ISSN 2186-7437

NII Shonan Meeting Report

No. 2018-14

Functional Stream Libraries and Fusion: What's Next?

Aggelos Biboudis
Oleg Kiselyov
Martin Odersky

October 22–25, 2018

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Functional Stream Libraries and Fusion: What's Next?

Organizers:
Aggelos Biboudis (EPFL, Switzerland)
Oleg Kiselyov (Tohoku University, Japan)
Martin Odersky (EPFL, Switzerland)

October 22–25, 2018

Stream processing defines a pipeline of operators that transform, combine, or reduce (even to a single scalar) large amounts of data. Characteristically, data is accessed strictly linearly rather than randomly and repeatedly—and processed uniformly. The upside of the limited expressiveness is the opportunity to process large amount of data efficiently, in constant and small space.

Stream processing underlies one of the first programming languages (COBOL), and it has been coming to forefront again, with the popularity of big data and MapReduce. The widespread automated trading is one example of extremely high-performant stream processing; software-defined radio is another such example. Widely-used stream libraries are now available for virtually all modern OO and functional languages, from Java to C# to Scala to OCaml to Haskell, for small and big data. Streams also motivate a vast range of topics in computer science such as dependent types, termination and partial evaluation. Streams stress virtual machines and CPU memory access paths, and pose challenges for compiler writers and hardware designers.

Functional stream libraries let us easily build stream processing pipelines, by composing sequences of simple transformers such as map or filter with producers (backed by an array, a file, or a generating function) and consumers (reducers). The purely applicative approach of building a complex pipeline from simple immutable pieces simplifies programming and reasoning: the assembled pipeline is an executable specification. To be practical, however, a library has to be efficient: at the very least, it should avoid creating intermediate structures – especially structures like files and lists whose size grows with the length of the stream. Even the bounded-size intermediate structures significantly, up to two orders of magnitude, slow down the processing. Eliminating the intermediate structures is the central problem in stream processing: so-called *stream fusion*.

Stream fusion has been the subject of intensive research since late 1950's. By now, the low-hanging fruit in stream processing has been all picked up – although some of it though quite recently, POPL 2017. Stream fusion made it to the front page of CACM[1]. Java 8 Streams and Haskell compilers, among others, have implemented some of the earlier research results. We have attained

[1] `https://cacm.acm.org/magazines/2017/5/216312-exploiting-vector-instructions-with-generalized-stream-fusion/fulltext`

a milestone. What are the further challenges?

We thus propose a discussion-heavy workshop to assess the state of the art and to answer:

1. Can the recent advances in 'traditional' stream fusion be extended to non-traditional domains such as GPGPU, heterogenous architectures (FPGA)? Can the methods of stream fusion be applied to functional reactive programming (FRP) and optimizing database queries?
2. Can we bridge functional and effectful stream libraries?
3. What are the next big challenges? Which real world scenarios require new advances in stream fusion?

Just as the two Shonan meetings (No.2012-4 "Bridging the Theory of Staged Programming Languages and the Practice of High-Performance Computing" and No. 2014-7 "Staging and High-Performance Computing: Theory and Practice") aimed to solicit and discuss real-world applications of assured code generation in HPC (High-Performance Computing) that would drive programming language research, we likewise aim to compile the case studies requiring further research in stream fusion, and distill them into a benchmark.

To promote mutual understanding, we plan for the workshop to have lots of time for discussion. We will emphasize tutorial, brainstorming and working-group sessions rather than mere conference-like presentations.

List of Participants

The following people have participated in the seminar, beside the organizers.

1. Sven Bodo-Scholz, Heriot-Watt University (Scotland)
2. Timothy Bourke, INRIA & ENS, FR
3. Peter Braam, Consultant
4. Oliver Bračevac, Technical University Darmstadt (DE)
5. Shigeru Chiba, University of Tokyo (Japan)
6. Jeremy Gibbons, University of Oxford (UK)
7. Christoph Koch, EPFL (Switzerland)
8. Ugo Dal Lago, Universitè degli Studi di Bologna
9. Ben Lippmeier, University of New South Wales (AU)
10. Geoffrey Mainland, Drexel University (US)
11. Hidehiko Masuhara, Tokyo Intitute of Technology (JP)
12. Trevor L.McDonell, Utrecht University, NL
13. Marc Pouzet, Université Pierre et Marie Curie & École normale supérieure
14. John Reppy, University of Chicago (US)
15. Amir Shaikhha, EPFL
16. Jeremy Yallop, University of Cambridge (UK)

Main Questions

The following points and questions were raised repeatedly during the seminar. *Push and pull duality* appears in many guises in stream processing research and also in database community, as it became clear during the meeting. By now researchers know the differences and what challenges each design faces and designers can choose from a combination of techniques combining different properties. For example Strymonas applied the pull approach with elements taken from push-based composition. Ben Lippmeier, deals with the problem that short-cut stream fusion cannot fuse a producer into multiple consumers. It was argued that fusion order is significant and both pull and push must co-exist. Similarly pull semantics appeared in John Reppy's Second-Order Array Combinators (SOAC) where the combinators use a pull thunk parameterised by the array indices, to get their inputs. The Ziria stream design follows a pull-based philosophy at the core of its design.

Another question was: how do the different technologies implement stream *error handling*? Peter Braam raised this question and in his presentation the reasons why this is important were made explicit. A full imaging pipeline implemented in Cloud Haskell means that errors can bring down whole nodes. What is the re-execution protocol in this situation? It was argued that stream error handling or clocked networks probably enable this easily. From his point of view, an iteratees/pipes-based DSL seemed the right direction (reinvented as he said).

Furthermore, streams are not *control aware* in general: Ziria solved this. In that language two combinators were first-class. A take and an emit. Strymonas also raises the need to treat take in a special way. Again, in Ziria the consumer/producer model has appeared. Hardware appears to implement simple consumers and producers.

Throughout the whole meeting *sparsity* of data was mentioned by Peter Braam, Trevor McDonell, John Reppy. Sparsity of data appears on many tensor libraries but not in mainstream programming frameworks for stream processing. It needs special handling specially when mapping computations on GPU cores.

Tangible Outcomes

The following challenges where described as a possible set of problems that we can use to assess the expressivity of different designs.

filterMax. This function is the core of the QuickHull algorithm. The latter takes a line and an array of points, and finds the farthest point above the line. `filterMax` demonstrates that you cannot perform fusion without some sort of tupling transformation [1].

```
filterMax: Vector Int -> (Vector Int, Int)
filterMax: xs0
  = let xs1 = map (+1) xs0
        xs2 = filter (>0) xs1
        m   = fold max 0 xs2
    in (xs2, m)
```

Listing 1: filterMax

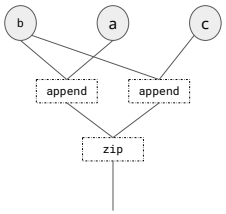
Sorted Merge. This demonstrates a potential limitation for Synchronous Dataflow Programming languages like Lustre. The question arisen concerns the base clock of the computation. This operation is the corner stone of databases and it would be interesting to answer questions like this.

```
merge [1, 4, 8, 10] [2, 4, 6, 11] = [1, 2, 4, 4, 6, 8, 10, 11]
```

Listing 2: Sorted Merge

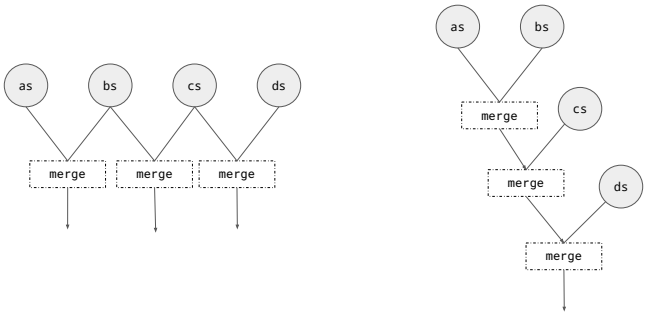
Multiple Appends. Another issue that demonstrates the need for modular and multiple scheduling strategies is outlined by the multiple appends problem (Figure 1).

Figure 1: Multiple appends



Parallel Merge. Figure 2 shows two cases of a parallel merge. On the left fusing this with Machine Fusion results in a process where the number of states is exponential in the number of merges. On the right, merge is a sorted merge, takes two sorted streams and it interleaves the elements so that the result is also sorted.

Figure 2: Parallel Merge



Meeting Schedule

October 22 (Monday)

Theme: Introductions, background tutorials

- Self-introductions
- A brief history of streams (Aggelos Biboudis)
- Lucid, Lustre and its descendants (Marc Pouzet)

October 23 (Wednesday)

Theme: Stream fusion, schemes and technologies

- Compiling Ziria to Hardware (Geoffrey Mainland)
- Machine Fusion is not Associative (Ben Lippmeier)
- What if — we considered our streams arrays? (Sven Bodo Scholz)
- Versatile Event Correlation with Algebraic effects (Oliver Bračevac)
- Streaming data compression (Jeremy Gibbons)
- Pull vs. Push-Based Loop Fusion in Query Engines (Amir Shaikhha)

October 24 (Thursday)

Theme: Streams on the big scale, Excursion

- Software for the SKA telescope (Peter Braam)
- Streaming with Nessie (John Reppy)
- Excursion and Main Banquet

October 25 (Friday)

Theme: Streams on the GPU, Conclusions

- Is this even a good idea? (Trevor McDonell)
- Vélus: A formally verified compiler for Lustre (Timothy Bourke)

1 Overview of Talks

A brief history of streams

Aggelos Biboudis, EPFL

Stream. What is a stream? Where is the start and where is the end? By re-interpreting Heraclitus, all things pass and nothing remains still. Everything can be viewed as a river's flow: you cannot step into the same river twice. So what is a stream? Is it the data itself? What if the data are in–memory? What

if it is the pipeline itself? Is it the series of combinators or is it an iterator that you cannot traverse twice? Is an iterator, the only form of traversal? Is the stream pushed to us by an external force or do we control the stream by letting the stream flow towards us according to our will? This presentation makes an attempt to set up the high-level terminology and give an overview of how streams started to come into existence as a programming language abstraction.

Lucid, Lustre and its descendants

Marc Pouzet, Université Pierre et Marie Curie & École normale supérieure

A tutorial on the origins, motivations, development and the state of the art of synchronous data-flow languages.

Compiling Ziria to Hardware

Geoffrey Mainland, Drexel University

Ziria offers a programming model based on a monadic language for specifying and composing stream processors. It enforces clean separation between control and data flow for the implementation of for implementing protocols for Software Defined Radios.

Machine Fusion is not Associative

Ben Lippmeier, University of New South Wales

Machine fusion is a new (2017) technique for fusing networks of functional stream combinators. The combinators themselves are written in a simple imperative language and can both pull-from and push-to input/output streams. The fusion technique can fuse networks that contain both splits and joins, where multiple combinators in the network read data from the same input stream. This is unlike simpler short-cut fusion techniques that depend on inlining, and thus do not work when an input stream is referred to multiple times. Unfortunately, the primitive operator that fuses two combinators is not associative (or commutative), which means there are many possible orders in which to attempt to fuse a network of many combinators. Not all fusion orders are valid, and we currently don't have a good way of determining the valid orders ahead of time. The current implementation uses heuristics and tries many possible fusion orders, but this is unsatisfying and risks combinatorial explosion for large networks.

What if — we considered our streams arrays?

Sven-Bodo Scholz, Heriot-Watt University

Arrays and streams seem to be fundamentally at odds: arrays require their size to be known in advance, streams are dynamically growing; arrays offer direct access to all of their elements, streams provide direct access to the front elements only; the list goes on. Despite these differences, many computational

problems at some point require shifting from one paradigm to the other. The driving forces for such a shift can be manifold. Typically, it is a shift in the task requirements at hand or it is motivated by efficiency considerations. In this talk, I provide the key details of our concept of "Transfinite Arrays" which bridges the gap between arrays and streams. I try to inspire a discussion on where the bridging might fail by showing how some of the typical streaming problems can be expressed in the transfinite array setting.

Versatile Event Correlation with Algebraic effects

Oliver Bracevac, Technical University Darmstadt

In recent work, we contribute an extensible language design for uniformly expressing variants of n-way joins over asynchronous, push-based event streams from different domains, e.g., stream-relational algebra, event processing, reactive and concurrent programming. We model asynchronous reactive programs and joins on top of algebraic effects and handlers. Join variants can be considered as cartesian product computations with "degenerate" control flow, such that unnecessary tuples are not materialized a priori (e.g., one does not need to materialize all tuples in order to "zip" two streams). Based on this computational interpretation, we decompose joins into a generic, naive enumeration procedure of the cartesian product, plus freely composable, variant-specific extensions, represented in terms of user-defined effect handlers. The latter specialize the enumeration procedure to obtain a desired materialization behavior.

In this talk, I will give an overview of our language design. Furthermore, while stream fusion is now well-understood for pull-based stream pipelines, I would like to have discussions on how to optimize push-based pipelines, especially in conjunction with coroutine-style programming models, e.g., as enabled by algebraic effects and handlers.

Streaming data compression

Jeremy Gibbons, University of Oxford

A wide class of problems can be characterized as changes of representation. I propose to talk about changes of representation for stream processing - transforming a stream of input symbols into a stream of output symbols, and in particular two data compression techniques, arithmetic coding and asymmetric numeral systems.

Push versus pull-based loop fusion in query engines

Amir Shaikhha, EPFL

Database query engines use pull-based or push-based approaches to avoid the materialization of data across query operators. In this talk, we study these two types of query engines in depth and present the limitations and advantages of each engine. Similarly, the programming languages community has developed loop fusion techniques to remove intermediate collections in the context of collection programming. We draw parallels between databases (DB) and

programming language (PL) research by demonstrating the connection between pipelined query engines and loop fusion techniques. Based on this connection, we propose a new type of pull-based engine, inspired by a loop fusion technique, which combines the benefits of both approaches. Then, we experimentally evaluate the various engines, in the context of query compilation, in a fair environment, eliminating the biasing impact of ancillary optimizations that have traditionally only been used with one of the approaches. We show that for realistic analytical workloads, there is no considerable advantage for either form of pipelined query engine, as opposed to what recent research suggests.

Software for the SKA telescope

Peter Braam, Consultant

An overview of the SKA telescope, its technical specification and software requirements.

Streaming with Nessie

John Reppy, University of Chicago

Nessie is a NESL-based compiler for GPUs that supports offloading sparse operations in a streaming fashion.

Is this even a good idea?

Trevor McDonell, Utrecht University

A tutorial on the principles and design behind Accelerate.

Specifying and Verifying a Lustre Compiler in an Interactive Theorem Prover

Timothy Bourke, INRIA & ENS

Tools like Scade and Simulink allows engineers to develop and validate systems at the level of abstract "block diagrams" that are automatically compiled into executable code. Behind these diagrams is a programming model based on (synchronous) dataflow streams combined with imperative features like hierarchical state machines.

We have been working on a verified compiler for Lustre—-the academic counterpart of SCADE-—in the Coq interactive theorem prover. This project involves specifying the high-level stream-based semantics, programming the compilation passes that successively transform Lustre programs into the Clight input language of the CompCert compiler, and showing that repeatedly executing the assembly code produced by CompCert recreates the streams associated with the input program.

This talk will focus on the stream-based semantic models we have developed, give an overview of the compilation process and the proof of its correctness, and

outline remaining challenges and open questions for both compiler and program verification in interactive theorem provers.

The work described is the result of collaborations with Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg.

References

[1] Hu, Z., Iwasaki, H., Takeichi, M., and Takano, A. Tupling calculation eliminates multiple data traversals. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 1997), ICFP '97, ACM, pp. 164–175.