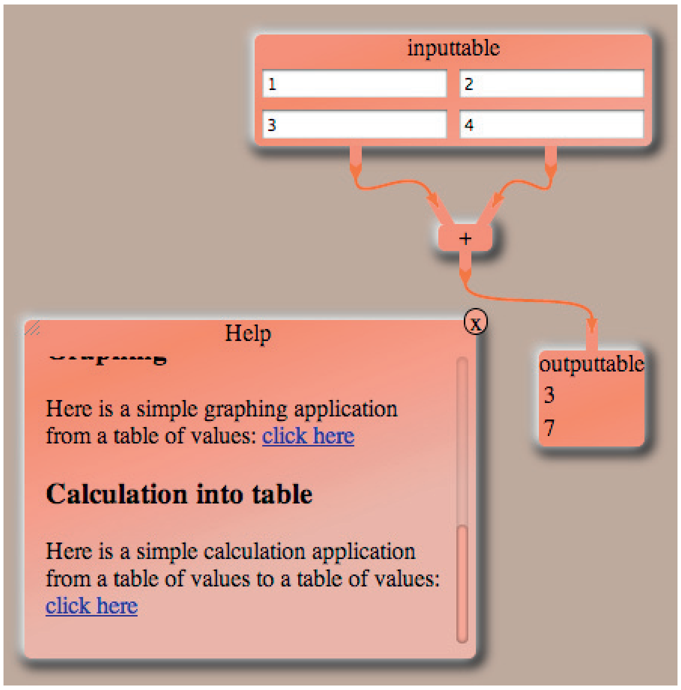
2. Language and Code Chunks

The user-programmer is presented with a veritable blank slate,[1] and code chunks can be built up in arbitrary order. Code and data sources/sinks are represented as 2-dimensional glyphs representing operators and data values. Data flow is represented as queues or streams - even scalar data values provide constant streams. Looping control structures appear to be unnecessary, as sequences of data provide intrinsic iteration/mapping operations. The language itself is not particularly remarkable, apart from its visual representation and our efforts to make it intuitive and non-threatening. The language is heavily leveraged by the data options and the ways in which data can be fed into, and between blocks of code.

All program entities, whether they be datasets, data selectors, code, or visualizations (all of which we'll refer to as blocks) are treated similarly as they each have a set of named inputs and/or a set of named outputs. These inputs and outputs can be connected via pipes. This is designed to be very dynamic, such that any input/output pair can be connected at any time (possibly with renamers/transducers to make them match). Typing is dynamic - if some block needs a number or a boolean, it will block when the wrong type is presented.

Figure 1 shows a small example of the environment, with a couple of connected tables and a help window. The example code was loaded from the link in the help page.

[1]The user interface is expected to be under active evolution for the foreseeable future, so details about user interface should be considered to be, at best, a snapshot.



Here is a simple graphing application from a table of values: click here

Calculation into table

Here is a simple calculation application from a table of values to a table of values: click here

Fig. 1. Connected Tables

2.1. *Execution Model*

The execution model is primarily demand-driven(2) - a block will not execute unless all of its outputs are required, and all of its input values are available.[2] Any blocked input will provide visual indication that it is preventing progress, and the programmer may choose to connect a queue from that input to some data source, at which point execution will proceed to the next blocked input.

However, the model is affected by our goal of tranparency and responsiveness. At any point in execution, the value of inputs (or indeed, program constants) can be modified (see §3.2) using selectors appropriate to their type. Currently this forces a re-execution of every block that interacts with the modified value, but in the future we may consider tagged dataflow models(3) as a more efficient way of handling this.

2.2. *Composition*

The Deutsch Limit(4, 5) suggests a limit of 50 primitives on-screen at once,[3] before user confusion sets in. We address this in 3 ways.

1. A named block group can be created from any group of blocks. Named block groups have input and output buffers. The input buffers provide an input from outside the group and any number of outputs to the blocks within the group. The output buffers will merge inputs from any other blocks within the group and produce a single output to outside the group. Once created, the internals of the group are hidden, and the entire group is represented as a single named block, functionally equivalent to a primitive. At any point, a block group can be opened and editted, then re-closed.
   Figure 2 shows a simple calculation of the absolute value of a stream of values from an input table, whereas 3 shows that same calculation enclosed in a group block and minimized. Note that a minimized block is interchangeable with a primitive block, differentiated only in that clicking on the block offers an option to see the block in its expanded form.

[2]There are a few kinds of blocks that follow different firing rules, as explained below.
[3]Code/data blocks and pipes are primitives.

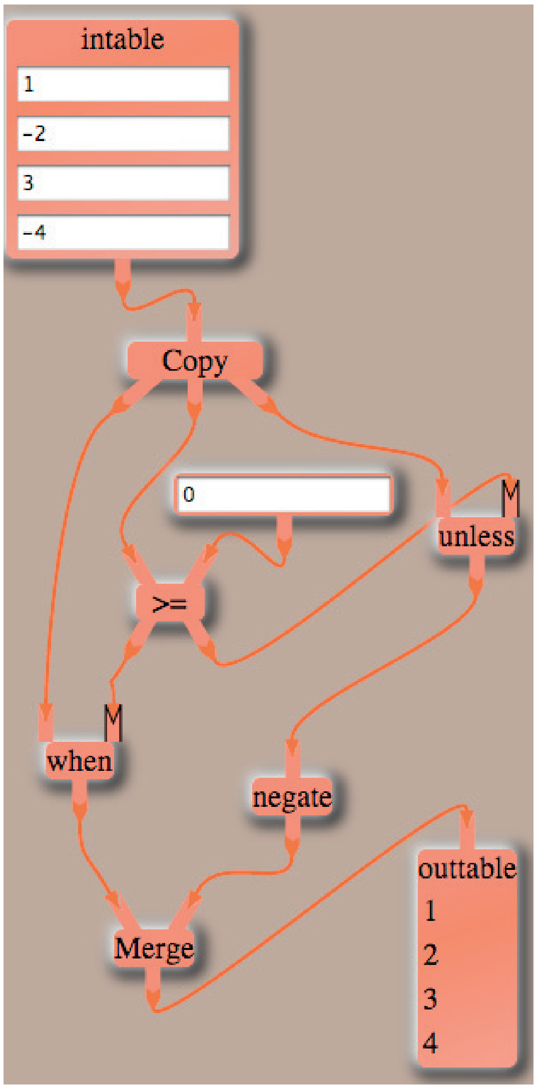


Fig. 2. Expanded View

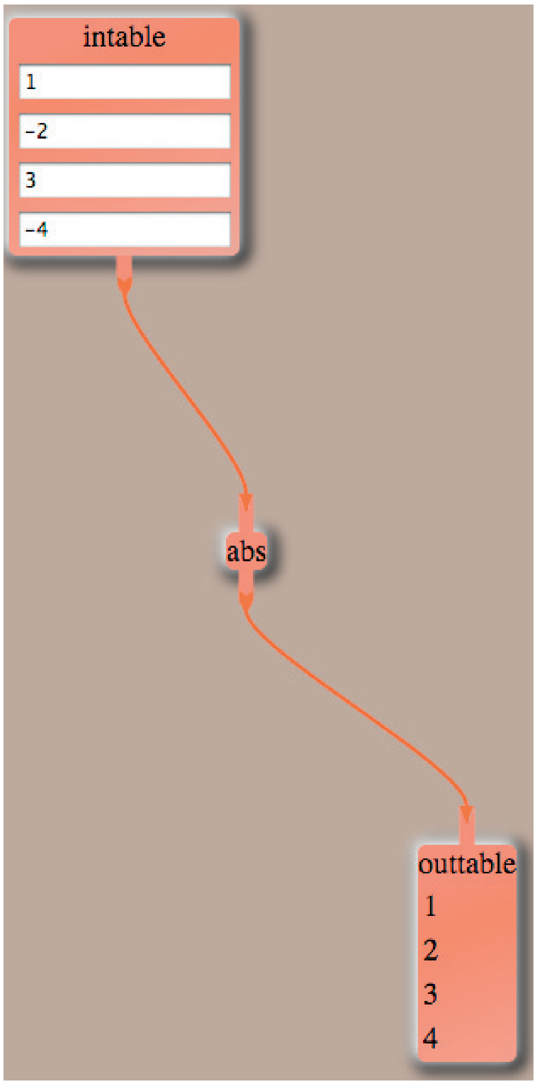


Fig. 3. Minimized View

2. When a "complete program" – a named block containing sources, code, and visualizations – is executed, the code can be made to hide, leaving the screen to the inputs and visualizations. Of course, the "program" can be opened and examined/modified at any point – including observing intermediate values as they flow through the program.
3. Group blocks can be parameterized over other blocks (whether primitive or groups) so that some dataflow structure can be saved and at some point applied to other functions or to themselves. The exact semantics of recursive application remain to be worked out.

The result of these decisions is that the visual field can be as clean as desired, and this form of mental confusion should be minimized. "Programs" can also be loaded into web pages (e.g. blogs) with the output (graph, table, etc.) calculated upon load with whatever dynamic state of the world exists at that moment, and can also expose input values to parameterize the output.

3. Sources, Sinks, and Interaction

The goal of this project is to allow easy programming of ubiquitous data. This will typically involve accessing data from a variety of external sources, processing that data to extract information, and then displaying that generated information in either a graphical or textual form.

3.1. Sources

To support this goal, there need to be a wide selection of input sources.

1. Firstly there are simple data inputs. They have input fields and visual representation appropriate to their type, and can be manipulated with a range of appropriate selectors (colour wheels, sliders, etc.) and control points. There are also tables of such values. Any time any value is modified all output dependent on that value is recalculated.
2. Next, there are traditional row-based data sources, such as spreadsheets, NoSQL databases. These are represented with a URL reference and a query (where appropriate), and each field has a name.
3. JSON Web APIs, such as Twitter or Facebook. These are represented by their URL, the API reference, and any credentials necessary to access them.
4. Web page scraping, such as accessing a table from a wab 1.0 website. These are represented by their URL, a description of how to find the field/table, and any credentials necessary to access them.
5. XML documents, parsed by XPath(6). These are represented by their URL, a description of how to extract each desired column, and any credentials necessary to access them. This approach can also be used to access real-time monitored data.
6. Output values from another code block executing within this image. Potential race conditions are automatically detected – where there are conditionals or loops among sources and sinks for multiple code blocks – and transactions implicitly added to eliminate them.

3.2. Interaction

In section §2.1 we mentioned that constants and simple data values can be modified "at any point" in the execution. This refers to both temporal and evaluative "points".

Firstly you can modify values at any temporal point in program execution, that is, part-way through execution. Blocks will then be re-evaluated as necessary to effect the change throughout the system, forcing recalculation of all dependent values, including visualizations.

Additionally you can modify such data values at any evaluative point in program execution, that is, at any point you can observe the original value. This means that if a value has been passed through a variety of blocks and queues unchanged, so that the original value presents itself at the input to some block or is displayed as part of some visualization, then the original value can be changed from that point in the system with exactly the same effect as if it had been changed at the point of creation. This facilitates very exploratory interaction with the data and the system.

To enable this interaction, as well as to assist in debugging, the user can decide to make any queue *stepped*, so that user interaction is required each time a value is requested.

3.3. Sinks

The sinks for the data are also diverse and will (eventually) include:

1. Graphs. A low-level *x-y* graph primitive takes a stream of graph primitives (Move(x,y), Line(x,y), Symbol(s), etc.). This allows this basic graph to be augmented with normal code blocks to support charts, turtle graphics, and other forms of graph. In graphs, simple data values (as mentioned in item 1) display as control points and can be directly manipulated, causing the original value to be changed and a revised graph to be drawn. Graphs can be composed with their origins aligned so that multiple series of values can be overlaid.
2. Tables. In tables, simple data values (as mentioned in item 1) display as any other data values but can be directly manipulated, causing the original value to be changed and a revised table to be generated. Tables can be composed horizontally and vertically to create arbitrary table structures.

3. Maps. Many of the big data sources have a geographical component, so it is natural to provide mapping visualizations. The primitive map will support selecting a colour for a region. This allows the basic map to be augmented with normal code blocks to support selecting a colour for a region based on the sum of values, pie-charts, and other forms of region-based graphics.
4. Web applications. We want to be able to create web resources as well as consume them, so we expect to support API updates, such as posting a visualization to an image storage site, tweeting or posting Facebook updates, as well as generating dynamic web pages.

4. Code Evolution and Reuse

For our target audience, we assume much programming will come from copying and modifying existing sources, sinks, and code – even more than for traditional programmers. To facilitate this, even very complex objects are intended to be implemented within the language as assemblies of very low-level primitives. At any point any block can be cloned and possibly modified. This is similar to that in Self(7) and Javascript(8).

At any point the user can also examine the variances in code of code blocks (not necessarily with a cloning relationship), with the possibility of bringing the code into a more coherent model. The idea is to allow refactoring, and bring the values of class-based coding style, without the up-front commitment implicit in that style.

5. Related Work

There are many efforts underway to enable users to access the vast pool of data at our electronic fingertips.

Search engines, such as Google(9), when presented with a search such as "correlation between streetlights and potholes" are excellent at finding what *other people* have already discovered, but can't create such information themselves.

Systems such as Wolfram's Alpha(10) and Apple's Siri(11) may aspire to answer such questions but - for now - simply answer, "no match found".

Drools(12) is a production rule system (a form of expert system) that can be used to extract information from a knowledge base - but it is too arcane for our target audience.

IFTTT(13) provides very rudimentary programming relating to web events and actions - but only those events and actions that someone else has programmed.

Yahoo Pipes(14) provides a somewhat similar, server-based, programming environment.

The programming model most closely related to ours is that of dataflow programming.(15, 16) We have attempted to take the best ideas of systems like Prograph(17), combine them with ideas from Scratch(1), and bring them into a modern, web/touch-device interface.

Principles of design and structure inspired by Papert(18) and Victor(19, 20). While Papert is talking about teaching children math, the principles of embodiment and feedback are key to understanding for any novice programmer. Victor points out that traditionally we expect programmers to simulate program execution in their heads, which is an especially high expectation for our users.

An earlier report on this work(21) was presented at the FREECO-2012 workshop.

6. Conclusions

Creating a language in which non-progammers will be happy programming creates many opportunities and challenges.

Non-programmers will not have the same investment in a particular syntax/semantics, which allows for language evolution in a way completely inconceivable for traditional languages and their programmers.

Conversely, non-programmers will not as easily recognize earlier code, so it is imperative that the data connections and language representation be as accessible as possible.

Of course, we can't know in advance if people will use the environment for programming in the large, but we are initially explicitly making decisions to preferentially support casual and small-scale approaches wherever a decision must be made - i.e. we are attempting to grow the language using agile principles.

References

[1] K. A. Peppler, Y. B. Kafai, Creative coding: Programming for personal expression (2005).

[2] W. M. Johnston, J. R. P. Hanna, R. J. Millar, Advances in dataflow programming languages, ACM Comput. Surv. 36 (1) (2004) 1–34. doi:10.1145/1013208.1013209.
URL `http://doi.acm.org/10.1145/1013208.1013209`

[3] Arvind, D. Culler, Dataflow architectures, Annual Review of Computer Science (1986) 225–253doi:10.1146/annurev.cs.01.060186.001301.

[4] B. Y. Ricardo, Visual programming (2002).

[5] Deutsch limit, `http://en.wikipedia.org/wiki/Deutsch%5flimit`.

[6] W3C, Xpath.
URL `http://www.w3.org/TR/xpath/`

[7] D. Ungar, R. B. Smith, Self: The power of simplicity, SIGPLAN Not. 22 (12) (1987) 227–242. doi:10.1145/38807.38828.
URL `http://doi.acm.org/10.1145/38807.38828`

[8] A. Guha, C. Saftoiu, S. Krishnamurthi, The essence of javascript, in: ECOOP, Lecture Notes in Computer Science, Springer, 2004, pp. 40–52.

[9] Google search engine, `http://google.com/`.

[10] Wolfram alpha, `http://www.wolframalpha.com/`.

[11] Apple Siri, `http://www.apple.com/ios/siri/`.

[12] JBoss, Drools.
URL `https://www.jboss.org/drools/`

[13] IFTTT: If This Then That, `https://ifttt.com/wtf`.

[14] Yahoo, Yahoo pipes.
URL `http://pipes.yahoo.com/pipes/`

[15] W. R. Sutherland, The on-line graphical specification of computer procedures, Ph.D. thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering (1966).

[16] M. Boshernitsan, M. Downes, Visual programming languages: A survey, Tech. Rep. UCB/CSD-04-1368, Computer Science Division, EECS University of California, Berkeley (Dec. 2004).

[17] S. Matwin, T. Pietrzykowski, Prograph: a preliminary report, Comput. Lang. 10 (2) (1985) 91–126. doi:10.1016/0096-0551(85)90002-5.
URL `http://dx.doi.org/10.1016/0096-0551(85)90002-5`

[18] S. Papert, Mindstorms : children, computers, and powerful ideas, 2^{nd} Edition, BasicBooks, New York, USA, 1993.

[19] B. Victor, Magic ink: Information software and the graphical interface, `http://worrydream.com/MagicInk/` (2006).

[20] B. Victor, Inventing on principle, in: Canadian University Software Engineering Conference, Montreal, Canada, 2012.

[21] D. Mason, Flexible structures for end-user programming, in: Proceedings of the 3rd international workshop on Free composition, FREECO '12, ACM, New York, NY, USA, 2012, pp. 9–11. doi:10.1145/2414716.2414720.
URL `http://doi.acm.org/10.1145/2414716.2414720`