What Do We Really Know about Data Flow Languages?

Guido Salvaneschi

TU Darmstadt, Germany

salvaneschi@cs.tu-darmstadt.de

Abstract

Over the last years, a number of languages based on data flow abstractions have been proposed in different important areas including Big Data, stream processing, reactive programming, real time analytics. While there is a general agreement that the data flow style simplifies the access to such complex systems compared to low level imperative APIs, this design has been substantiated by little evidence.

In this paper, we advocate a systematic investigation of the design principles of data flow languages and suggest important research questions that urge to be addressed.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords reactive programming, controlled experiments

> *I really wanted a PL that supports a language-integrated interface [..] because I thought that was the way people would want to program these applications [...]*
>
> Matei Zaharia, Apache Spark creator

In recent years, a number of programming languages have been proposed for processing event streams [1], real time data and batch data [10] that share a common design principle: modelling the computation with data flow abstractions rather than the traditional imperative, control flow abstractions [5]. Following this principle, programmers specify how values depend on each other (data dependencies) and the runtime is responsible for propagating changes. Despite its popularity – the data flow style is predominant e.g., in Big Data analytics where *data scientists* without a CS background are often exposed to this style in the first place – little research has been devoted to this solution to evaluate its advantages based on factual data.

It is convenient to review the principles shared by data flow languages. They **favour data flow** over control flow – control structures are abandoned, except inside lambdas. They are **declarative**. Programmers provide a high level specification of data processing instead of defining the steps for computing a value. They adopt a **functional programming flavour** that fosters composabililty and abstracts over state. Also it provides an advantage both to (1) easily express a computation as a pipeline of operators, and (2) to simplify parallelism, distribution and fault tolerance. They **hide the complexity** of the underlying framework (e.g., to automatically tracking dependencies among values, to achieve fault tolerance and replication).

Data flow languages: Examples Data flow languages have been applied in very diverse areas. **Streams for collection libraries** are available e.g., in Scala and Java 8. A variety of sources, such as in-memory collections or databases can be accessed uniformly. The following code filters a stream of strings selecting only those starting with "c", transforms them to uppercase letters and collects the result into a list:

```
List<String> l = Arrays.asList("a1", "c2", "b1", "c2");
  l.stream()
   .filter(s -> s.startsWith("c"))
   .map(String::toUpperCase)
   .sorted()
   .collect(Collectors.toList)
```

In the area of animations and user interfaces, **reactive programming** [1] is a programming paradigm that aims at supporting reactive applications with dedicated language abstractions (*e.g.,* event streams, signals). Rx is a library for asynchronous reactive programming where producers push values to consumers that process the values as soon as they are available. This code snippet[1] takes a value out of 10 from the network and prints string-converted batches of 5 values:

```
getDataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed" })
  .subscribe({ println "onNext => " + it })
```

The data flow style has been adopted also in **batch and real time analytics**, where it provide means to process large amounts of data on clusters of commodity hardware abstracting over low level details such as parallel execution and fault tolerance. These systems include PigLatin, Dryad, Spark and Spark Streaming. For example, the following code[2]

[1] http://reactivex.io/intro.html

[2] http://spark.apache.org/examples.html

implements the word count application in Spark and caches intermediate results in memory:

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .cache
                .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Despite its popularity, to date, the advantage of a spread adoption of the data flow style is far from obvious. For example, providing highly specialised versions of operators comes at a price: The Rx reactive framework lists more than 70 core operators (up to ∼400 including their variants). Another aspect concerns the semantics of data flows that is often subtle. Spark supports the same operators as Scala collections (e.g., `filter`), but Spark operators are lazy, Scala collections are strict. A similar mismatch can be observed between Java 8 streams and JavaRx, which have similar operators. Yet, Java 8 streams are pull based (an input collection is consumed upon stream creation), JavaRx is push based (operators process data as soon as data are created). Another issue concerns the level of abstraction adopted by the language. For example, in Spark one can `cache/persist` a stream to explicitly reify intermediate computations – a functionality not available in other frameworks. Approaches based on lifting, like Java 8, require the programmer to build potentially verbose abstract representations – a "receipt" of the computation – opened by `.stream` and converted back to a concrete collection with e.g., `collect(Collectors.toList)`.

The case of Reactive Programming In previous work, we started investigating the effect of data flow abstractions in the context of reactive programming [4]. We organised a controlled experiment with 38 subjects that was later repeated, extending it to 127 subjects. The experiment evaluated the impact of the reactive programming style on software comprehension compared to the Observer design pattern.

The results, based on 10 different applications in a between subjects design, show that correctness of comprehension is enhanced in the reactive style, and comprehension time does not increase (i.e., reactive programming does not trade accuracy for time).

Research questions We advocate similar studies that systematically investigate the design of data flow languages and provide guidelines to developers of future data flow languages. Research on the usability of such languages should answer, in particular, the following questions:

- Is the data flow style more comprehensible [7] than the imperative approach or pure functional programming?
- What is the effect on comprehension of semantics variations like push/pull, cached, lazy/eager evaluation and how easily do programmers master such concepts?
- What is the effect of the data flow style on API protocols [8]? It is simpler for programmers to understand and obey protocols in this style?
- A rule of thumb in language design is to make the common case easy to express, tolerating complexity for rare cases. This is especially an issue for data flow languages where specialised operators make the API much harder to master. Do data flow languages provide a "simple enough" solution for the common case without excessive proliferation of overspecialised operators?
- Expressing computations with data flows involves mixing the right operators in a combination that is specific for the problem at hand. Are such highly specialised pipelines more difficult to modify correctly than imperative code? What is the effect of the data flow style on maintainability?
- What is the effect of the data flow style on other activities in the development process besides text based programming such as debugging [6] and software design? For example, previous research found that visual programming does not improve readability for data flow languages [2].
- Does the data flow style influence the mental models [3] that programmers develop about software or impacts their approach to software comprehension?

To the best of our knowledge, only a few works [2, 4] have investigated usability of data flow languages. We believe that to answer these questions researchers need different approaches including qualitative studies to assess how developers work, repositories and forum mining, as well as controlled experiments that connect the data flow style to well known cognitive theories on program comprehension [9].

References

[1] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[2] T. R. G. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. In *In*, pages 167–180, 1992.

[3] T. D. LaToza and B. A. Myers. Developers ask reachability questions. ICSE '10. ACM, 2010.

[4] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. An empirical study on program comprehension with reactive programming. FSE 2014. ACM, 2014.

[5] G. Salvaneschi, P. Eugster, and M. Mezini. Programming with implicit flows. *IEEE Software*, 31(5):52–59, 2014.

[6] G. Salvaneschi and M. Mezini. Debugging for reactive programming. ICSE '16. ACM, 2016.

[7] J. Siegmund and J. Schumann. Confounding parameters on program comprehension: A literature survey. *Empirical Softw. Engg.*, 20(4), 2015.

[8] J. Sushine, J. D. Herbsleb, and J. Aldrich. Searching the state space: A qualitative study of API protocol usability. ICPC '15. IEEE Press, 2015.

[9] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. ICSE '94. IEEE, 1994.

[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. SOSP '13. ACM, 2013.