Open Piping: Towards an Open Visual Workflow Environment

BOISVERT, Charles <http://orcid.org/0000-0002-3069-5726>, ROAST, Chris <http://orcid.org/0000-0002-6931-6252> and URUCHURTU, Elizabeth <http://orcid.org/0000-0003-1385-9060>

Open Piping: Towards an Open Visual Workflow Environment

Charles Boisvert *Chris Roast* *Elizabeth Uruchurtu*

Department of Computing. Communication and Computing Research Centre.
Sheffield Hallam University, City Campus, Howard Street, Sheffield S1 1WB, UK
initial.surname@shu.ac.uk

Abstract

The most popular visual programming tools focus on procedural, object-oriented and event-based programming. This paper describes a boxes-and-wires functional programming tool, aimed to be accessible to novice programmers, while also supporting open access to the specified processes, executable programs and results for study and deployment.

Keywords Computer science education, data science, functional programming, end-user programming

1 Introduction

Visual, block-based environments such as ALICE [4] or Scratch [17] have recently transformed the teaching of computing [9, 1].

Yet this development in procedural and object-oriented programming tools has not disseminated to analysing and processing data. For example, the nifty assignments repository of computing assessment ideas [15, 14] contains 107 assignments, collected for their quality, but only eight of these incorporate work with a real data set.

Of particular concern to us, at Sheffield Hallam University, is adapting our tools, teaching methods and resources in order to facilitate access to and process of data by students at any level. Specific interest areas have been working with open data advocacy groups [12] and making data analytic tools more available [19].

The Open Piping project pursues this idea with an open-source functional programming environment and visual data flow interface for data processing[1].

[1] http://boisvert.me.uk/openpiping

2 Project motivations

Open Piping is a visual functional programming environment, based on a boxes and wires model, intended for data processing applications.

Visual boxes and wires environments are common [11, 13], including some in commercial [8] and scientific [7] use. But in many cases, the value of the tools is limited due to the poor transparency of the processes and technology they implement.

Take the case of the popular - until its end in 2015 - Yahoo pipes [13]. To execute pipes on systems of their choice, users had to go through a complex export process. This was their only option when Yahoo support ended.

Open piping aims to propose an ease of use comparable to commercial tools, in an open architecture to facilitate development flexibility, reuse and allow richer exchanges between users.

2.1 Open by design

Our ambition is to propose a graphical tool for user-defined data processes, which would include, by design, the transparency and flexibility needed to apply user-defined processes in a range of languages and environments. Open piping aims to be at once:

Open. That is, Open Source; the system's source code is available under the GNU licence. But so is the notation used to define processes. Any user process can then be transformed from this notation into executable code in a target programming language.

Interoperable. The process specification format is openly available, and uses a human-readable, JSON formatted S-expression. This is needed to ensure the interoperability of the system with any manner of services, such as alternative end-user interfaces, new languages or process hosting and remote execution tools.

Easy to use. The user interface makes it easy to define data flows and shows clearly the relation between data flow, resulting S-expression, and executable functionality.

With resulting processes easy to deploy. The ability to choose from multiple languages and standards for services and content integration, would facilitate the re-use of user-defined processes in different environments, such as within content-management systems, as web or application widgets, or within a service-oriented architecture.

Altogether, these characteristics aim to ensure that users can easily define the processes they want to operate on data, while also retaining control of these processes to use them in new environments.

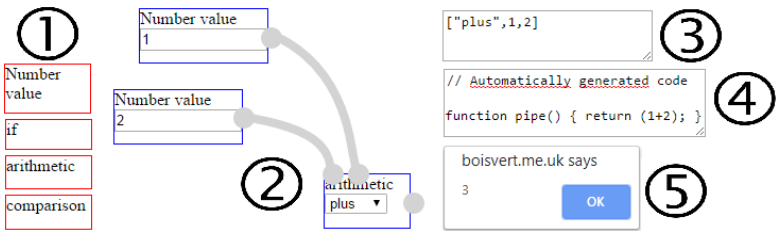

Fig. 1: Open piping main interface elements

3 Open piping Operation

3.1 System architecture

The boxes-and-wires model describes the directed acyclic graph for a function, with the boxes representing functions and the wires, the data to which they apply.

Configuration data defines base functions available to the end-user. This information at once determines primary graphical blocks, provides access to basic processing capabilities, and limits that access, for security, to a chosen set with defined functionality.

The end-user defines a function by wiring elementary blocks. This function is translated into an S-expression in JSON, which can be compiled into an executable function in any number of languages, provided that calls to the primitive functions can be defined.

The interface elements presented fig. 1 sum up the use of Open Piping. The end-user chooses elementary blocks (1) to define a flow (2) which is translated to a symbolic expression (3) encoded in JSON to use the many existing tools for this format. The expression is then interpreted (4) and executed (5).

3.2 Defining and encoding a data flow

The block description and interface configuration also uses the JSON format. For instance, fig. 2 shows the configuration lines to define the box representing arithmetic operations. The user can choose add, subtract, divide, or multiply from a single ‘arithmetic’ box.

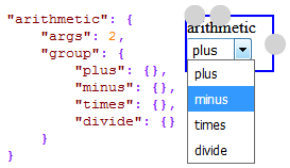

Fig. 2: Defining and representing graphically a box of arithmetic functions

An example data flow is presented fig. 3. The web interface uses the JSPlumb library [10] to manipulate and represent the screen objects. Traversing the graph recursively provides a symbolic expression. An advantage of symbolic expressions is that code

remains close to existing languages such as LISP or Scheme. For instance, in a LISP-like language, the workflow figure 3 results in the structure:

```
(if (isNumber 15) (plus 1 15) "not a number")
```
[1]

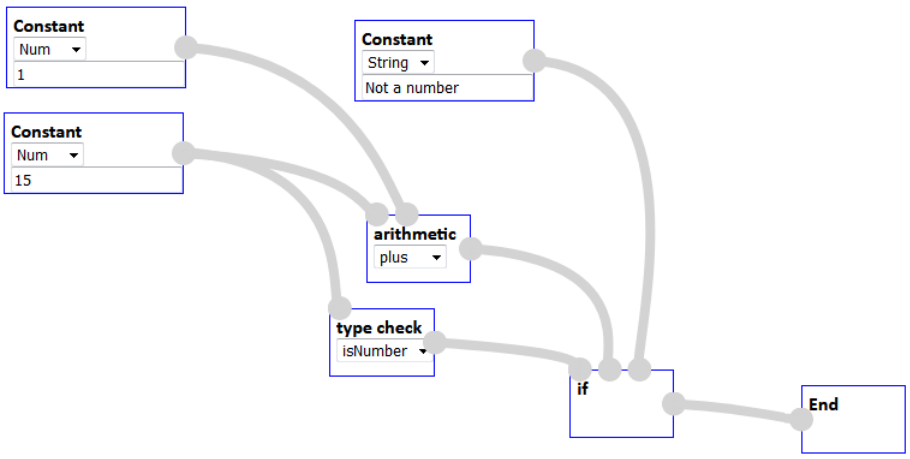

Fig. 3: An example workflow

Another benefit of S-expressions is the original argument for this notation: executable code and data follow the same conventions. This facilitates the processing of an expression like line [1] in multiple environments.

The expression is encoded in JSON, to provide to the interpreter. JSON's wide use and readability make it particularly suitable to this purpose. The encoding follows these simple rules:

- JSON notation defines *objects*, *arrays*, *strings*, *numbers* and the values `true`, `false`, and `null`. Our encoding relies on all but *objects*.
- Atomic values are *strings*, *numbers* and the values `true`, `false`, and `null`.
- Lists are represented by a JSON array. Each element of the list can be an atomic value or a list, and so on recursively.

Respecting this convention, the process shown fig. 3 is written:

```
["if", ["isNumber",15], ["plus",1,15], "Not a number"]
```
[2]

3.3 Interpret a symbolic expression in executable language

To allow the execution of the same expression in diverse environments, we rely on characteristics present in most programming languages - use of variables, of a means of conditional execution, of functions - but we must provide elementary information to

support the interpretation in each language. These data are themselves written in JSON.

To illustrate the interpretation process, let us study the case of interpreting expression [2] above in JavaScript and JQuery.

The interpretation relies on a list of predefined functions and string substitutions for the language:

```
"plus": {"args": "a,b", "sub": "(@a+@b)"}
"if": {"args": "a,b,c", "sub": "@a?@b:@c"}                    [3]
"isNumber": {"args": "n", "body": "return $.isNumeric(n);"}
```

Some operators are interpreted by substituting character chains to form the target code. Arithmetic operators like + use this technique, but so do conditionals, which we interpret in JavaScript with the ternary operator. Functions are identified and composed from arguments and body information. So [4] contains all the information needed to interpret the example completely.

Using this data, the expression is interpreted recursively. First the expression

```
["isNumber", 15]                                              [4]
```

results in the definition of function `isNumber`,

```
function isNumber(n) {return $.isNumeric(n);}                 [5]
```

and into one function call. The `plus` function is then interpreted by substituting strings, and finally `if` to compose the overall result:

```
process(isNumber(15)?(1+15):"Not a number");                  [6]
```

We can see that the interpretation of a user-defined function is simple; to be able to execute a process in a given language, we simply need to define and execute safely the primitive functions required.

3.4 Overcoming visual limitations

The graphical model shown above should support end-user's understanding and programming of simple processes. However, based on our experience and prior research such as [18, 3, 2], we speculate that several aspects of the visualisation are not easily represented in ways that end-users spontaneously understand. Here, we present a number of potential solutions to support end-users as programs become more complex.

3.4.1 Coordinating visual code with results

Visual programming can support end-users with a number of displays - the results of a program, of its code, of its execution. The wires and boxes model is a form of visual code, but many systems show a visual representation of execution results.

Coordinated views can also apply to viewing code. Yahoo pipes [16] is an example of this approach: its visualisation showed code, in boxes and wires form, along with a sample of the data resulting from it. Users could also select subsets of the code to view its result. This supported end-users with a presentation of the code, of some results, and of execution information (as partial execution results), as well as debugging support by means of choosing code subsets to test.

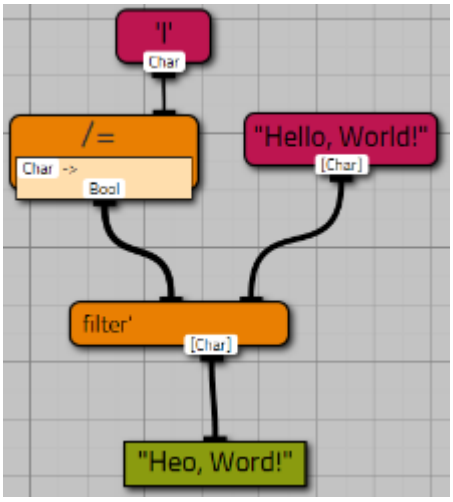

Fig. 4: Viskell shows data type explicitly

3.4.2 Data Typing

The boxes and wires model shown in our example fig. 3 does not show any type information. Typing has many advantages for novice programmers, in particular limiting errors by constraining the validity of constructs, ensuring security, and facilitating debugging.

Typing can be presented in textual form, a solution adopted by Viskell as shown in fig. 4 [20]. An alternative is visual clues, such as colour, shape, or icons: languages like MIT Scratch [17] adopt this approach, and use the added advantage of shape as a metaphor for syntactic validity. Type can also be implemented in the language and enforced in the interaction, yet not presented visually: that is the solution adopted by Yahoo pipes, which enforces type checking with the impossibility of connecting a wire to a box if types do not match, but give no visual typing clue.

3.4.3 Representing Conditionals

Conditional execution is one of the basic elements of programming. A three-argument function, for the Boolean that determines which branch is executed, and each of the

two branches, is a suitable technical answer, but as the prototype workflow shown earlier in fig. 3, visual clues in support of the user are clearly lacking.

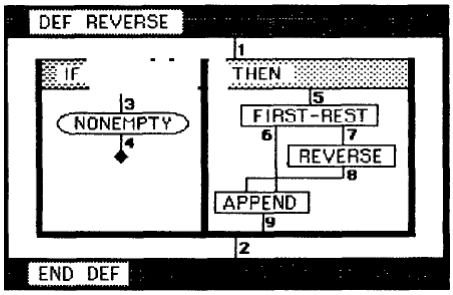

Fig. 5: Prograph shows the conditional branches within two frames for clarity.

Prograph [5] solves this problem by adding to boxes and wires a third construct, *frames*, for sections of code that are end-users should consider separately.

3.4.4 First class functions

First-class functions are a fundamental benefit of functional programming, but also a difficult concept to represent in ways that users can understand and control. The earlier illustration of Viskell (fig. 4), shows a lambda-expression within the model, supported by textual type annotation: not every end-user will find it clear.

An alternative relies on the same notion of frames as for conditionals: a function that accepts another as a parameter, represents that parameter within a frame. Yahoo pipes adopts that solution, albeit for a limited use of first-class functions: it implements user actions to drop a box into a functional parameter slot. [6] have investigated the primitives needed to represent completely the power of first-class functions within frames, but the solution is not an easy visualisation of the notion.

4 Conclusion and future work

The structure of our system lets users retain control of their processes. In particular:

Limits to processing capabilities are not inherent to the system, but instead to the environment in which the process is deployed, for example by setting a processing time limit.

The visual language is loosely coupled to the execution environment, by producing a function definition in an open intermediate representation; this ensures that changes to the visual interface, to the target language, and to the execution environment are independent.

Risks of code injection are limited by transmitting the symbolic expression to an interpretation environment hosted with the execution environment, rather than communicate executable code, as well as by defining in the interpreter what primitive functions are allowable.

We believe that these characteristics can support adoption and self-learning through greater open access to computation.

Currently our prototype ensures that end-users can define processes, and demonstrates the compilation from the S-expression to JavaScript and execution. Multiple environments common on web servers and clients are considered - e.g. JQuery, PHP, node.js, etc, as well as deployment of executable results in new systems.

Developing this prototype's capabilities to support users further, will require a balance of technical feasibility, theoretical clarity and empirical evidence to identify the most appropriate solutions.

References

[1] D. Adshead, C. Boisvert, D. Love, and P. Spencer. Changing culture: Educating the next computer scientists. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pages 33–38. ACM, 2015.

[2] A. F. BLACKWELL. Pictorial representation and metaphor in visual language design. *Journal of Visual Languages Computing*, 12(3):223 – 252, 2001.

[3] A. F. Blackwell. The reification of metaphor as a design tool. *ACM Trans. Comput.-Hum. Interact.*, 13(4):490–530, Dec. 2006.

[4] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[5] P. Cox and T. Pietrzykowski. Advanced programming aids in prograph. In *Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pages 27–33. ACM, 1985.

[6] A. Fukunaga, W. Pree, and T. D. Kimura. Functions as objects in a data flow based visual language. In *Proceedings of the 1993 ACM Conference on Computer Science*, CSC '93, pages 215–220, New York, NY, USA, 1993. ACM.

[7] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl 2):W729–W732, 2006.

[8] N. Instruments. What is labview. http://www.ni.com/en-gb/shop/labview.html. Accessed: 2019-30-04.

[9] S. P. Jones, T. Bell, Q. Cutts, S. Iyer, C. Schulte, J. Vahrenhold, and B. Han. Computing at school. *International comparisons. Retrieved May*, 7:2013, 2011.

[10] I. JSPlumb. Jsplumb toolkit documentation. https://jsplumbtoolkit.com/docs.html. Accessed: 2017-13-04.

[11] D. Le-Phuoc, A. Polleres, G. Tummarello, and C. Morbidoni. Deri pipes: visual tool for wiring web data sources. *)ˆ(Eds.):'Book DERI pipes: visual tool for wiring web data sources'(2008, edn.)*, 2008.

[12] M. Love, C. Boisvert, E. Uruchurtu, and I. Ibbotson. Nifty with data: Can a business intelligence analysis sourced from open data form a nifty assignment? In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 344–349, New York, NY, USA, 2016. ACM.

[13] T. O'Reilly. Pipes and filters for the internet. http://radar.oreilly.com/2007/02/pipes-and-filters-for-the-inte.html. Accessed: 2016-10-10.

[14] N. Parlante. Nifty assignments. http://nifty.stanford.edu. Accessed: 2016-01-12.

[15] N. Parlante, J. Popyack, S. Reges, S. Weiss, S. Dexter, C. Gurwitz, J. Zachary, and G. Braught. Nifty assignments. In *ACM SIGCSE Bulletin*, volume 35, pages 353–354. ACM, 2003.

[16] M. Pruett. *Yahoo! pipes*. O'Reilly, 2007.

[17] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[18] C. Roast, R. Leitão, and M. Gunning. Visualising formula structures to support exploratory modelling. In *Proceedings of the 8th International Conference on Computer Supported Education*, CSEDU 2016, pages 383–390. SCITEPRESS - Science and Technology Publications, Lda, Portugal, 2016.

[19] C. Roast, D. Patterson, and V. Hardman. Visualisation —- it is not the data, it is what you do with it. In P. Kommers and P. Isaías, editors, *e-Society 2018 Conference Proceedings*, pages 231–238. IADIS, 2018.

[20] F. Wibbelink. Interacting with conditionals in viskell. 2016.