All Things Flow

Unfolding the History of Streams (extended abstract)

Aggelos Biboudis
Swisscom AG
Switzerland

Jeremy Gibbons
University of Oxford
United Kingdom

Oleg Kiselyov
Tohoku University
Japan

Abstract

Heraclitus observed that all things flow and nothing remains still; "you cannot step into the same river twice". So what is a *stream* in computer science, and where did this notion come from? We divide streaming abstractions into four categories: a) as a means of processing lots of data in limited memory; b) as event processing and correlation; c) to capture the semantics of I/O; and d) as iteration abstractions. Following these four axes, we unfold the history of streams, and give an overview of how this abstraction started to come into existence as a mainstream programming language facility. Our goal is to present briefly the related concepts through literature review, drawing connections between programming language features and technologies. This discussion will be of interest to the young computer science researcher, the curious software engineer, and the grizzled database query optimization specialist.

1 Facets of Streams

Over several decades programming languages have shifted away from the sequential programming model that the von Neumann architecture so vigorously imposed [2]. Nowadays, instead of *commands* and static *storage*, programmers often have to think in terms of processes and (high-performance) transformations over *streams*. By streams we, in computer science, mean a sequence of elements that can be piped through a series of transformation steps. Characteristically, data is accessed in strict sequence rather than in a random access pattern. In return for limited expressiveness, we gain the opportunity to process large amount of data efficiently, in limited space.

Streams for data processing in sublinear space. One of the most recognizable examples of streams is Unix pipes [23, 24], instigated by McIlroy. He recorded his vision on a yellowed sheet of paper kept pinned to his office wall [18]: "We should have some ways of connecting programs like garden hose – screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also." This image of a 'hose', carrying, and transforming potentially infinite amount of data is evocative of all streaming libraries. A library user, like in assembling a hose, gets to declare *what* activities take place but not *how* the data access is scheduled. We can trace this thread back at least to Conway's design [5] for a one-pass COBOL compiler, whose modules are like segments of a hose.

This massaging – analyzing and transforming – large amounts of data is the domain of the traditional database management (see §3), which has evolved into so-called *data analytics*: performing Extract–Transform–Load jobs (ETL) over centralized architectures called *data lakes*. ETL consists of moving data from heterogeneous sources (E) to other targets (L), after several transformation steps (T). Streams are promoted to first-class status [26].

Streams as event processing and correlation. *Information flow processing* is the other class of stream processing applications. It is defined as "processing continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries" [7]. Its characteristic example is *complex event processing*: given incoming notifications of events observed by sources (e.g., sensor readings), "filter and combine such notifications to understand what is happening in terms of higher-level events to be notified to sinks, which act as event consumers." Intrusion detection, environment monitoring, and online analysis of stock prices are clear examples. A less obvious example is complex user-interfaces such as modern GUI and multimedia applications.

Information flow processing thus encompasses dataflow, reactive and signal processing – often captured under the name of *stream processing systems* [29]. It also has deep roots to Kahn process networks [13]; networks where concurrent processes communicate only through one-way FIFO channels with unbounded capacity. Dataflow networks are usually considered a special case of Kahn networks. The first dataflow language, Lucid [30], appeared in 1970s. The two modes of executions arise: data driven (eager evaluation) where computation depends on the availability of data; and demand driven (lazy evaluation) where the agents request data from their inputs. These terms are commonplace in programming language semantics, modern streaming libraries and more.

Streams to capture the semantics of I/O. Landin [15], in 1965, was the first to observe that a denotation of ALGOL-60's `for`-statements with `for`-lists leads to a peculiar list processing where the items of an intermediately resulting list "never exist simultaneously." Noting the similarity with Conway's approach, he coined the term "stream". Much later

streams have been used to give the 'pure functional' semantics of I/O in early Haskell [11]. The popular 'trace semantics' is also based on streams [12].

We connect the dots between lists (and their constructors), streams (and their destructors), recursion over finite lists, and co-recursion over infinite streams [25].

Streams as iteration abstractions. In order to understand the nature of streams as programming abstractions for *iteration*, we have to take a good look at the past. We trace the lineage of streams from FORTRAN [3] and IPL [21] to LISP [17], Common LISP [28] and Clojure [10]. The former already include powerful mechanisms for streaming computations: sequences, streams, loops and series. We also look at the early abstractions of these mechanisms: iterators in CLU [16], a form of generators as in IPL-V [20] and Alphard [27]. Alphard in particular was aimed at the development of verifiably reliable software. Abstraction was indispensable in simplifying and modularizing Hoare-style proofs. Since iterative computations invariably loop over the elements of some collection (be it as simple as a range of integers), it made sense to separate operations on the current element (the loop body) from obtaining the next element (the loop control). The goal was to hide the details of the collection, encapsulate the state of the enumeration, and separate concerns in the proof.

These abstractions appear again and again, from modern C++, for example, offering a multitude of iterators explicitely or even implicitly through "smarter" `for`-loops, to C# and Python's first-class support for generators. We revisit the historical milestones that gave shape to the semantics of streaming APIs from the perspective of programming language abstractions, and we argue that the same patterns emerge in other disciplines of computer science. We believe that a concise and focused study of the history of streams will guide the engineer in navigating the design space effectively. In the full paper we complement the conceptual discussion with an analysis of how streams took a form as a linguistic construct in several programming languages.

2 Motivating Example 1: Streaming APIs

Java 8 introduced lambdas (i.e., anonymous functions) with the explicit purpose of enabling streaming abstractions, which present an accessible, natural path to multicore parallelism – perhaps the highest-valued domain in current computing. Other languages, such as Scala, C#, and F#, also support lambda abstractions and streaming APIs, making streams a central theme of their approach to parallelism. Although the specifics of each API differ, there is a core of common features and near-identical best practices for users of these APIs in different languages. However, with a closer look, we can also identify key differences in the designs [4]. For example, Java 8 introduced streams in a different way from other ecosystems, drawing inspiration from Lisp and Smalltalk. We will discuss the two topics of external versus internal iteration as the two dual dataflow representations that appear in the literature of programming languages and systems, two terms that appear frequently under different names: external vs. internal, pull vs. push and data-driven vs demand-driven.

Finally, what is a representation of streams that fully captures the generality of streaming pipelines and allows desired optimizations? To understand how the representation affects implementation and optimization choices, we review past efforts in deforestation [31] and stream fusion [6].

3 Motivating Example 2: Database Systems

Streaming libraries and query engines in database systems share many ideas. Both areas offer a high level API, describing low-level operations over data that needs to be executed efficiently. A query expression is typically translated by a query engine into a physical plan of execution. These plans consist of (physical) operators that consume data stored in tables, in a streaming fashion using composed iterators over tuples. A database query optimizer, after cost analysis and reordering of operators, determines the most efficient physical plan. This plan, akin to a pipeline in streaming libraries, specifies how to traversing data efficiently without accumulating intermediate values. Query optimization relies on the same core idea of a dataflow representation [22]. For example, the dominant in the 90's Volcano model [8, 9] implements the iterator model (external iteration). In 2011, Thomas Neumann [19] proposes an alternative model where "instead of pulling tuples up, we push them towards the consumer operators" – which has inspired the prominent framework for big data, Apache Spark [1].

This work unfolds the history of iteration and streams, following the lineage from mathematics to other sub-disciplines of computer science. We start from Brouwer's intuitionistic analysis. We trace streams as a mathematical abstraction for the construction of infinite sequences and argue that it was Brouwer who fully grasped streams. He was handicapped however by the lack of suitable notation, which only came to light with coinduction. Tracing these foundations all the way until the present-time, we lay the ground for an interesting, holistic discussion over the principles behind such a pervasive topic.

References

[1] Sameer Agarwal, Davies Liu, and Reynold Xin. Apache Spark as a compiler: Joining a billion rows per second on a laptop. https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html, May 2016.

[2] John Backus. Can programming be liberated from the Von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[3] John Backus. The history of Fortran I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages*, pages 25–74. Association for Computing Machinery, 1978.

[4] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Clash of the lambdas. In *Proc. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '14, 2014.

[5] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963.

[6] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326. ACM, 2007.

[7] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3), June 2012.

[8] Goetz Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[9] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. of the 9th International Conference on Data Engineering*, ICDE '93, pages 209–218. IEEE Computer Society, 1993.

[10] Rich Hickey. A history of Clojure. *Proceedings of the ACM on Programming Languages*, 4(HOPL), June 2020.

[11] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Alan Jeffrey and Julian Rathke. The Lax Braided Structure of Streaming I/O. In Marc Bezem, editor, *Computer Science Logic*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 292–306, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[13] Gilles Kahn. A Preliminary Theory for Parallel Programs. Research Report R0006, 1973. Rapport IRIA.

[14] Gilles Kahn. A preliminary theory for parallel programs. 1973.

[15] Peter Landin. Correspondence between Algol 60 and Church's lambda-notation: Part i. *Commun. ACM*, 8(2):89–101, February 1965.

[16] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, August 1977.

[17] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, August 1978.

[18] Doug McIlroy. Advice. https://www.bell-labs.com/usr/dmr/www/mdmpipe.html, October 1964.

[19] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[20] Allen Newell. Documentation of IPL-V. *Commun. ACM*, 6(3):86–89, March 1963.

[21] Allen Newell, editor. *IPL-V Programmers Reference Manual*. RAND Corporation, Santa Monica, CA, 1963.

[22] Holger Pirk, Jana Giceva, and Peter R. Pietzuch. Thriving in the no man's land between compilers and databases. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019.

[23] Dennis M. Ritchie. The evolution of the Unix time-sharing system. In Jeffrey M. Tobias, editor, *Language Design and Programming Methodology: Proceedings of a Symposium Held in Sydney, Australia, 10–11 September, 1979*, pages 25–35. Springer Berlin Heidelberg, 1980.

[24] Dennis M. Ritchie and Ken Thompson. The Unix time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[25] J. J. M. M. Rutten. Behavioural differential equations: A coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1-3):1–53, 2003.

[26] Gwen Shapira. The future of ETL isn't what it used to be. https://www.confluent.io/blog/the-future-of-etl-isnt-what-it-used-to-be/, June 2017.

[27] Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, August 1977.

[28] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[29] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[30] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., 1985.

[31] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, January 1988.