Declarative Distributed Stream Processing

PhD Stage 1 Progression Report

Jonathan Dowland <jon.dowland@ncl.ac.uk>

June 2019

Background

Many modern applications, in domains ranging from smart cities to healthcare, have a requirement for the timely processing of data arriving at high speed, for example that generated by sensors in the Internet of Things (IoT). Such systems may need to meet a range of other requirements, including: reliability; security; energy efficiency, for example to prolong the battery life of sensors in the field; or privacy, to remove or de-personalise data prior to transmitting it over open networks.

The combination of requirements, very high data arrival rates, and the desire for timely processing, makes the design and management of the supporting infrastructure very challenging. The current generation of IoT tools adopt the principles of stream processing and are designed around a three-tier architecture: sensors generate data, which is sent on to a local gateway (e.g. a smartphone or embedded device) to be collated before being passed on to the Cloud for processing.

However, in some domains it can be beneficial to perform some processing on the gateway or on the sensors themselves [7], to reduce the volume of data sent onwards to the cloud; or to reduce the frequency with which sensors must invoke their networking hardware, thus reducing energy expenditure; or to avoid transmitting sensitive data across public networks, by filtering or anonymising data sets at the point of collection.

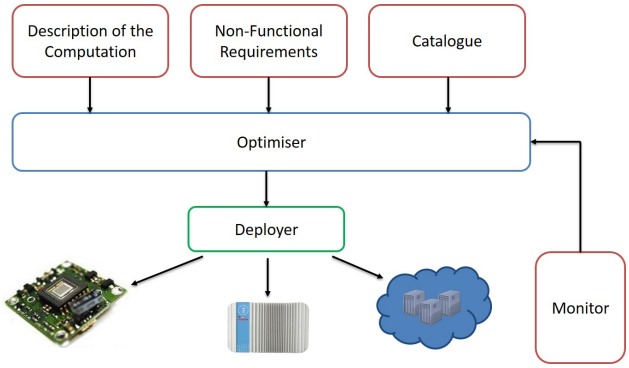

Figure 1: StrIoT architecture diagram

Foundations

We are exploring an alternative approach: a system whereby the stream processing operations and the non-functional requirements are described declaratively as inputs to an Optimiser, which automatically determines the most appropriate deployment onto the available resources, which may include sensors and

gateways. Monitoring of the deployment is used to evaluate the performance of the Optimiser and could also be used for run-time adaptation. The initial approach (by PhD student Peter Michalák [7]) used an extended version of SQL as the method of describing the computation.

In contrast, in our project we are exploring using functional programming to describe the computation. Using the pure functional language Haskell, a prototype has been developed named "StrIoT" [15] where the stream processing is defined in terms of a restricted set of functional operators with well-understood semantics. This prototype has two distinct components: Library code to support stream processing in which segments of the stream are spread across multiple compute nodes; and the Optimiser.

My research will be focussed on the optimiser and deployer (another PhD student — Adam Cattermole — is working on the stream processing library).

A high-level overview of the StrIoT architecture is provided in Figure 1.

Research Question

Does purely functional programming offer any benefits for the design and implementation of distributed stream-processing systems?

Purely-Functional Programming

Functional Programming (FP) is a software development paradigm where the principle building blocks of programs are functions (as oppose to e.g. abstract objects in Object-Oriented Programming) and programs are composed declaratively using expressions, rather than sequences of statements.

Advocates of FP believe that many of its properties have advantages for the design and implementation of large and complex software systems [3]. Constructing systems declaratively results in the programmer describing *what* a system should do, rather than the minutiae of *how* the work should be performed.

Functional programming languages often have strong type systems which help to catch and prevent a large class of programming errors at compile time. Strong, expressive type systems can aid with the legibility of code, with function type signatures serving as a form of documentation as to the behaviour of the function.

Purely-functional programming is a variant where the behaviour of functions is entirely defined in terms of their input arguments and output value when evaluated and can perform no other actions (referred to as *side-effects*). As a consequence pure functions are easier to reason about.

Referential transparency is a property of pure functions where an expression in a program can be safely substituted by any other which evaluates to the same value.

Referential transparency enables a performance optimisation known as *memoisation*, where the value of an expression can be recorded in a look-up table when first calculated, avoiding the need for repeated evaluation where the expression is referenced more than once.

Referential transparency also enables *equational reasoning*, a technique for transforming functions through a process of substitution by applying laws or rules [1]. An example law is provided in Figure 2.

$$map\ f\ .\ map\ g = map\ (f\ .\ g)$$

Figure 2: The law of map fusion

Purely-functional programming allows for *lazy evaluation*: a strategy where expressions are only evaluated at the point at which their value is required (such as when a value is to be printed) [4]. Conversely an expression is not evaluated if the value is not required: this permits programs to manipulate expressions that might be infinite, such as the set of natural numbers.

Functional Stream Processing

My research aim is to establish to what extent the advantages of purely-functional programming can be applied to the design and operation of a distributed stream-processing system, subject to the challenges outlined in the "Background" section, above.

Modern distributed stream-processing systems attempt to separate the functional definition (typically specified as a software program) and the non-functional requirements (the deployment environment and constraints such as power requirements, network utilisation limits, etc.). The declarative nature of purely-functional programming allows for a high degree of abstraction. I will investigate whether this enables the construction of a system where the user can specify the functional behaviour of the program declaratively and independent of the deployment and non-functional requirements.

In order to meet non-functional requirements, it may be necessary to adjust the stream-processing specification provided by the user whilst preserving the functional behaviour. Equational reasoning is a powerful tool for encoding program transformations and can be used to build rewriting systems [9]. I will investigate whether rewrite rules are expressive enough to encode and apply useful transformations for stream-processing optimisation.

A distributed stream-processing system needs to partition and distribute a stream-processing definition onto individual nodes. I will investigate whether any facets of purely-functional programming are particularly beneficial for the design and implementation of the Partitioner, and how automated partitioning should interact with the Optimiser.

Work Completed

Engineering

I have re-engineered and expanded an existing proof-of-concept implementation of a functional stream-processing system. I reorganised the source code to follow community conventions and made it available to the public under an Open Source license at GitHub [14]. I introduced a test harness for the system to provide assurance that it behaved correctly, and to verify that subsequent refactoring work preserved correct behaviour.

I introduced a third-party Graph library [8] to describe the stream-processing graph. The graph representation is based on a formal algebra of graphs which provides us with a strong formal foundation for further work on graph rewriting.

I expanded the Partitioner and Deployer to an end-to-end system that takes in a stream-processing graph and outputs source code and build instructions for the Docker container system [12]. These can be deployed to separate nodes (via a system such as Docker Compose [13]) and executed.

Derivation of stream rewriting rules

The StrIoT system provides a set of 8 functional operators from which a user can compose a stream-processing program. These operators have well understood semantics which should enable the Optimiser component to more easily reason about a stream-processing program. These operators are listed in the first column and row of the table in Figure 3.

I systematically compared the 8 functional operators pair-wise in order to derive some potential rewrite rules for the Optimiser. For each pair I attempted to find a rewrite that demonstrated an example of a well-known stream-processing optimisation [5]. This approach yielded 24 rules, enumerated in Figure 3.

A sample rule is provided in Figure 4. This is an example of a well-known functional optimisation [16] named *deforestation*, as it removes the overhead of intermediate list construction and deconstruction.

In order to gain further assurance about the validity of the 24 rules, I deployed the QuickCheck property-checker [2] to generate targeted test data sets tailored to the types of the operators.

	filter	map	filterAcc	scan	window	expand	join	merge
filter	1	-	3	-	-	-	-	8
map	9	10	11	12	13	-	15	16
filterAcc	17	-	18	-	-	-	-	-
scan	-	-	-	-	-	-	31	
window	-	-			-	-	-	40
expand	41	42	-	44	45	46	-	48
join	-	-	-	-	-	-	-	-
merge	57	58	-	-	-	62	-	64

Figure 3: Operator pairs that yielded rewrite rules. "-" indicates no rule was discovered

$$streamFilter\ q\ .\ streamFilter\ p = streamFilter\ (\lambda v\ \rightarrow\ p\ v\ \&\&\ q\ v)$$

Figure 4: example stream rewriting rule describing filter fusion

This exercise also yielded some promising rules which, whilst not total, provide insight as to possible design choices for the system, including rules that could be beneficial performance-wise but did not strictly preserve the order of incoming events.

$$streamFilter\ f\ (streamMerge\ [s1,\ s2]) =\ streamMerge\ [streamFilter\ f\ s1,\ streamFilter\ f\ s2]$$

Figure 5: A rewrite rule that does not preserve event ordering

One example of such a rule is provided in Figure 5: a filter downstream of a merge operation could be moved upstream. This could be advantageous for performance: by rejecting data in a stream earlier; or for cost: by reducing the volume of data transmitted over expensive network links. However, it results in a different ordering of events, which may or may not matter depending on the problem domain. In this example the extent of the reordering is known to be a function of the number of input streams to the merge operator.

Work Remaining

My future work will focus on the optimisation of distributed stream processing graphs.

Optimisation of the Stream Processing Graph

The immediate engineering work necessary is to complete an encoding of the stream rewrite rules yielded from the pair-wise comparison; design and implement an algorithm to apply them to a supplied stream-processing program; implement a cost model so that rewritten stream-processing programs can be evaluated for their fitness against non-functional requirements.

Once this initial system is complete we will be able to evaluate the effectiveness of the rewrite rules derived by pair-wise comparison. We can then look to see whether the system is capable of encoding rules representing all classes of well-known stream optimisations [5] as well as rules from other domains shown to be effective in prior work [7].

Other sources of optimisations may include those from the relational world, as well as possibly domain-specific rules, perhaps supplied by the user of the system as an additional input.

Looking below the level of Operators

As currently designed, the system requires the user to describe the stream-processing definition in terms of a Graph data structure within a Haskell program. The rewrite rules and the Optimiser only concern themselves with the semantics of the operators and their inter-connectivity. The system cannot reason about the semantics of the functional parameters supplied to the operators (e.g. the user-supplied predicate to `streamFilter`), which are presently represented as snippets of Haskell code embedded within string literals.

This likely rules out some classes of optimisation such as operator fission (separating a single stream processing operator out into two or more sub-operations). There are a number of alternative approaches that could be used for representing the stream-processing definition which may enable deeper reasoning. I intend to evaluate at least two of these: Template Haskell [11], a meta-programming tool; and Arrows [6], an alternative logical abstraction to Graphs that could be used to model the stream-processing computation.

Automatic Partitioning of the Stream Processing Graph

The present system requires the user to supply a predetermined partitioning scheme, specifying which functional operators shall be distributed to each node in the deployment environment. We will explore the automatic selection of a mapping by the Optimiser based on the non-functional requirements, a "catalogue" of available infrastructure for deployment (sensors, gateways, public/private cloud nodes, etc.), and a set of cost models predicting the cost of a specific deployment.

Run-time Performance

Initially, the Optimiser will generate a configuration of the input stream-processing graph once, based on the information available to it at "compile" time. However, we enhance this through collecting run-time performance information in order to evaluate the performance of the Optimiser and whether or not it has made an appropriately optimal assignment of operators and partitions. There are a number of ways that this could be achieved in a black-box fashion, depending on the specifics and features provided by the deployment environment.

It is likely desirable to be able to collect information on the performance of individual stream operators, perhaps amongst several deployed to a particular node. To achieve this, we may wish to extend the stream graph to add "taps" for collecting such data. If the system was to be extended to operate at run-time, this performance information could be used as an input for run-time recalibration of the stream processing graph, without dropping any data in transit from sensors. Further exploration is needed before we will know if it realistic for this to be done within the scope of this PhD.

Project Plan

I am working towards completing a PhD part-time and my milestones are scaled accordingly from a full-time schedule at a ratio of two calendar years to each PhD Stage. In addition to the end-of-stage milestones, I will present a progress report at the end of each intermediate year.

Balancing the demands of the PhD with my work and family commitments will be challenging, and careful planning is required.

The broad outline of my plan is summarized in the GANTT chart in Figure 7. I consider the project in four distinct phases: design; implement; test; results and write-up. At a high-level these phases are presented in a sequenced manner, as in the traditional Waterfall model of project planning.

I have not included a distinct Literature Review phase in the GANTT chart. I expect to continue to research and monitor relevant literature throughout the duration of the project. Each distinct project phase will require its own specific review, to be conducted primarily towards the beginning of each phase.

At a lower level, I intend to apply an agile project management approach [10]. This should provide rapid feedback as to the rate of progress and success for each intermediate objective, allowing small, frequent adjustments to be made to the ongoing plan.

At the time of writing, we have made significant progress towards a first paper outlining some initial results, and I have sufficient material to form the basis of a second paper. Extrapolating from this early stage progress I plan to attempt four academic publications over the duration of the project.

Risks and Mitigations

Identified risks and planned mitigations are summarized in Figure 6.

Risk	Impact	Likelyhood	Mitigations
Supervisor availability	High	Low	Remote-based workflow (e.g. email). Co-supervisor(s) focussing on different facets (A Mokhov; P Ezhilchelvan)
Data loss due to unexpected hardware loss or failure	High	Low	Personal data backup scheme; Software and written work stored on a public code sharing service (GitHub)
Research objective too ambitious	High	Low	Distinct low/medium/high risk objectives.
Failure to obtain improved stream graph from rewriting	Medium	Medium	Plan project around distinct basic/intermediate/advanced goals.
Relevance issues in fast-moving field	Medium	Low	Avoid specialising on a particular instance of technology

Figure 6: Project risks and mitigations

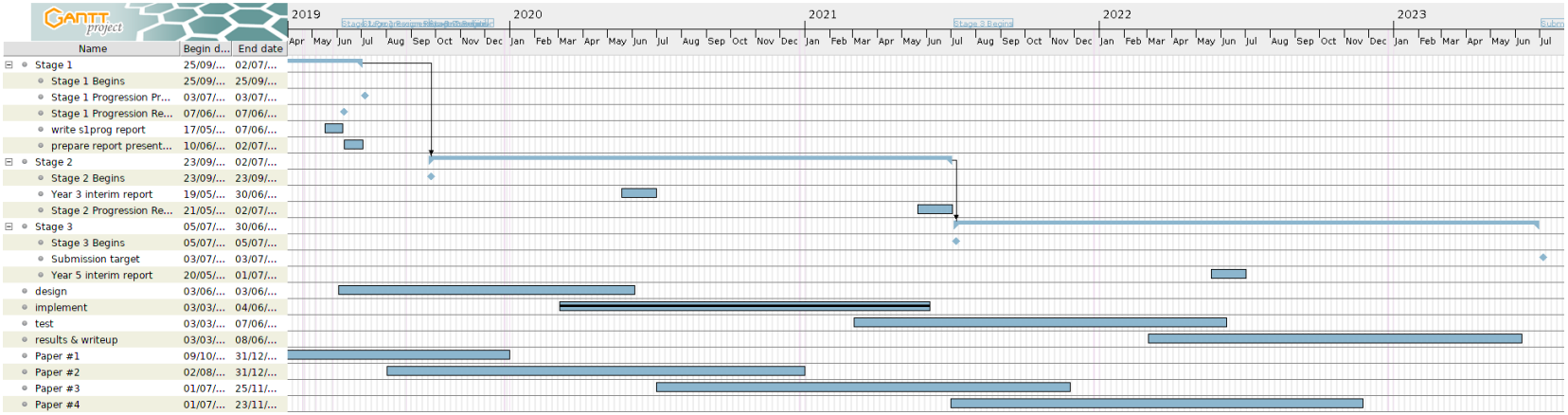

Figure 7: Project GANTT Chart

References

[1] R. Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2014.

[2] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[3] John R. G. Cupitt. *The Design and Implementation of an Operating System in a Functional Language (Miranda)*. PhD thesis, University of Kent at Canterbury, Canterbury, UK, 1989. AAIDX92465.

[4] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 95–103, New York, NY, USA, 1976. ACM.

[5] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46, 03 2014.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000.

[7] Peter Michalák and Paul Watson. Path2iot: A holistic, distributed stream processing system. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 25–32. IEEE, 2017.

[8] Andrey Mokhov. Algebraic graphs with class (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 2–13, New York, NY, USA, 2017. ACM.

[9] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. ACM SIGPLAN, September 2001.

[10] Laura Pirro. How agile project management can work for your research. *Nature*, Apr 2019.

[11] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. pages 1–16, October 2002.

[12] Various. Docker. https://docker.com/, 2019.

[13] Various. Docker compose. https://docs.docker.com/compose/, 2019.

[14] Various. Github. https://github.com/, 2019.

[15] Various. Striot. https://github.com/striot/striot, 2019.

[16] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP '88*, pages 344–358, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.