# Declarative Distributed Stream Processing

PhD Stage 2 Year 1 Report

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

July 2020

I have been studying part-time towards a PhD since 2017/18. This report marks the end of my 3rd year of study, which corresponds to half-way through Stage 2. In this report I outline the work completed during this academic year, the work remaining and the plan for continuing to completion.

## 1 Background

Here is a brief outline of the background to my research topic. For more detail, please refer to my Stage 1 Proposal[Dow18] and Progression report[Dow19].

I am exploring whether the discipline of purely-functional programming offers any advantages for the design and implementation of distributed stream-processing systems. To answer this question, I am collaborating on a proof-of-concept stream processing-system — *StrIoT*[aut20c] — implemented using the Haskell purely-functional programming language.

Within *StrIoT*, a user writes a contiguous stream-processing program in terms of 8 provided functional operators[aut20a]. *StrIoT* rewrites the program via the application of logical rewrite rules before partitioning it into sub-programs which can be distributed and executed on separate nodes (e.g. cloud instances, gateways or edge devices such as Raspberry Pis). The *StrIoT* runtime takes care of connecting the distributed nodes together over TCP/IP.

My area of focus has been in the derivation of rewrite rules and implementation of the partitioning and logical optimiser.

## 2 Work completed

### 2.1 Encoding rewrite rules

Subsequent to the work to derive stream-processing rewriting rules described in my Progression report[Dow19], I have designed a scheme for encoding rewrite rules as pure Haskell functions and completed the implementation of all of these rules within *StrIoT*.

#### 2.1.1 Categories of Stream Processing Optimisations

Some of these rules were self-evidently useful, directly implementing optimisations from two of the topology-altering categories identified in [Hir+14]: *Operator Re-ordering* and *Fusion*. The remaining rules, whilst not self-evidently useful, might improve a stream-processing program with respect to some external non-functional requirement, or simply

open up opportunities to apply further optimisations in series, e.g. by moving two fuseable operators adjacent to each other.

It's clear that the utility of a rewrite rule is dependent on both the context in which it is to be applied (the program to which it is to be applied, the nodes available for execution and the non-functional requirements for that program) and the approach used to apply them (e.g. whether several rules are applied in sequence).

### 2.1.2 Rules and Event re-ordering

During the original derivation work, I wrote QuickCheck[CH00] properties for each rule in order to gain further assurance that they were valid. By doing so I discovered that some promising rules did not strictly preserve the order of events compared to the original input program. I initially decided to exclude those rules for that reason.

Separately to this work, when considering the semantics of the `streamMerge` operator, I discovered that we could not guarantee event ordering after partitioning the program if the merge operation was at the ingress of a node, due to unpredictable event arrival times over TCP/IP.

Once it became clear that event ordering was unlikely to be preserved for any practical program, I decided to include the promising rules.

If event ordering is critical for a particular application, it is possible to handle event re-ordering within the stream-processing program. Further work could determine the extent of re-ordering that is required for each rewrite rule.

### 2.1.3 Encoding as Haskell functions

During the rule derivation process, I used a simple pseudo-code representation of rewrite rules, with a *pattern* on the left-hand side identifying two connected operators and the rewritten form on the right-hand side, e.g.:

```
streamExpand . streamExpand == streamExpand . streamMap concat
```

To support this style of encoding within *StrIoT*, we would need to formalise a grammar for the rules and write a corresponding parser. Instead, I designed an encoding directly into regular Haskell functions.

The left-hand side of a rule is represented using Haskell's *pattern-matching*. The pattern matches against the `StreamVertex` type defined in *StrIoT* and the `Connect` constructor provided by the graph library we use[Mok17].

The right-hand side is defined by the body of the function. Rewrite rules may need to affect changes beyond the sub-graph that matched the pattern, and so they compose and return a function that takes the full graph as its argument. The returned function is wrapped in a `Maybe` type to signal cases where a rewrite rule is not applicable (such as the case when the pattern does not match).

The earlier pseudo-code rule is encoded as the following Haskell function:

```
expandExpand :: RewriteRule
expandExpand (Connect (Vertex e@(StreamVertex _ Expand _ _ _))
                      (Vertex    (StreamVertex _ Expand _ _ _))) =

    let m = e { operator=Map, params=[| concat |] }
    in Just (replaceVertex e m)

expandExpand _ = Nothing
```

Due to the nature of the constructors provided by the graph library, it is not possible to represent the left-hand side of rewrite rules involving more than two nodes entirely with pattern-matching. Those rules require further checks in the function body. I have implemented several such rules, however the vast majority of the rules I have derived involve just two nodes.

## 2.2 Applying rewrite rules

The algorithm to apply rewrite rules is represented by an Inner and an Outer function[Dow20b].

The inner function attempts to apply the given rewrite rule to the stream graph by traversing the stream graph structure recursively. It stops at the first match and returns the result of applying the rule.

The outer function maps the inner function over the list of all rewrite rules to yield a list of rewritten graphs. It then recurses. The rules are not convergent and so termination is not guaranteed when applying multiple rules in succession. The outer function therefore implements a maximum limit to the number of successive rule applications.

Rewrite rule application is demonstrated by several examples distributed with *StrIoT*, including an example of filter fusion within a solution[aut20b] to the 2015 DEBS Grand Challenge[Var15].

## 2.3 Looking below the level of Operators

Since rewrite rules operate at the level of inter-connections between operators, they cannot match against structures within the parameters to an operator, such as within a filter predicate. This means the rules cannot achieve *operator separation*: for example decomposing a filter into a pair of filters, as in this rule:

```
streamFilter (\v -> q v && p v) == streamFilter p . streamFilter q
```

In order to unlock this category of optimisation, I investigated alternative ways of encoding stream-processing programs. Of particular promise was Template Haskell[SP02], a compile-time meta-programming system which provides the programmer with a mechanism to translate any Haskell expression into an instance of an Abstract Syntax Tree data structure. My hypothesis was that by obtaining the AST for the code representing an operator and its parameters, it would be possible to access a decomposed form of the parameters.

I determined that it was technically possible to achieve this but impractical. The AST that Template Haskell provides to you corresponds to the precise expression that was used in the source code. Logically equivalent expressions that differ syntactically yield different ASTs and so simple equality tests or pattern matching techniques cannot be relied upon. The level of abstraction of the AST is significantly lower than that of the existing rewrite rules and straddling that divide would be complex and error-prone.

If the expression being converted to an AST references names that are not defined within that expression (e.g. $p$ in $streamFilter\ p$) the resulting AST will lack those definitions. A rewrite rule that would match against the definition of such a name could not be applied in this circumstance.

It's possible that with further work this situation could be improved. *operator separation* might be achievable using a different approach than Template Haskell. I currently have no further plans to pursue this avenue of investigation.

### 2.4 Stream-program representation

Despite the negative result above, I managed to apply Template Haskell to improve the ease of use of the system. I simplified the construction of stream-processing programs by using a TH expression data-type for the representation of operator parameters. The user can now use first-class Haskell expressions when defining a stream-processing program instead of embedding code within strings. I also managed to simplify much of the implementation of rewrite rules. A full explanation of this work is available in a blog post[Dow20c].

### 2.5 Automatic Partitioning of the Stream Processing Graph

At the end of Stage 1, our proof-of-concept was capable of partitioning a stream-processing program into sub-programs in conjunction with a user provided manual partition map. It is desirable for the system to instead automatically partition the supplied program, freeing the user from having to compose a manual mapping.

I have implemented the first stage of an automatic partitioning scheme. *StrIoT* is now capable of generating every possible partitioning of the supplied program[Dow20a]. These are combined with all rewritten programs derived by the Logical Optimiser. However, it still remains the responsibility of the user to pick the preferred scheme and rewritten program from the generated list. For sufficiently complicated stream-processing programs, the list of combinations may be very large.

### 2.6 DEBS 2020 paper submission

Paul Watson (my supervisor), Adam Cattermole and I wrote and submitted a paper to the 14$^{th}$ ACM International Conference on Distributed and Event-based Systems [Var20].

The paper is an overview of the *StrIoT* system and work completed to-date. My main contribution is a section on logical optimisation and deployment.

During the process of writing the paper we decided that we would be unable to complete the work needed to implement a Cost Model into our proof-of-concept system in the time available to us before the submission deadline. For this reason we instead focussed on ensuring the other components were completed and functioning to a high standard.

## 3 Work remaining

### 3.1 Cost Model

In order to determine whether a rewritten stream-processing program is preferred, according to the specified criteria, we need to be able to evaluate several plans and compare them.

I began work on implementing a cost model based on Jackson Networks[Jac57] but paused this work when we approached the deadline for DEBS 2020. I plan to complete this work to establish whether a Jackson Network is sufficiently rich to model a stream-processing system adequately for some sample non-functional requirements. Two promising outcomes of a Jackson Network-based model might be

- detecting breach of utility in a potential plan, i.e., where a plan results in a node receiving more work than it is capable of, which could either aid in ruling out plans which over-utilise lightweight processing nodes such as Edge nodes, or identify opportunities for horizontal scaling in e.g. Cloud nodes.

- using bandwidth information from a Jackson Model to minimize bandwidth over a network link or meet bandwidth constraints.

### 3.1.1 Phase re-ordering

Once we have a Cost Model in *StrIoT* we need to apply it to every combination of potential partition scheme and rewritten program, in order to select the best combination.

*StrIoT*'s current workflow has the following phases, some of which are manual steps for the user to execute, and others are executed automatically:

1. The user writes a stream-processing program
2. *StrIoT* generates all possible rewritten programs, subject to the available rewrite rules and recursion limit
3. for each rewritten program, *StrIoT* calculates all possible partitionings
4. The user selects their preferred pair of optimised program and partition mapping
5. *StrIoT* partitions the selected program according to the selected partitioning
6. The user deploys the resulting sub-programs

The above workflow requires the user to manually evaluate the rewritten programs and possible partitionings of the programs to pick the best. There are potentially a very large number of options. It would be much better if *StrIoT* could perform step 4 automatically, using a cost model as described in the previous section.

By doing so, the workflow would be entirely automated from the submission of the program up to the deployment of the optimised and partitioned sub-programs.

In order to do so, the user would need to supply *StrIoT* with a *catalogue* describing the available nodes for deployment (e.g. cloud instances or edge devices) and the *non-functional requirements* against which plans are evaluated.

## 3.2 Paper on derivation of rewrite rules

I plan to write a separate paper based on the work described in Section "Derivation of stream rewriting rules" of [Dow19] and the encoding and categorisation described in Section 2.1.

The Cost Model work described above may impact this paper, in particular looking at how Cost Model information about a stream-processing program, such as the selectivity of each operator, might be transformed by stream rewriting.

I need to identify appropriate conferences and journals to which this paper could be submitted in order to develop a plan for submission.

### 3.2.1 QuickSpec

QuickSpec[Sma+17] is a system for discovering rules and laws from a set of pure functions. It would be interesting to see whether QuickSpec or a similar tool could be used to derive further logical rewrite rules that I did not manually derive using a pairwise comparison of operators.

### 3.3 DEBS 2020 submission feedback

The paper described in 2.6 was ultimately rejected but we received very constructive feedback from the reviewers, which broadly fell into the following categories.

The evaluation of our system was very limited. We will address this by completing the implementation of a cost model, as described in Section 3.1.

Our comparison to other systems was insufficient. I will perform a more thorough review of existing systems and their capabilities.

We made reference to IoT and Edge computing at the beginning of the paper but did not return to it later on. I am trying to construct a new working example based on an IoT problem that can be used throughout the paper.

Once these shortcomings are addressed we may then attempt to update the paper and submit it to another conference. Alternatively we may restructure the material into a longer, more detailed paper, for submission to a journal.

## 4 Conclusion

I am currently on-track according to my plan described in [Dow19].

I have a plan of action for taking forward the paper that I co-authored this stage and addressing the reviewer feedback we received. I have an outline plan for writing a second paper (Section 3.2).

The main focus of my work going forward will be the Cost Model described in Section 3.1, which will address the most serious concerns raised in the paper feedback and also open up the opportunity to pursue research questions such as the impact of graph rewriting on Jackson network parameters outlined in Section 3.2.

## References


[aut20a] *StrIoT* authors. *StrIoT — Stream Operators*. 2020. URL: https://github.com/striot/striot/blob/7acb1cb057ba8f3569b9ab42030bb5b43acca49a/docs/Operators.md.

[aut20b] *StrIoT* authors. *StrIoT — Taxi example*. 2020. URL: https://github.com/striot/striot/tree/81e5e369f51eff098970ddbe47d8bc1c4a8bd68a/examples/taxi.

[aut20c] *StrIoT* authors. *StrIoT*. 2020. URL: https://github.com/striot/striot.

[CH00] 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. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266.

[Dow18] Jonathan Dowland. *PhD Proposal. Declarative, Distributed Functional Stream Processing*. 2018. URL: https://jmtd.net/log/phd_proposal/PhD_Proposal_-_Jon_Dowland.pdf.

[Dow19] Jonathan Dowland. *Declarative Distributed Stream Processing. PhD Stage 1 Progression Report*. 2019. URL: https://jmtd.net/log/phd/dowland_phd_stage1_progression_report.pdf.

[Dow20a] Jonathan Dowland. *Implement allPartitionings — StrIoT Pull Request*. 2020. URL: https://github.com/striot/striot/pull/98.

[Dow20b] Jonathan Dowland. *StrIoT source file* ***src/Striot/LogicalOptimiser.hs***. 2020. URL: https://github.com/striot/striot/pull/71/files#diff-cbd139ce86f89b1fc1c054e10f002f20R16.

[Dow20c] Jonathan Dowland. *Template Haskell and Stream-processing programs*. 2020. URL: https://jmtd.net/log/template_haskell/streamgraph/.

[Hir+14] Martin Hirzel et al. "A Catalog of Stream Processing Optimizations". In: *ACM Computing Surveys (CSUR)* 46 (Mar. 2014). DOI: 10.1145/2528412.

[Jac57] James R. Jackson. "Networks of Waiting Lines". In: *Oper. Res.* 5.4 (Aug. 1957), pp. 518–521. ISSN: 0030-364X. DOI: 10.1287/opre.5.4.518. URL: https://doi.org/10.1287/opre.5.4.518.

[Mok17] Andrey Mokhov. "Algebraic Graphs with Class (Functional Pearl)". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: ACM, 2017, pp. 2–13. ISBN: 978-1-4503-5182-9. DOI: 10.1145/3122955.3122956. URL: http://doi.acm.org/10.1145/3122955.3122956.

[Sma+17] Nicholas Smallbone et al. "Quick specifications for the busy programmer". In: *Journal of Functional Programming* 27 (2017), e18. DOI: 10.1017/S0956796817000090.

[SP02] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: Oct. 2002, pp. 1–16. URL: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[Var15] Various. *ACM DEBS 2015 Grand Challenge. New York Taxi Trips*. 2015. URL: http://www.debs2015.org/call-grand-challenge.html.

[Var20] Various. *14th ACM International Conference on Distributed and Event-based Systems*. 2020. URL: https://2020.debs.org/.