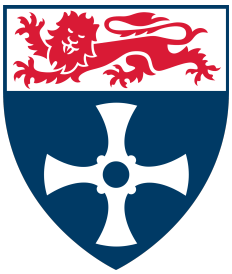
Purely-Functional Stream Processing



Jonathan Dowland

School of Computing

Newcastle University

A thesis submitted for the degree of

Doctor of Philosophy

April 2025

To all artists, musicians and creatives:
Thank you for making the world a warm, colourful, fun place.

Abstract

Extracting value from streams of data generated by sensors and software is key to the success for many important problem domains including the *Internet of Things* (IoT).

However there are many non-functional challenges to be overcome in achieving this, including very high data rates, a deployment environment featuring nodes with differing capabilities and limitations, energy and bandwidth constraints, performance requirements, and security guarantees.

Most modern stream-processing systems leave addressing these challenges for the application programmer to solve by making manual adjustments to their program. This entanglement of the functional and non-functional aspects of stream processing increases the risk of mistakes and the potential cost of future maintenance.

We describe an alternative approach — a declarative architecture ("*StrIoT*") based on purely-functional programming. The application programmer provides a program encoding the functional requirements, a separate description of the operating environment, and the applicable non-functional requirements. *StrIoT* then derives functionally-equivalent program variants, generates corresponding deployment plans and automatically deploys the best-cost plan.

In order to explore the viability of purely-functional programming for this problem domain, we designed and built a proof-of-concept, end-to-end implementation of this architecture using the purely-functional language Haskell, Linux containers and orchestration technology.

This thesis focusses on two key components: the Optimiser and Evaluator.

The Optimiser leverages purely-functional semantics to apply term rewriting and derive functionally-equivalent variants of the user's Haskell program.

The Evaluator filters and selects deployments plans using modelling techniques including queueing theory, in order to select the plan which will perform best for two non-functional requirements: bandwidth and cost.

To evaluate these components and approaches we implemented several solutions to real-world stream-processing problems. In some cases, *StrIoT* can generate programs which meet applicable non-functional requirements even when the user-supplied program does not.

Declaration

I declare that this thesis is my own work except where specific reference is made to the work of others. No part of this thesis has been previously submitted for a degree or any other qualification at Newcastle University or any other institution.

Jonathan Dowland

April 2025

Acknowledgements

I wish to thank Professors Paul Watson and Mark Little, without whose support this work would not have been possible.

My eternal gratitude to my wife Sarah and children Holly and Beatrice for bearing the burden alongside me.

Thank you to my parents for giving me the foundations in life to build on.

Thanks to Professor Emeritus Isi Mitrani and Doctor Paul Ezhilchelvan for their thorough advice (patiently delivered) on queueing theory.

Thank you Doctor Andrey Mohkov for answering many questions both on algebraic graphs and more generally on the business of Science.

Thanks to Doctors Peter Michalák, Adam Cattermole and Thomas Cooper: for a brief time we journeyed together, but you patiently helped me across the finish line.

Thank you Doctor Sophie Watson for walking me through the Undergraduate-level Statistics I'd hitherto avoided.

Finally, thanks to Professor Chloë Starr, whose encouragement (at a dinner conversation likely forgotten) set me on this path.

Table of Contents

List of Figures

List of Tables

List of Publications

Workshop papers

[1] A. Cattermole, J. Dowland, and P. Watson, "Run-time adaptation of stream processing spanning the cloud and the edge," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21, Leicester, United Kingdom: Association for Computing Machinery, 2021, ISBN: 9781450391634. DOI: 10.1145/3492323.3495627. [Online]. Available: https://eprints.ncl.ac.uk/280160.

[2] J. Dowland, P. Watson, and A. Cattermole, "Logical optimisation and cost modelling of stream-processing programs written in a purely-functional framework," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, ser. UCC '22, Vancouver, Washington, USA: IEEE, Dec. 2022. DOI: 10.1109/ucc56403.2022.00048. [Online]. Available: https://eprints.ncl.ac.uk/289943.

Chapter 1. Introduction

1.1 Motivation

Distributed stream-processing is applied to modern problems from a wide range of domains, including healthcare[58], environmental monitoring, manufacturing, social media[47], and particle physics. These problems require real-time responses to both large volumes of data, as well as data arriving at high velocity: the ALICE detector at the Large Hadron Collider in CERN generates more than $3.5\,\mathrm{TB}$ of data every second[2].

Designing and deploying such systems is complex and requires expertise across a heterogeneous range of technologies, including embedded programming, the "Internet of Things" (IoT), sensors and actuators, low-power signalling, data science, mesh networking and cloud computing. It can be very difficult to hire or train programmers to the required level of expertise across the full breadth of these technologies.

Calibrating such systems to meet *non-functional requirements* (such as energy, networking, security or performance) largely remains the responsibility of the application programmer, by adjusting and conflating the data-processing logic of their program with deployment considerations such as operator placement. This can obscure clarity, increase the risk of programming mistakes and incur a higher maintenance cost.

1.1.1 Declarative Stream-Processing

Michalák et al proposed a declarative model for application development [57]. A high-level visualisation of this architecture is provided in Figure 1.1.

To address the difficulty of managing the disparate technologies required for current stream-processing solutions, the model exposes a single, unified programming environment for the application developer to gain familiarity. Alongside the program describing the application logic, the developer supplies the system with a separate specification of the relevant non-functional requirements: for example, cost, energy, or performance; and a catalogue describing the deployment environment. By defining the non-functional requirements independently from the application logic, the clarity of the logic need not be compromised, reducing the risk of programming mistakes and the cost of future maintenance.

The system then automatically optimises the program and tailors a deployment plan to maximise performance against the specified non-functional requirements.

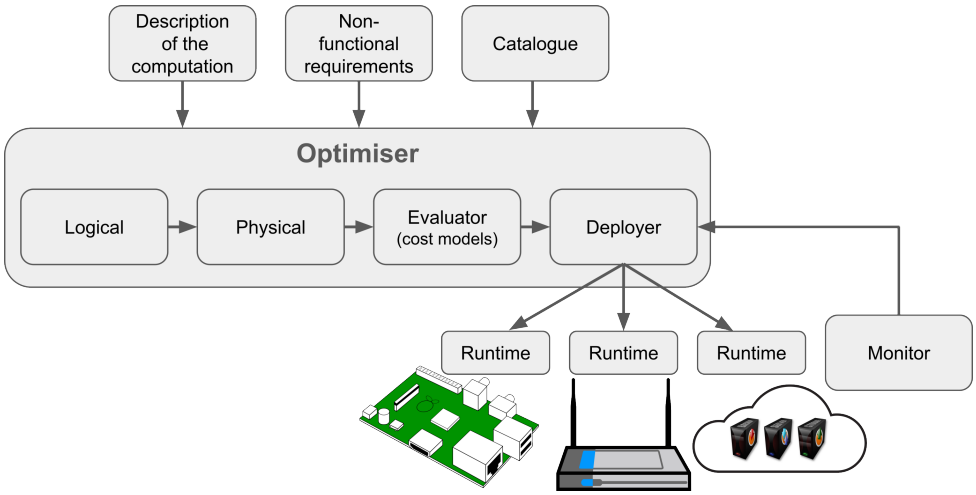


Figure 1.1: An overview of the declarative stream-processing architecture

1.2 Functional Programming

Functional Programming is a methodology where the principal abstraction is the function. Functions can be passed as arguments to other functions and returned as values. Programs are built declaratively from expressions, which are produced by applying and composing functions together.

Purely-functional programming emphasises functions without side-effects: functions which produce no result beyond the return value. Pure functions always evaluate to the same value for the same inputs. This means program evaluation can proceed with many different evaluation strategies, including lazy evaluation, where a term is only evaluated when its value is required. Pure functions are referentially transparent: any pure expression can be substituted for another which is semantically equivalent, i.e., evaluates to the same value for the same inputs.

Purely-functional programming is described in more detail in Section 2.2.

1.3 Functional Stream-Processing

Stream-processing systems originated in the relational community and began as extensions to batch-processing. The adoption of functional concepts by MapReduce [19] served as an inflection point in their evolution, enabling the separation of data-processing logic from deployment considerations and horizontal scaling across distributed, commodity hardware. Functional programming has been successfully used to build large, complex systems [16].

The declarative model was initially explored via a relational implementation [58]. An initial prototype of a functional implementation (described in Section 2.5) was produced in 2017. This prototype lacked the Logical Optimiser and Evaluator components of the declarative architecture, and the Deployer was not fully functional.

1.4 Research Questions and Objectives

By designing and implementing the missing components from the functional prototype and producing a functionally complete end-to-end implementation, we aim to address the following research questions:

1. Is the declarative model viable? By building a second, independent implementation of the model, we will provide further evidence as to its viability.
2. Can purely-functional programming be used to build an end-to-end system this complex?
3. By comparison with the earlier relational implementation: Does purely-functional programming offer any specific advantages or disadvantages in the design or implementation of specific components from the declarative model?

1.5 Thesis Structure

In Chapter 2, we survey the background and key concepts in stream-processing and functional programming that inform this research. We describe in detail the proof-of-concept functional prototype that forms the foundation of the work of this thesis.

Chapter 3 describes the design and implementation of the Deployer, the approach used for partitioning and the choice of data-types for representing stream-processing programs.

Chapter 4 describes the application of term rewriting for transforming programs, categorising transformations according to established categories of stream-processing optimisation, considerations of program semantics and transformation correctness, machine-assisted rule derivation and implementation of a Logical Optimiser leveraging rewrite rules.

In Chapter 5 we explore approaches for modelling cost in a stream-processing deployment in order to evaluate and rank rewritten programs, our chosen approach and the implementation of the Evaluator.

In Chapter 6 we evaluate the performance of our system by re-implementing solutions to real-world stream-processing problems with *StrIoT* and comparing the cost of the original programs against variants derived by the Logical Optimiser.

Chapter 7 discusses the results of this doctoral research and looks at further areas for study.

We have published the system we built to support this work — *StrIoT* — as open-source software. We include excerpts of the source code when doing so improves clarity, but we do not reproduce the full text within this thesis. We reference specific lines from files in the *StrIoT* source code, such as lines 13 from the `README`, as follows: [79, `README#13`] .

Information on obtaining the full source code is provided in Section A.1.

Chapter 2. Background and Related Work

2.1 Stream Processing

Stream processing originally emerged from the database community as an alternative approach for reasoning over data-sets which were either too large to be practically stored in traditional database systems, or not available all at once. Stream processing has been applied to problems in domains including environmental monitoring, civil engineering, sensors networks, meteorology and healthcare [20], [58].

In the Big Data era, Stream processing has evolved from its relational origins, influenced by developments in Cloud and Edge Computing, to address increasing demands for low-latency responses to both large volumes of data as well as data arriving at high velocity [14].

We now provide an overview of the history of streams and distributed stream-processing. For a more thorough analysis, we recommend [28].

2.1.1 Stream definition

A Stream is a data-set which is not available all-at-once. The full data-set may not yet exist, may be too large to be practically stored or retrieved, or may be infinite. Sub-sets of data may arrive at the processing system at varying times or rates, and potentially out-of-order [30].

Since the full data-set is never available, strategies to tackle problems involving streamed data must use incremental methods. Information from past data that is required for future calculations must be retained until those calculations are complete.

Consider the mean-average of a set of numbers, normally calculated by dividing their sum by the size of the set. In a streaming scenario, the set size could be unknown (in the case of finite streams) or infinite. It would be impractical to store every number received in the stream. If the stream is finite, it may be valuable to output an intermediate answer, calculated on the available data, prior to the ultimate answer. In the infinite case there will be no ultimate answer.

An incremental approach to this problem would be to continually emit the average of the data received to-date. The system would need to calculate and store a pair of numbers: the sum and count of numbers received. The storage requirement for the pair is constant. These values should be continually updated as new data is received. The running average can be quickly and continually calculated from the current pair

in constant time.

2.1.2 Origins of Stream-Processing

Stream processing originated as an extension of techniques from the database community. Relational systems were extended to support the notion of operating on sub-sets, or "windows", of data. The user typically composed their queries using domain-specific languages (such as dialects of Structured Query Language) and in terms of pre-defined operations, e.g. selects and joins. Data was organised according to a defined schema [28].

Scaling the performance or capacity of these systems was usually achieved "vertically", with faster processors, larger capacity storage and memory, and faster network equipment. Fault tolerance was achieved via a "Highly Available" approach, using state replication, hot spares and failover. Systems typically prioritised service availability over strict accuracy, using techniques such as load shedding: temporarily dropping some input or skipping some processing to to manage temporary overload situations at the cost of result accuracy [28].

2.1.3 MapReduce

An inflection point occured in the evolution of stream-processing systems in 2004 with MapReduce [19], a batch-processing system inspired by concepts from Functional Programming. MapReduce abstracts away the complexity of both load scaling and fault tolerance from the application programmer, providing a simplified interface of only two parameters: what transformation to apply to each input datum (map) and how to collate the results (reduce).

By partitioning data and the work to perform and distributing them to multiple independent machines, MapReduce could be scaled up horizontally across conventional, commodity hardware, as opposed to vertically on expensive, specialised equipment.

MapReduce itself wasn't made available outside Google, but it inspired the design of subsequent open batch systems including Hadoop [17] and Spark [82], as well as the design of subsequent stream-processing systems [7], [49], [61], [73].

2.1.4 Second Generation Stream-Processing

Modern stream-processing systems have been heavily influenced by MapReduce and cloud computing. In contrast to the first generation's "data first, then query" approach, the second generation of stream-processing systems prioritise the data transformation, taking a "data-in-motion" perspective [37].

The user first describes the data transformation to take place in terms of a "dataflow" graph of operators. High-level operations with a user-defined payload (e.g. map, filter, reduce) are prevalent, rather than the traditional relational-inspired operators (select, join, etc.)

In contrast to the schema-led approach inherited from their roots in relational databases, with second-generation systems data is frequently either unstructured or the responsibility for applying and interpreting structure is left to the programmer, for example, to manually serialise and de-serialise structure from strings, and manually handle any format errors.

Rather than relying on acquiring more powerful processors, larger storage, or faster network equipment, scaling is achieved horizontally, utilising commodity hardware or cloud computing resources. This approach to scaling allows systems to be designed to adjust the quantity of deployed resources both up and down in response to workload demand.

On-demand resource provision has also influenced a move away from active replication and hot spares for fault tolerance, as in the first generation, towards passive replication and allocating additional resources (e.g. cloud instances) in response to failure events during operation.

Second-generation systems generally place a greater importance on correctness than the first generation and this is reflected in the techniques used for load management, such as back-pressure, where operators upstream of a load hotspot temporarily slow down their emission rate, as opposed to load shedding.

2.1.5 Hybrid architectures

The batch-oriented nature of both the traditional relational systems as well as the newer systems inspired by MapReduce mean they suffer from latency: no answer can be calculated until sufficient data has been received to fill the current batch. Some systems, such as Spark Streaming [81], attempt to address this by reducing the batch size. This reduces the latency but fundamentally cannot eliminate it. For problems which require more prompt answers to high-velocity data, a different architectural approach was needed [14].

However, the aforementioned batch-processing systems are well established and understood. In contrast, the newly emerging distributed stream-processing systems were complex and fragile. A hybrid approach was popularised in 2011 [54] whereby batch and stream-processing systems were both run in parallel, with the streaming system limited to processing data that had arrived in the interval since the last batch was collected. This aimed to bound the risk associated with deploying the stream-processing system, and leverage the well-understood performance and operation of the batch-processing system [76].

A drawback of this hybrid approach was the need for end-users to implement their data-processing logic twice. Kreps [46] proposed instead improving stream-processing systems to address the reliability and complexity concerns and leveraging them for both stream and batch workloads.

2.2 Purely-Functional Programming

Functional programming (FP) is a software development paradigm where the principal 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 functional programming believe that many of its properties have advantages for the design and implementation of large and complex software systems [16].

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 of functional programming in which pure functions cannot perform *side-effects*, or, actions that cause a change of state. To illustrate, consider a function with signature `foo :: Integer`. Once evaluated, a pure function of this type will have a value of type `Integer` and evaluating the function will have no other effect. An "impure" function however could produce other effects: it could scan over your email, delete files from your computer, or send source code to your printer.

Pure functions can be easier to reason about than impure functions, both for humans and machines such as compilers. In the expression `f g h`, where the functions `g` and `h` are passed as arguments to the function `f`, the order of evaluation of `g` and `h` does not impact their value. These properties give a compiler flexibility to re-order many expressions for optimisation without altering the semantic behaviour of the program.

Pure functions are *referentially transparent*: an instance of an expression in a program can be safely substituted for any other which is semantically equivalent, i.e. evaluates to the same value for the same inputs. Referential transparency enables equational reasoning, a technique for transforming functions through a process of substitution by applying laws or rules [5]. We explore equational reasoning in Chapter 4.

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) [36]. 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.

2.2.1 Haskell

Haskell is a lazy, purely-functional programming language, originally designed by committee with an initial release in 1990 [40].

The Haskell language is formally specified in a series of reports, each of which

supersedes the older versions. The latest report [53] was published in 2010. In practice, many programs (including the work of this thesis) rely upon language extensions which are not formally described in the reports.

There have been many compilers and interpreters for Haskell over its lifetime. Today, the Glasgow Haskell Compiler (GHC) [31] (originally from 1989 [40]) is predominant. The vast majority of language extensions are developed for, and ultimately supported within, GHC.

There are a great many books and online resources for learning Haskell. For an especially clear work, focussed on problem solving with pure functions, the author recommends [5]. For a more pragmatic look at applying Haskell to modern case studies, see [62]. We provide a short summary of the Haskell syntax used in this thesis in Chapter B.

2.2.2 Functional Streams

The natural functional data-type for representing a sequence of items for processing is the *recursive list*, defined as either the empty list, or a "head" element pre-pended to another list (the "tail"). Originating in LISP [39], the recursive list is a core data-type in the majority of subsequent functional languages, including Haskell.

Listing 2.1 illustrates the outline of a data-processing pipeline using lists. A generator produces a long list of data which is consumed by a series of processing steps before being displayed or otherwise processed at the end.

```
[0..1_000_000_000] & map (*5) & map (-2) & mapM_ print
```

Listing 2.1: An example of a list-based data-processing pipeline

With applicative-order (eager) evaluation, performing successive data processing on lists is very inefficient in both space and time. Both the list structure and the data within are fully evaluated at each step in processing. [1]

Functional Streams were introduced to address this [51]. A functional stream is defined as either empty, or a head element pre-pended to a *promise*, which, when evaluated, yields the tail of the stream. The evaluation of the promise is delayed until the value is required, or avoided entirely when the value is not required. Using functional streams, it becomes possible to work with infinite sequences.

This type is analogous to but distinct from *list*, requiring an independent set of functions to manipulate them (`head`, `tail`, `length`, `cons`, etc.). This prevents the programmer from re-using existing libraries of routines that are specialised for lists.

The introduction of lazy evaluation [36] alters the semantics of lists (and other data structures) such that there is no longer a need for a distinct stream type.

Even with lazy evaluation, there remains a cost associated with deconstructing and reconstructing lists at each stage in processing. Approaches to solve this include algorithms to re-write programs to remove intermediate lists [33], [75]. These techniques

were adopted in GHC in the form of re-write rules [66].

A related performance issue remains: sometimes, deconstructing the entirety of a list is unavoidable. Consider the final step in Listing 2.1: the full list and all data within must be evaluated. This process involves allocating temporary objects which will eventually be removed by garbage collection. However, in the short term, they consume resource, and in some circumstances this can be significant. Attempting to execute the example in Listing 2.1 results in an Out Of Memory error on the author's contemporary workstation.

Haskell Stream-Processing

There are a number of stream-processing systems within the Haskell community. Those surveyed for this work are Conduit [71], Pipes [34], io-streams [13], Streaming [72] and Streamly [50].

In order to avoid the performance problems described above, they all provide new data-types for modelling streams, distinct from lists, as well as a library of functions to manipulate them, in most cases closely following established functions and patterns from the Haskell Prelude.

Listing 2.2 is a re-implementation of the example in Listing 2.1 using functions from Streaming. The revised routine requires a constant amount of memory to execute and, in contrast to the original example, completes successfully on the author's contemporary machine.

```
S.each [0..1_000_000_000] & S.map (*5) & S.map (-2) & S.print
```

Listing 2.2: Example from Listing 2.1 re-implemented with Streaming [72]

These systems all provide support for *effectful streams*: the ability to interleave side-effects within a stream-processing computation.

They are generally written from the perspective of operation on a single computer. None are designed for distributed operation: although two provide rudimentary support for connecting streams over TCP networks (Conduit and Pipes), they offer limited support for serialising data structures over the connection, and the programmer is burdened with determining which streaming operations should be executed on what nodes.

2.3 Containers

An OCI container image is an application together with all its software dependencies, packaged in a standardised format, such that a container runtime can start the application without requiring further software. A running container will typically be isolated from the host system, with a private filesystem, process table, virtual network devices and user metadata.

Devised and popularised by dotCloud, inc (later Docker, Inc) in 2013 [41], Linux Containers were subsequently standardised by the Open Container Initiative (OCI) in 2017 [64].

OCI containers are a very popular method for building and distributing software. In contrast to Virtual Machines, they are more lightweight, running as a regular (albeit isolated) process on top of a conventional operating system, not requiring software-virtualised hardware or individual operating system kernel.

By bundling the run-time dependencies of an application and isolating the running application from the host machine, containers enable developers to simplify deployment operations by eliminating unintended discrepancies between the development and production environments. This facet also greatly improves the reproducibility of scientific workflows [67].

The runtime isolation of containers from both the host system and other containers has benefits for security, in particular for multi-tenant environments: A compromised application is prevented from accessing the data belonging to other applications.

Prior to the popularisation of containers, Virtual Machines were a popular choice for isolating application workloads from each other. In comparison to Virtual Machines, the runtime overhead of containers is very low. This greatly improves the utilisation of host systems.

2.3.1 Orchestration

Modern applications can require multiple independent containers, for example separate back-end storage and front-end business logic, or applications de-composed into separate micro-services. Container orchestration is responsible for managing and coordinating the separate constituent containers for an application deployment.

Docker Compose [21] is an example of a lightweight container orchestrator. Configured with a single file, Docker Compose allows the user to describe the containers for a deployment alongside any shared resources, such as persistent data volumes, private virtual networks or environment variable definitions. The Compose tooling will then ensure the resources and containers are started and stopped in the correct order.

Docker Compose is typically used for deploying upon a single host and is not recommended for production deployments across multiple hosts.

Kubernetes [48] is an open-source, enterprise-scale container orchestration system which manages deploying containers across a distributed environment.

Cattermole subsequently extended the deployment element of *StrIoT* with support for Kubernetes [8].

2.4 Declarative Stream-Processing

The design of modern data-processing applications often conforms to a three-tier model: sensors and low-powered devices at the Edge; elastically-scaleable computational resources in private or public clouds and "gateway" devices serving to connect and bridge the two.

Developing applications in this model can be very challenging due to the wide range of different technologies, each of which often require specific tools and skills to operate. it can be very hard to hire or train application programmers to a high level of expertise across the full heterogeneous suite of technologies that may be required.

To address these problems, a declarative architectural model for application development was proposed in [57]. A high-level visualisation of this architecture is provided in Figure 1.1, Chapter 1.

The model exposes a single programming environment for an application developer to gain familiarity.

Alongside the program describing the application logic, the developer supplies the system with a specification of the *non-functional requirements* the deployment should be optimised for: for example, cost, energy, or performance, and a *catalogue*, describing the deployment environment.

The system then automatically optimises the program and tailors a deployment plan to maximise performance against the specified non-functional requirements.

In order to measure the performance of the deployment and respond to changing environmental conditions, the model includes a Monitor to collect run-time information. By feeding this back to the Optimiser, the model can support run-time adaptation.

2.4.1 Relational Implementation

This design was first explored by Michalák et al [56], [57], [58] who designed and developed a relational-based system which leveraged Event-Processing Language (EPL) , a domain-specific relational language for interacting with stream data.

Michalák demonstrated the validity of the declarative architectural model with two example programs, greatly improved by the system's Optimiser: one program in terms of power consumption and another in terms of bandwidth. Michalák noted some limitations of the approach [56]. in particular as EPL is a domain-specific, rather than general-purpose programming language, it was not expressive enough for some of the required computations, which had to be addressed using user-defined functions (UDFs). The Optimiser was unable to reason about the performance of UDFs which placed limits on what it could achieve.

2.5 Functional Prototype

In contrast to the relational approach, Watson and Woodman began to explore a functional design in 2017 producing an initial prototype dubbed *StrIoT* – **Str**eam processing for the Internet of Things (**IoT**) [78].

The prototype implemented the *description of the computation*, *physical optimisation* and *run-time* components from Figure 1.1. It did not include the Logical Optimiser, Evaluator or Run-time Monitor.

StrIoT is available as open-source software [79]. The remainder of this chapter describes the design as of the 2017 prototype. *StrIoT* has subsequently been extended by Cattermole [8], [9] and the work described in this thesis.

2.5.1 Stream and Event types

We model a stream in *Haskell* as a (possibly infinite) list of events:

```
data Event a = Event { time  :: Maybe Timestamp
                     , value :: Maybe a }

type Stream a    = [Event a]
```

`Event` has been designed to be as general as possible: it can hold data of any type (e.g. integers, strings, tuples, lists, trees, graphs and even functions). In addition to the data-type, an `Event` can also optionally hold a timestamp. [1]

From the perspective of the application programmer, the `Stream` type is opaque: the user writes their program in terms of functions which operate on the stream's payload and are not burdened with manipulating the stream type itself.

2.5.2 StrIoT Operators

Based on an analysis of the literature on both stream-processing and Complex Event Processing [18] and by experimenting with the implementation of a range of applications, Watson and Woodman provided application developers with eight pure stream-processing operators to compose their applications. The operators are listed in Table 2.1.

The approach to designing the operators was to provide a balance between defining a small core set with the simple, clear semantics needed for analysis by a future Logical Optimiser, whilst providing a sufficiently expressive programming environment for the application writer (by directly supporting the main operations found in stream-processing applications).

We are confident that any stream-processing problem that can be described in a conventional, non-pure stream-processing system can also be implemented in *StrIoT*. A

[1] The `Event` type presented here was modified from the 2017 prototype by Lukyanov, Watson and Cattermole in 2018, in order to avoid partial functions [77].

Category	Function
Map	`streamMap`
	`streamScan`
Filter	`streamFilter`
	`streamFilterAcc`
Aggregate	`streamWindow`
	`streamExpand`
Combine	`streamMerge`
	`streamJoin`

Table 2.1: *StrIoT* Stream operators

range of example programs are provided within the *StrIoT* source repository, including a solution [79, `examples/taxi`] to the DEBS 2015 Grand Challenge [43].

We now give an overview of the stream-processing operators, and demonstrate their use in an example program which we build up, one operator at a time. This example is based on a real-world medical problem originally addressed with the relational system described in Section 2.4.1.

The program converts a stream of sensor data from a smart watch being worn by a patient into an estimate as to the patient's relative activity level. The relational system went on to feed this forward into a statistical model running in the cloud to predict the risk of hyperglycaemia and alert the patient accordingly.

A detailed explanation of the wearable example is in [58]. We describe a more thorough re-implementation in Section 6.2.2.

We will connect the operators in our example with the Haskell function `&`: the order of the stream can be read top-to-bottom. (See Chapter B for a summary of relevant Haskell syntax).

We begin with a source function which produces the data to be streamed:

```
getWearableData
```

Filtering

Basic stateless filtering is achieved with `streamFilter`. The user provides a predicate which operates independently on values from the input stream and returns a boolean to signal whether the event should be emitted on the output stream.

```
streamFilter :: (a -> Bool) -> Stream a -> Stream a
```

For example, if a wearable device provided a set of sensor readings, we could filter the events to only those where a desired property held with a predicate function `vibrationModuleActive` provided by the user:

```
getWearableData
  & streamFilter vibrationModuleActive
```

Mapping

The function `streamMap` is used to transform the values in a stream. The programmer supplies a transformation function which is applied to each event in the input stream.

```
streamMap :: (a -> b) -> Stream a -> Stream b
```

Continuing the previous example. Consider that the sensor readings include movement data from motion sensors and the user wishes to calculate the magnitude of the vector of movement by applying a Euclidean function:

```
getWearableData
  & streamFilter vibrationModuleActive
  & streamMap euclideanDistance
```

`streamMap` is stateless: the user-supplied function operates on a single event at a time and does not have access to the values of earlier events in the stream. Where it is necessary to take into account previous events, for example to build stateful aggregations, we provide `streamScan`:

```
streamScan :: (b -> a -> b) -> b -> Stream a -> Stream b
```

Here, the user-supplied function takes a second parameter, the *accumulator*. When the function is invoked, the value returned by the previous invocation of `streamScan` is provided.

In addition to the user-supplied function, the user also provides an initial value for the accumulator.

Filtering with state

`streamFilter` is stateless. For situations where knowledge of prior filtering decisions is required, Watson and Woodman designed `streamFilterAcc`.

Much like `streamScan`, `streamFilterAcc` and the predicate function are extended to operate with an accumulator. Unlike `streamScan`, the accumulator value is not emitted on the output stream. The accumulator could, for example, be a list of previously seen values. In addition to the filter predicate, `streamFilterAcc` requires an accumulator update function to be supplied.

```
streamFilterAcc :: (b -> a -> b)     -- accumulator update function
                -> b                 -- initial accumulator
                -> (a -> b -> Bool) -- predicate function
                -> Stream a
                -> Stream a
```

Continuing the running example. Suppose the user wishes to filter out events which describe a movement which is below a threshold relative to the previous event.

```
getWearableData
  & streamFilter vibrationModuleActive
  & streamMap euclideanDistance
  & streamFilterAcc (\old new -> new) 0 thresholdCheck
```

In the above example, the accumulator update function simply returns the most recently seen value and the accumulator is initialised to `0`.

Aggregation

`streamWindow` collects together incoming events and batches the data from them into lists to be emitted. `streamWindow` must be provided with a window-maker function which implements the criteria for which events to group together.

```
type WindowMaker a = Stream a -> [Stream a]
streamWindow :: WindowMaker a -> Stream a -> Stream [a]
```

The user can write their own window-maker or use one of a set of common ones provided by *StrIoT*. These are: `sliding`, for overlapping windows of fixed length; `chop`, for fixed-length, non-overlapping windows; and two variants which operate based on time intervals: `slidingTime` and `chopTime`.

In our running example, the user's final goal is to collect together batches of events that occur within a specified time interval. They can use the built-in window-maker function `chopTime`:

```
getWearableData
  & streamFilter vibrationModuleActive
  & streamMap euclideanDistance
  & streamFilterAcc (\old new -> new) 0 thresholdCheck
  & streamWindow (chopTime 120)
```

`streamExpand`, the dual of `streamWindow`, receives events which contain lists of some type and unpacks the list, emitting each individual item as separate `Events`.

```
streamExpand :: Stream [a] -> Stream a
```

We illustrate `streamExpand` with a new example. Consider a function that takes in a stream of short text messages from a social media site and generates a stream of the *hashtags* (specially delimited words) found within those messages.

```
getHashtags :: String -> [String]
```

This function could be applied to a stream of type `Stream String` by passing it as the parameter to `streamMap`, resulting in a stream of type `Stream [String]` (a stream of lists of strings). Applying `streamExpand` then expands the list from each `Event` and emits the individual hashtags in new events:

```
streamExpand . streamMap getHashtags
```

The resulting type is `Stream String`.

Combining Streams

Many stream-processing programs receive input from multiple sources and there is often a requirement to combine them together. `streamMerge` takes a list of stream inputs (of the same type) and interleaves their events into a single output stream.

```
streamMerge :: [Stream a] -> Stream a
```

Consider the problem of combining readings from multiple temperature sensors deployed in the field. If there are three such sensors, then we could combine them as follows:

```
streamMerge [ temperatureSensor1
            , temperatureSensor2
            , temperatureSensor3 ]
```

The final operator, `streamJoin`, is used for combining exactly two inputs of possibly different types. `streamJoin` pairs events from each input stream together and emits them as tuples.

```
streamJoin :: Stream a -> Stream b -> Stream (a,b)
```

Consider the case that we wish to capture the readings from a temperature sensor and pair them with readings from a nearby humidity sensor:

```
streamJoin temperatureSensor1 humiditySensor1
```

2.5.3 Deployment and Runtime

We now describe the run-time and Deployer components of the prototype, including code generation and its shortcomings.

Runtime

The *StrIoT* prototype includes a basic run-time component with support for transmitting and receiving `Streams` between nodes via TCP/IP.

The run-time provides a set of functions to enable nodes to participate within a number of fixed topologies.

For example, the function `nodeSink` (type signature in Listing 2.3) enables a node to operate at the end of a stream-processing program. `nodeSink` decodes incoming TCP/IP data to the type `Stream a` and applies it to the first argument, a pure stream-processing function, to yield a type `Stream b`. It then applies this to to the second argument to yield an I/O action.

```
nodeSink :: (Stream a -> Stream b) -> (Stream b -> IO ()) -> IO ()
```

Listing 2.3: `nodeSink` type signature

Similar functions are provided for nodes operating at the start of a stream and in the middle of processing (`nodeSource`, `nodeLink`).

These functions expected to operate on a single incoming (or outgoing) stream. Basic multivalent topologies were supported by `nodeLink2` and `nodeSink2`, which instead accepted two incoming streams. Rather than attempt to de-multiplex stream data from separate sources, each incoming stream was required to connect to a distinct TCP/IP port. There was no support for receiving more than two incoming streams.

This initial run-time has been subsequently improved and extended by [8], [9].

Example Programs

The prototype included configuration to build an OCI container image consisting of the *StrIoT* runtime, GHC and supporting libraries. This was used to support several provided example programs.

Each example consisted of a stream-processing program that had been pre-partitioned into separate sub-programs, packaged as containers building upon the *StrIoT* base image.

Each example also included a configuration file for Docker compose (See Section 2.3.1) describing how the separate containers should be inter-connected and allowing for the examples to be run quickly and easily.

Code Generation

The prototype included some initial work on code generation. A set of types were defined to describe stream-processing programs, entirely distinct from the user-oriented functions described above in Section 2.5.2. These types were designed to make the manipulation of a stream-processing program as data more straightforward.

A series of functions were provided to divide instances of these data-types into sub-programs and generate code for them, suitable to be compiled by GHC independently from each other. The generated code used the stream-processing functions, `Event` and `Stream` types described in Section 2.5.1, as well as the runtime routines described in Section 2.5.3.

Runtime constraints

The prototype had limitations which restrict the topologies that can be used for composing stream-processing programs.

The set of runtime routines that are used as sub-program entry-points (described above) was limited to three classes of deployment node:

1. "sources", which do not receive incoming streams, perform IO to generate stream events and emit exactly one outgoing stream.
2. "sinks", which receive one or two incoming streams, perform IO to output the result of stream processing and emit no further events.
3. "links" which receive one or two incoming streams, emit exactly one outgoing stream and cannot perform any IO.

Any topology that does not match these classes, for example a node receiving more than two incoming streams, was not supported.

The code generator does not generate correct code for the `streamMerge` operator: it incorrectly applies two stream arguments, where `streamMerge` expects to receive one argument (a list of streams).

However, the code generator detects topologies where a `streamMerge` operator occurs as the first operator within a deployment node. In this scenario, the work of merging the streams is actually performed by the run-time as part of de-serialising events arriving over TCP/IP. The code generator replaces the superfluous `streamMerge` operator with a no-op (`streamFilter true`). As a result, the bug in code generation for `streamMerge` is not triggered and these deployment plans are supported.

2.5.4 Missing Elements

The *StrIoT* prototype [78] did not include the Logical Optimiser, Evaluator, Deployer or run-time Monitor components from Figure 1.1.

The Logical Optimiser and Evaluator components are critical for an implementation of the architecture to be fully functional. The initial focus of the work described in this thesis was therefore to design and implement these missing components. The Logical Optimiser is discussed in Chapter 4 and the Evaluator in Chapter 5.

Although we do not explore run-time adaptivity in this thesis, it has been independently explored by Cattermole [8], [9].

2.6 Summary

Whilst second-generation stream-processing systems have adopted some concepts from functional programming (most notably the pure, higher-order functions `map` and `reduce`, via MapReduce) and mostly support a general-purpose programming interface for end-users, these tend to be conventional imperative languages such as Java. Second-generation systems also tend to lack support for data-types in streams, leaving the responsibility of serialisation, de-serialisation and error checking to the end-user.

The difficulty of supporting a wide range of heterogeneous technologies in developing modern streaming applications led to the declarative streaming model. An initial implementation of this model used a relational approach. One drawback of this system was an inability to reason about user-defined functions, which limited the capabilities of the Optimiser.

Applications of Purely-functional programming have demonstrated that strong type systems and pure semantics are valuable tools for developing and maintaining software. Pure semantics in particular may permit an Optimiser to reason about user-defined functions.

Existing Haskell stream-processing systems have mostly focussed on efficient and predictable memory management and do not provide facilities for distributed stream-processing.

We will build a new implementation of the declarative stream-processing model, using purely-functional programming for both the implementation and exposing a purely-functional interface to the application programmer. We will leverage pure semantics in the design of our optimiser and provide a programming interface supporting strong-typing for streaming data.

Chapter 3. Partitioning and Deployment

In this chapter we discuss the design and implementation of two components from the declarative architecture (Figure 1.1): the Physical Optimiser and the Deployer. Completing these components allow a *StrIoT* user to write, deploy and execute stream-processing programs.

The Physical Optimiser is responsible for generating the partition map, a mapping of stream-processing operators to deployment nodes. We refer to the process as *partitioning*, and our Physical Optimiser as the Partitioner.

The Deployer translates a deployment plan — the combination of a stream-processing program and partition map — into the source code and configuration necessary to deploy the program in a distributed environment.

3.1 Physical Optimiser

The functional prototype (Section 2.5) considered the available nodes upon which the program should be deployed to be homogeneous: it represented them with simple integers that could be distinguished from each other, but did not model any of their individual properties. We have not developed this aspect of the prototype in our work and so we have inherited the simple homogeneous representation.

The Partitioner is the component responsible for generating all possible valid partition maps for a given stream-processing program. Our implementation is in [79, `src/Striot/Partition.hs`] . We now describe the algorithm, followed by a worked example.

3.1.1 Constraints

An operator can be assigned to no more than one deployment node. The upper limit on the number of deployment nodes for a given stream-processing program is therefore the number of operators within the program: one node per operator.

The runtime implementation imposes two constraints on permitted partition maps (described in Section 2.5.3): sources and sinks cannot co-exist on a node, and `streamMerge` operators must be placed as the first operator in a node.

3.1.2 Partitioning Algorithm

Recognising that a stream-processing program is a tree rooted at the sink operator, our partitioning algorithm is built around a traversal of the graph from the root node backwards.

The traversal maintains a list of partial partition maps, containing only the operators we have visited. The list is initially empty. For each operator:

1. Extend all partial maps with the current operator assigned to a new deployment node. For the special case of the sink operator, we add a new partial map, assigning the sink operator to a new node.
2. Unless the previously-visited operator was `streamMerge`, or both the current and previously-visited operator are source or sink nodes, we also extend all partial maps by assigning the current operator to the last-used node. This ensures the algorithm encodes the constraints specified in Section 3.1.1.

When we reach a branch, we apply the above algorithm to each sub-tree and then concatenate the resulting partial maps.

3.1.3 Worked Example

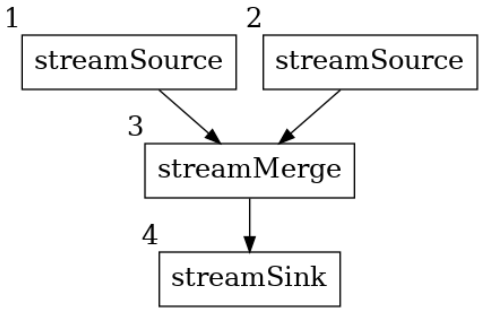


Figure 3.1: A simple example of a stream-processing program

We illustrate the algorithm using the simple stream-processing program in Figure 3.1.

We begin at the root sink operator (ID 4). Our list of in-progress partition maps is empty. We insert a new in-progress partition map, assigning the root sink operator to a new deployment node. The list now contains one partition map: `[[4]]`.

We now consider the `streamMerge` operator (ID 3). Assigning to a new node results in `[[3],[4]]`. Extending the in-progress partition map does not violate the constraints, so we also generate `[[3,4]]`.

Next we consider the `streamSource` operator (ID 2). Assigning to a new node updates the two in-progress partition maps to `[[2],[3],[4]]` and `[[2],[3,4]]`. We

cannot add the `streamSource` operator to the existing nodes as doing so would violate all of our constraints.

Similarly, the last `streamSource` (ID 1) cannot be added to the existing node. We finish with two valid partition maps: `[[1],[2],[3],[4]]` and `[[1],[2],[3,4]]`.

3.2 Data-Types for Representing Stream-Processing Programs

StrIoT's stream-processing operators (Section 2.5.2) were designed to be simple and easy to use. They are defined as standard Haskell functions with the intent that the end-user describes the stream-processing to take place in terms of these functions in a regular Haskell program.

In contrast, the Logical Optimiser and Partitioner components of the architecture have different design requirements: they need to manipulate the stream-processing program as data in order to apply transformations, divide it into sub-programs and generate code for those sub-programs to be independently compiled and executed within the deployment environment.

Two approaches that we could haven taken to manipulate program source code as structured data were to build a Haskell parser, or to define independent data-types.

3.2.1 Parsing

We could have incorporated a Haskell parser into *StrIoT* in order to read the user's stream-processing program as source code and build an equivalent data structure for processing.

This would be a significant undertaking. Haskell is a mature language with a complex, context-sensitive grammar. If our parser diverged from that of GHC, we could introduce errors or unexpected behaviours into the stream-processing program.

We judged to be too much of a diversion from the main focus of our research.

3.2.2 Independent Data-Types

The approach taken by the *StrIoT* prototype (Section 2.5) was to define a set of data-types well suited to the requirements of the Logical Optimiser and Partitioner and wholly independent from the stream-processing operators provided to the end-user. This design, including a bespoke graph implementation, is described in Section 2.5.3.

The main advantage of this approach is the design and implementation of the Logical Optimiser, Partitioner and Evaluator components can focus on their core responsibilities and not be concerned with issues of parsing source code.

Rather than develop, test and maintain our own graph implementation, we wished to leverage work to encode and manipulate graphs, so we explored the available graph libraries within the Haskell ecosystem.

3.2.3 Graph Libraries

Most graph libraries encode graphs as a pairing of a list of vertices $[V]$ and a list of edges, each of which is defined as a pair of vertices that the edge connects: $(v1, v2)$.

Traditional graph algorithms are typically imperative and adapting them to a purely-functional context can result in poor performance. The traditional representation also permits invalid graphs, such as a graph containing edges between vertices that do not occur in the list of vertices for the graph.

The *fgl* [26] library attempted to address the performance issue by defining graphs as *inductive types*. It provides a set of purely-functional graph algorithms. However, the library interface consists of partial functions. As a consequence, programming mistakes such as attempting to add an edge connecting a non-existing vertex (a malformed graph) are not caught at compile time, and result in a runtime error.

An alternative library `Algebra.Graph` [60] uses a novel approach: an algebraic representation of graphs, encoded as type constructors which are total functions. This prevents the construction of malformed graphs by making them unrepresentable.

Other features of `Algebra.Graph` library which were attractive for our purposes include the ability to pattern-match on graph constructors and its support for generating graphical depictions of graphs, via the GraphViz software [35].

For these reasons we opted to use `Algebra.Graph` as the basis of our stream-processing program data-type.

3.2.4 Vertex and *StreamGraph* types

`Algebra.Graph` provides the higher-kinded type `Graph a`, where `a` represents the type of vertices. The user must define and provide an appropriate vertex type.

For the vertices we defined a `StreamVertex` type to fully describe an instance of a stream-processing operator. The full type is provided in Listing 3.1[1]. Observe that none of the graph connectivity is encoded in the vertex type: that is entirely represented by the enclosing `Graph`.

For convenience, we also define a type alias `StreamGraph`.

```
data StreamVertex = StreamVertex
  { vertexId   :: Int
  , operator   :: StreamOperator
  , parameters :: [ExpQ]
  , intype     :: String
  , outtype    :: String
  }
type StreamGraph = Graph StreamVertex
```

Listing 3.1: The StreamVertex and StreamGraph types.

[1]These types were further modified during the development of cost models. See Section 5.7.1.

We now detail the constituent members of `StreamVertex`.

operator

We encode the operator type as an instance of a separate sum type, `StreamOperator` (Listing 3.2). We re-use the `StreamOperator` sum type from the *StrIoT* prototype. This consists of one value per stream-processing operator, as well as special values to represent sources and sinks.

```
data StreamOperator  = Map             |
                       Filter          |
                       Expand          |
                       Window          |
                       Merge           |
                       Join            |
                       Scan            |
                       FilterAcc       |
                       Source          |
                       Sink
                       deriving (Eq)
```

Listing 3.2: The `StreamOperator` data-type

input and output types

The input and output types for an operator are encoded as separate `String`s. It would have been possible to only encode the output type and to infer the input type by traversing the enclosing Graph to find the incoming edges, but we chose simplicity over conciseness.

parameters

The *StrIoT* prototype encoded operator parameters as `String`s. A significant drawback of this approach is it prevents us from taking advantage of the Haskell compiler for syntax or type checking, and introduces the risk of program errors being missed at compile time. Manipulating code-in-strings, such as building larger expressions, without any further insight into the structure of the expressions, can be difficult and error prone.

We have already discussed and dismissed the idea of implementing our own Haskell parser (Section 3.2.1).

Another approach is to leverage an existing Haskell parser. Template Haskell [69] (TH) is a meta-programming tool integrated into GHC [31]. With TH, a user can surround an expression in their source code with *oxford brackets* which replace the

expression by the result of parsing — an expression tree — which can be inspected and manipulated by the main program. Consider the following expression:

```
[| \x -> x + 1 |]
```

Listing 3.3: Lambda expression surrounded by Oxford brackets

From the perspective of a surrounding program, this segment of code corresponds to the expression in Listing 3.4:

```
LamE [VarP x_1]
    (InfixE (Just (VarE x_1))
            (VarE GHC.Num.+)
            (Just (LitE (IntegerL 1))))
```

Listing 3.4: TH Exp instance corresponding to Listing 3.3

Such instances can be straightforwardly constructed, traversed, and deconstructed (via pattern matching) by normal Haskell code.

TH provide a second pair of brackets that perform the reverse operation: take an instance of an expression structure, in terms of the TH expression types, and *splices* it into code within the surrounding program, to be compiled as normal. For example, The expression `1 + $( litE (IntegerL 2))` is evaluated at compile-time to `1 + 2`.

By encoding the stream operator parameters with Template Haskell types, the user can write real Haskell expressions which are syntax-checked by the Haskell compiler and the Deployer can safely manipulate the expressions as a data-type.

We identified several other opportunities to apply Template Haskell in order to simplify or otherwise improve our stream-program encoding. We discuss these in Section 7.3.5.

vertexId

`Algebra.Graph` requires vertices to be distinguishable from each other via equality, i.e., distinct vertices should not be considered equal through the `Eq` type class. Consider the program in Figure 3.2, consisting of two `streamFilter` operators connected to a `streamMerge`. The filters have identical properties: they apply the same predicate and produce output of the same type.

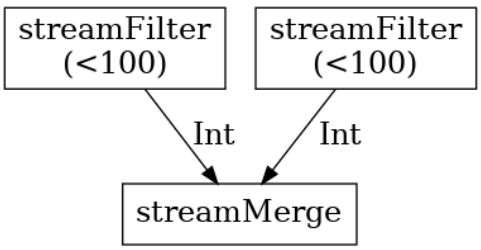


Figure 3.2: Two filters with identical properties.

Let's now consider a vertex type representing these operators. Listing 3.5 contains an encoding of the above example using an alternative vertex type which lacks the `vertexId` field of `StreamVertex`. Notice that the two filter operators, labelled `f1` and `f2`, are indistinguishable.

```
let f1 = Vertex (AltVertex Filter [[| (<100) |]] "Int" "Int")
    f2 = Vertex (AltVertex Filter [[| (<100) |]] "Int" "Int")
    m  = Vertex (AltVertex Merge  [] "Int" "[Int]")
    in  Overlay (Connect f1 m) (Connect f2 m)
```

Listing 3.5: Example Graph with indistinguishable vertices

`Algebra.Graph` considers these two filter operators to be the same vertex. Figure 3.3 illustrates the actual graph encoded by the code in Listing 3.5. Observe that it contains only one filter.

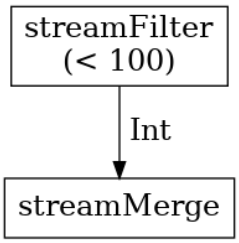


Figure 3.3: Demonstration of the interpretation of Listing 3.5.

In order to differentiate vertices representing these operators, we need to add a unique identifier to the type. For this we define the integer `vertexId`.

3.3 Deployment

The Deployer is responsible for taking a deployment plan (consisting of a `StreamGraph` and partition map) and generating the code for sub-programs for each node in the deployment.

We now illustrate the design and implementation of the Deployer by providing a full example of a deployable stream-processing program (Listing 3.6), taken from the *StrIoT* source distribution [79, `examples/pipeline/generate.hs`] . This simple program serves as a demonstration of some of *StrIoT*'s operators.

The program consists of a single source which emits a text string every second. There follows a series of transformations: first, the text string is appended to itself; then the result is reversed; finally the prefix "Incoming :" is pre-pended.

The stream is then aggregated into lists consisting of two events per list via the `chop` window-maker (Section 2.5.2) and finally printed.

Figure 3.4 contains a graphical depiction of the program.

```
1  import Striot.CompileIoT
2  import Striot.CompileIoT.Compose
3  import Striot.StreamGraph
4  import Algebra.Graph
5
6  source = [| threadDelay 1000000 >> return "Hello␣from␣Client!" |]
7
8  graph = path
9   [ StreamVertex 1 (Source 1) [source]              "IO␣()" "String" 0
10  , StreamVertex 2 Map    [[| \st->st++st |]]      "String" "String" 1
11  , StreamVertex 3 Map    [[| reverse |]]          "String" "String" 1
12  , StreamVertex 4 Map    [[| ("Incoming:␣"++) |]] "String" "String" 1
13  , StreamVertex 5 Window [[| chop 2 |]]         "String" "[String]" 1
14  , StreamVertex 6 Sink   [[| mapM_ print |]]     "[String]" "IO␣()" 0
15  ]
16
17 pmap = [[1,2],[3],[4,5,6]]
18 main = do
19     partitionGraph graph pmap defaultOpts
20     writeFile "compose.yml" (generateDockerCompose
21                              (createPartitions graph pmap))
```

Listing 3.6: *StrIoT* pipeline example

3.3.1 Representing Deployment Nodes

We define the type `PartitionMap` (Listing 3.7) as a list of required deployment nodes, each element of which is a list of operators assigned to that node.

```
type Partition = Int
type PartitionMap = [[Int]]
```

Listing 3.7: The `PartitionMap` type.

Since `StreamVertex`, our representation of operator instances, includes an unique integer ID, we represent operators within the partitioning plan as integers. The `PartitionMap` instance within Listing 3.6, line 17, assigns two operators to the first node (IDs 1,2), a single operator to the second (ID 3), and the remaining operators to a third node (IDs 4,5,6).

We discuss deriving possible partition maps from a given `StreamGraph` in Section 3.1.

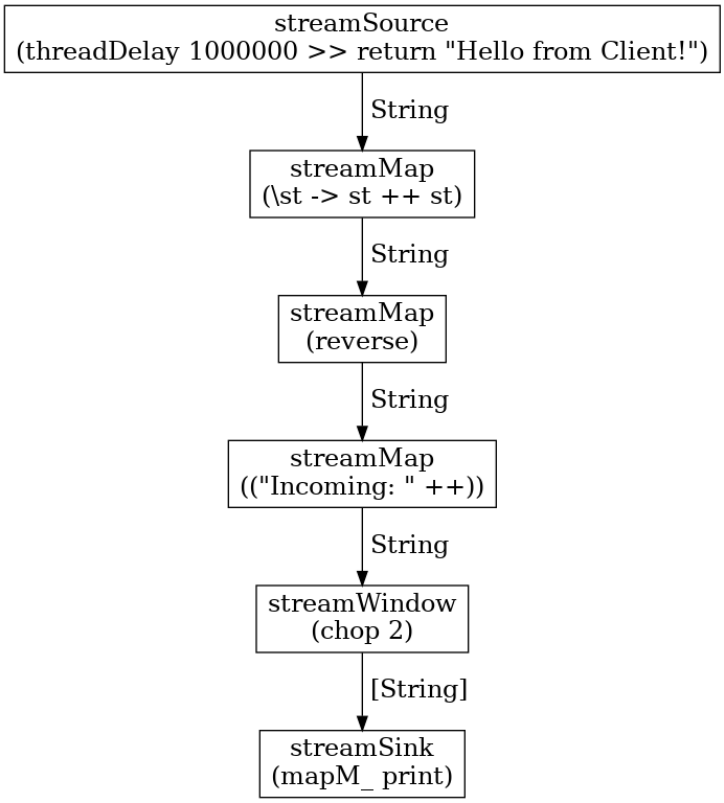


Figure 3.4: Graphical depiction of `StreamGraph` from Listing 3.6

3.3.2 *partitionGraph*

```
partitionGraph :: StreamGraph
               -> PartitionMap
               -> GenerateOpts
               -> IO ()
```

Listing 3.8: Type signature for `partitionGraph`

The program in Listing 3.6 calls `partitionGraph`, [79, `src/Striot/CompileIoT.hs#409`] a high-level function designed to be the entry-point into the deployment code. The type signature is provided in Listing 3.8.

`partitionGraph` is responsible for converting the stream-processing program into a partitioned representation (described in Section 3.3.3) and generating source code corresponding to the sub-programs within (Section 3.3.4).

3.3.3 Creating *PartitionedGraph*s

The result of partitioning a `StreamGraph` is a list of the sub-programs and a description of the inter-node connections between them in a deployment. We were able to re-use the `StreamGraph` data-type for both purposes.

The inter-node connections are encoded as a `StreamGraph` consisting of only the edges between the vertices that serve as the ingress and egress points for the sub-programs.

For example, consider the stream-processing program and partition map in Listing 3.6. The node boundaries in the partition map occur between the operators with IDs 2-3 and 3-4. Figure 3.5 containers a graphical depiction of the `StreamGraph` representing these inter-node connections.

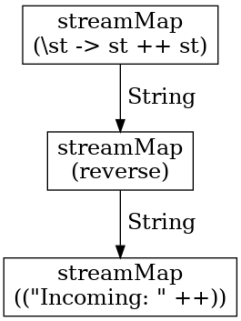


Figure 3.5: Graphical depiction of `PartitionedGraph` inter-node connections

The resulting type `PartitionedGraph` is listed in Listing 3.9.

```
type PartitionedGraph = ([StreamGraph], StreamGraph)
```

Listing 3.9: `PartitionedGraph` type.

To build a `PartitionedGraph` instance which can be used in the later stages, we define `createPartitions` [79, `src/Striot/CompileIoT.hs#67`] , with the type signature listed in Listing 3.10.

```
createPartitions :: StreamGraph -> PartitionMap -> PartitionedGraph
```

Listing 3.10: Type signature for `createPartitions`

`createPartitions` recursively traverses the `PartitionMap` list, building up the constituent parts of `PartitionedGraph` for each element.

To provide assurance that `createPartitions` was correct, we wrote the dual function `unPartition` to perform the reverse operation: collapse a `PartitionedGraph` back into a single `StreamGraph`. Using `unPartition` we were able to write tests which checked to ensure the result of partitioning and subsequently un-partitioning a `StreamGraph` was equal to the original `StreamGraph`.

The implementation of `unPartition` is provided in Listing 3.11.

```
unPartition :: PartitionedGraph -> Graph StreamVertex
unPartition (a,b) = overlays (b:a)
```

Listing 3.11: `unPartition` to convert `PartitionedGraph` back to `StreamGraph`.

3.3.4 Code Generation

The code generator routines are responsible for producing the source code for each of the sub-programs in the deployment plan. An example of the generated code for one sub-program derived from Listing 3.6 is provided in Listing 3.12.

```
1  -- node3
2  import Striot.FunctionalIoTtypes
3  import Striot.FunctionalProcessing
4  import Striot.Nodes
5  import Control.Concurrent
6  import Control.Category ((>>>))
7
8  sink1 :: Show a => Stream a -> IO ()
9  sink1 = mapM_ print
10
11 streamGraphFn :: Stream String -> Stream [String]
12 streamGraphFn n1 = let
13     n2 = (\s -> streamMap (("Incoming: " ++)) s) n1
14     n3 = (\s -> streamWindow (chop 2) s) n2
15     in n3
16
17 main :: IO ()
18 main = nodeSink (defaultSink "9001") streamGraphFn sink1
```

Listing 3.12: One of three sub-programs generated by Listing 3.6

Generated programs all begin with a comment indicating which node they are (line 1: the third node this example). Then follows a list of Haskell `import` statements (lines 2-6), which import the *StrIoT* routines into the program, as well as other supporting libraries.

Nodes containing sources or sinks require a function of return type `IO ()` which is responsible for either producing data in the case of sources (for example by reading from sensors, or a file on disk) or consuming the data at the end of the stream-processing program. In Listing 3.12 a sink function is generated (lines 8-9), corresponding to the final `StreamGraph` operator in the source program Listing 3.6.

The code generator produces a function `streamGraphFn` (lines 11-15) for all node types. This encodes the portion of the original stream-processing program assigned to the given node. In the running example, two operators (besides the final sink) are assigned to the node: `streamMap` and `streamWindow`.

Each operator is converted from the `StreamVertex` representation into an invocation of the basic operator functions described in Section 2.5.2. To support operator parameters referencing the source stream, we wrap each invocation in an anonymous function, using the identifier `s` for the incoming stream.

The operator parameters themselves are converted from the Template Haskell representation into strings of Haskell source code via the splice operator (See Section 3.2.4).

The sequence of operators are chained together with a series of `let`-bindings. This

clearly separates out the individual operators and their parameters in the generated function and aided with debugging during the development of the code generator.

Finally, we generate the `main` entry-point for the program (lines 17-18). This calls one of the run-time functions ([79, `src/Striot/Nodes.hs`] , described in Section 2.5.3) `nodeSource` (for source nodes), `nodeSink` (for sink nodes, as in this example) or `nodeLink` for other nodes.

3.3.5 Generation Options

We defined the `GenerateOpts` data-type (Listing 3.13) to specify options to the code generator in addition to the stream-processing program. These include specifying a function that should be run at the beginning of operation for a source node, for example to set up file descriptors for the source function to read from; and to specify additional libraries, required by the user's code, that need to be added to the import list.

```
data GenerateOpts = GenerateOpts
  { imports            :: [String]
  , packages           :: [String]
  , preSource          :: Maybe String
  , rules              :: [LabelledRewriteRule]
  , maxNodeUtil        :: Double
  , maxBandwidth       :: Double
  }
```

Listing 3.13: `GenerateOpts` data-type

3.3.6 Containers and Orchestration

We have retained and extended the *StrIoT* prototype's support of OCI-format containers (Section 2.3) for deployment and Docker Compose (Section 2.3.1) for orchestration, which are described in Section 2.5.3.

We generate the `Dockerfile` necessary for building the container to be deployed. Listing 3.14 is an example of a generated `Dockerfile` for a deployment node produced by the stream-processing program in Listing 3.6.

```
FROM ghcr.io/striot/striot:main
WORKDIR /opt/node
COPY . /opt/node

RUN ghc node.hs
EXPOSE 9001
CMD /opt/node/node
```

Listing 3.14: An example generated `Dockerfile` for a deployment node

We also generate the Docker Compose configuration, describing the interlinking of nodes from the `PartitionedGraph`. Listing 3.15 contains the `compose.yml` generated by `generateDockerCompose` on line 20 of Listing 3.6.

```
services:
    node1:
        build: node1
        tty: true
        depends_on:
        - node2
    node2:
        build: node2
        tty: true
        depends_on:
        - node3
    node3:
        build: node3
        tty: true
```

Listing 3.15: An example generated configuration for Docker Compose

3.3.7 Parallel Execution

The run-time functions we have supplied with *StrIoT* ([79, `src/Striot/Nodes.hs`] , described in Section 2.5.3) are all serially executed: they do not support executing different parts of the stream-processing program in parallel.

For example, consider the simple program from Figure 3.1: It is not possible to deploy and run a *StrIoT* program with the two source operators 1 and 2 allocated to the same deployment node.

The partitioning algorithm described in Section 3.1 will not generate a partition map which allocates the two source operators to the same deployment node.

StrIoT addresses parallel execution by requiring parallel operations to be allocated to separate deployment nodes.

Horizontal scaling

The pure semantics of the stream-processing operators permit *horizontal scaling*: distributing the workload of a single logical operator to more than one physical machine. The focus of this thesis is on logical optimisation and we did not explore horizontal scaling as part of this work.

Cattermole separately implemented support for horizontal scaling of the stateless *StrIoT* operators by extending the `Event` type (Section 2.5.1) iwith a unique sequence number and using it as the input for a key-based sharding algorithm [8].

3.4 Chapter Summary

We have described how we replaced the Deployer component from the prototype (Section 2.5). We have focussed on the issue of designing appropriate data-types for representing a program to be deployed, and the trade-offs of the approaches we explored, including our chosen solution. There is a discussion of potential future work on representation in Section 7.3.5.

Noting that our representation of deployment nodes is unchanged from the basic homogeneous representation from the prototype, we described the design and implementation of our Physical Optimiser, a process we call partitioning.

We described the design and implementation of our Physical Optimiser, a process we call partitioning. Our representation of deployment nodes is unchanged from the basic homogeneous representation from the original prototype. This limitation, as well as design limitations in the runtime, constrain the deployment plans that are supported. We discuss supporting heterogeneous ndoes as future work in Section 7.3.2.

The completion of these components from the declarative model (Figure 1.1) allow us to write, deploy and execute sample stream-processing programs.

This work provides an end-to-end framework upon which we can design and evaluate the Logical Optimiser (Chapter 4) and Evaluator (Chapter 5) components from the architecture.

Chapter 4. Logical Optimisation

In this chapter we describe the design and implementation of a Logical Optimiser: the component from the declarative architecture (Figure 1.1) responsible for generating modified versions of the user's stream-processing program. This is a key component of the architecture, as it is required to provide variants to the Evaluator (discussed in Chapter 5).

The stream-processing operators defined in *StrIoT* (Section 2.5.2) are pure functions. This property enables the use of term rewriting – a powerful technique for rewriting programs – in the design of the Logical Optimiser.

Term rewriting requires a catalogue of rewrite rules. We design an initial set of semantically-preserving rewrite rules suitable for stream-processing programs using a manual, systematic analysis of each possible pairing of stream-processing operators. Where possible we categorise these rules according to established categories of stream-processing optimisations.

We initially approach rewriting as being semantically-preserving before discovering that some semantics — such as stream ordering — are not important for the program's correct operation and changing them can improve performance.

We investigate tools to gain assurance that rewrite rules are semantically-preserving. Finally we describe an initial exploration into machine-assisted derivation of rewrite rules.

4.1 Term Rewriting

StrIoT is built with *Haskell*, a purely-functional programming language. Purely-functional expressions are *referentially transparent*, and can be substituted for any other expression which evaluates to the same value for the same inputs. This enables equational reasoning, a technique for transforming functions through a process of substitution by applying laws, or rules [5].

A rewrite rule consists of an equation, with an expression (the pattern) on the left-hand side and a functionally-equivalent expression on the right-hand side [3]. We delimit the left and right-hand sides with "$\longrightarrow$". Listing 4.1 is an example of two basic rewrite rules for arithmetic expressions.

$$a + 0 \longrightarrow a$$
$$a \times 1 \longrightarrow a$$

Listing 4.1: examples of simple arithmetic rewrite rules.

To apply a rule to a candidate expression, we must first determine that the left-hand pattern matches the expression. If so, the candidate can be substituted with the right-hand side of the rule.

The left-hand pattern may contain variables which serve as placeholders for arbitrary sub-expressions. For example, in the rule $a \times 2 \longrightarrow a + a$, the left-hand side features the variable a. When matching this rule to the expression 4×2, the sub-expression 4 is *bound* to the variable a.

The right-hand side may contain references to the same variables. When applying the rule, those references are substituted for the bound values. In the running example, each occurrence of the variable a in the expression $a + a$ is substituted for the value 4, resulting in $4 + 4$.

Equational reasoning is an attractive technique for use in the design of a Logical Optimiser, as valid program transformations can be concisely represented as rewrite rules. Rewrite rules have been successfully deployed as a compiler optimisation tool within the Glasgow Haskell Compiler (GHC) [66]. We were also inspired by the use of term rewriting to build a simple equational calculator as an exercise in [4].

We began the work described in this chapter with the view that rewrite rules must be semantically-preserving: a rewrite rule that changed the meaning of a program could introduce errors. We discovered later that preserving certain semantic behaviours was unimportant for our particular application, and that, in some cases, it was beneficial to change them. This is discussed in Section 4.4.3.

4.2 Semantic Analysis of the Operators

StrIoT's operators are described in Chapter 2, Section 2.5.2. In this section we explore and compare the semantics of the operators and their parameters.

When analysing the semantics of the stream-operators, we assume that we have no insight into their parameters (e.g. the user-supplied transformation function for `streamMap`) beyond what can be determined by their type.

For example, `streamMap` may change the type of a stream. Even though many common mapping operations do not , e.g. `streamMap (+1)` (which receives and emits numbers), in the general case we cannot assume that the type will be unchanged.

This impacts the kind of rewrites we can design: we could not, for example, reorder `streamFilter p . streamMap f` without accounting for the change in stream type. (This particular pairing is explored in Rule 11, below.)

4.2.1 Higher-Order Functions

`streamMap`, `streamFilter`, `streamScan` and `streamFilterAcc` are higher-order functions that accept parameters which operate only on the payload of a stream, and not on the `Stream` or `Event` types themselves. The parameters therefore cannot influence metadata such as the time-stamp of `Event`s in the stream.

The implementation of `streamMap` and `streamScan` ensure that the value returned by their parameter functions replace the original value: they cannot introduce newly-synthesised `Event`s into the stream, nor prevent them from being emitted.

The implementation of `streamFilter` and `streamFilterAcc` ensures that `Event`s are either rejected or emitted unaltered.

4.2.2 Operators with Memory

`streamMap` and `streamFilter` are memoryless: they operate solely on the value of the `Event` under consideration. By contrast, `streamScan` and `streamFilterAcc` have memory: past events can influence the evaluation for future ones. When designing rewrite reles involving operators with memory, we have to be concious that altering which events arrive at the operator, or the event ordering, could alter its behaviour.

4.2.3 *`streamMerge`*

`streamMerge` interleaves events from multiple incoming streams. It doesn't modify the data or `Event`s wrappers that it receives.

Arity

`streamMerge` takes a single argument, the list of incoming streams, which has a type of `[Stream a]`. This type can be satisfied by an empty list. This corresponds to a merge with no incoming streams. This cannot occur in a useful stream-processing program, since without data, no processing can take place. For this reason we disregard the case of `streamMerge` with an empty list in the following analysis.

It is also possible to satisfy the type with a list consisting of a single element, corresponding to a single incoming stream. In this scenario, `streamMerge` is not performing any useful work and could be eliminated. (See Rule 8).

For the remaining analysis of `streamMerge`, we assume that there are at least two incoming streams.

Pairings

When considering pairings of some operator connected to a `streamMerge` operator, we need to determine the placement of the first operator within the list of inputs to the `streamMerge`.

For example, consider the pairing of `streamFilter` followed by a `streamMerge`. Instances of the filter could be present on all incoming streams, or a sub-set, such as one, depicted in Figure 4.1.

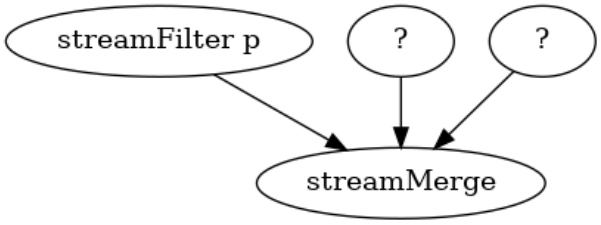


Figure 4.1: A pairing of `streamFilter` and `streamMerge` with further unknown streams

In the case of a sub-set, it would be very difficult to design rewrite rules that preserved the semantic behaviour without more information about the other inputs. Therefore in designing rewrite rules, we only considered the case where instances of the first operator (e.g. `streamFilter`) occur on all incoming streams (Figure 4.2).

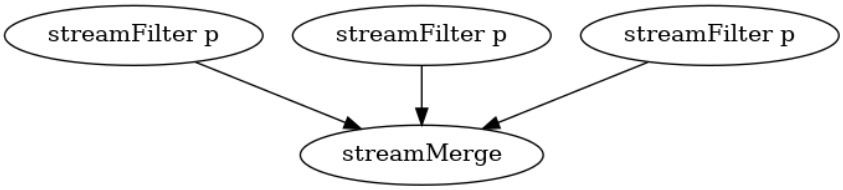


Figure 4.2: Pairing of `streamFilter` and `streamMerge` with the filter present on all inputs

4.2.4 Event Ordering

Rewriting has the potential to alter the order of events within a stream.

For a pair of connected `streamMerge` operators, the order of emitted events is determined by the presence or absence of time-stamps on incoming events and the placement of the first `streamMerge` operator within the list of incoming streams to the second: i.e. does it occur at the beginning, middle, or end of the list of incoming streams.

We have designed a set of rewrite rules that can cause stream re-ordering but may be useful in situations where the stream order does not matter. We have collected these separately to the main rule set, in Section 4.5.6.

In a real distributed environment with events arriving over a potentially unreliable or unpredictable network such as the Internet, packets (and therefore `Events`) could be lost, re-transmitted, or delayed, and in the general case their arrival order is therefore unpredictable [28].

It's likely that the exact ordering of events may be unimportant to the stream-processing program. For example, a program responsible for calculating a running

average of sensors readings received from a set of temperature sensors will be unaffected by the arrival order of the readings.

StrIoT does not provide any automated means for managing stream ordering. If stream order is important, the user must perform the necessary re-ordering manually within the stream-processing program. This could be achieved through the use of `streamWindow`, to collect a batch of sufficient size for the required re-ordering, followed by a `streamMap` to re-order the batched events and `streamExpand` to transform them back into a top-level stream for further processing.

The practitioner must determine whether stream order is important by understanding the specifics of their program logic. We have not explored automation or assistance for this process.

4.2.5 *streamWindow*

In contrast to the parameters for the higher-order functions (Section 4.2.1), the parameter for `streamWindow` (referred to as the window-maker), with type `Stream a -> [Stream a]`, operates on the full stream rather than being limited to reasoning solely about the data within.

The four window-makers provided with *StrIoT* (described in Section 2.5.2) preserve incoming data unmodified in their output. The two time-based window-makers reason solely about event time-stamps and not data from the incoming stream.

By contrast a user-supplied window-maker could be defined in terms of the data within the stream. For example Listing 4.2 demonstrates an example window-maker which produces lists of copies of each incoming value, where the length of each list is determined by the value of the incoming data. For the input `[1,2,3]`, this example window-maker produces the output `[[1],[2,2],[3,3,3]]`.

```
wmInspect [] = []
wmInspect ((Event _ Nothing):s)     = wmInspect s
wmInspect (e@(Event _ (Just i)):s) = take i (repeat e) : wmInspect s
```

Listing 4.2: A window-maker which inspects `Stream` data

A window-maker could in theory select only a sub-set of data from the input and discard the rest; or mutate the data or metadata in some way; or even emit lists of streams with no correlation to the input.

This versatility makes analysis of the `streamWindow` operator using an arbitrary window-maker function very difficult. For the rules involving `streamWindow` in Section 4.5, we note that the rules only apply when the window-maker's behaviour does not depend upon the values of the data in the incoming stream.

4.2.6 `streamExpand`

`streamExpand` receives lists of data and emits a new `Event` for each item in each list. Empty lists are ignored. The emitted `Events` derive their time-stamp from the incoming `Event` within which they originated.

4.2.7 `streamJoin`

`streamJoin` receives two input streams and creates an output stream consisting of pairings of data from the inputs. It will only emit an event once it has received an event containing data (not `Nothing`) on both inputs. Events received without data are ignored.

When analysing a pairing of `streamJoin` preceded by another operator, the operator could be connected to either of the inputs, and we know nothing about the other input. To simplify analysis (and later, implementation), we only consider the case where the preceding operator is the second input to `streamJoin`, as depicted in Figure 4.3.

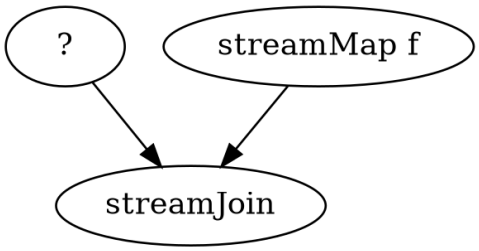


Figure 4.3: Example of `streamMap` connected to the second input of `streamJoin`

This is an arbitrary choice. The related rules we have designed (Rule 14, Rule 15) can be straightforwardly adjusted to operate in the reverse scenario, with the preceding operator under consideraetion instead attached as the first input to `streamJoin`.

4.3 Categories of Stream-Processing Optimisations

Hirzel et al [38] identify ten categories of stream-processing optimisations, five of which are logical optimisations: *Operator reordering*, *Redundancy elimination*, *Operator separation*, *Fusion* and *Fission*.

Whilst designing rewrite rules, we examined each rule to determine whether it belonged to one of these categories. We present the rules first grouped by category, followed by those that did not correspond to the existing categories.

We did not discount any rules that we could not categorise: the utility of a rule in isolation may not be immediately obvious but it may create further optimisation opportunities by moving two operators adjacent to each other, such that a previously-inapplicable rule becomes suitable for applying subsequently.

As described in Section 4.2, we have no insight into the arguments to stream operators (such as the user-supplied predicate for `streamFilter`) and so did not reason

about them as part of the process. If we did explore the semantics of operator parameters, we could potentially have written rules implementing *Operator separation* – separating an operator into distinct sub-operators – but would require us to reason about the structure of Haskell expressions, which we judged out of scope for this work.

4.4 Creating Rewrite Rules

To build an initial catalogue of rewrite rules, we performed a systematic analysis of each of 64 pairs of *StrIoT*'s eight operators.

4.4.1 Method

For each pairing, we considered that they were adjacent operators in a larger stream-processing program and looked for transformations we could make which would not alter the program semantics. We determined whether the operators could be swapped; fused into a single operator; or if one or both could be removed altogether.

This was a manual exercise. Our goal was to produce a useful list of rules, but not to be exhaustive, so after analysing a given pairing for a reasonable interval we stopped and moved onto the next pairing.

4.4.2 Validating rewrite rules

This systematic — but manual — process introduces the risk that if we made a mistake, an incorrect rewrite rule could change the semantics of a stream-processing program such that it produced incorrect results.

To gain assurance that the rules were correct, we used the property-testing tool QuickCheck[12] and wrote an accompanying set of invariant properties for each rule. This is described in Section 4.6.

4.4.3 Rules That Alter Semantics But May Still Be Useful

During this process, it became clear that there were several potentially useful rules that, upon inspection, were not semantically-preserving. Rather than reject them, we catalogue them separately below in Sections 4.5.6 and 4.5.7.

4.5 Rewrite Rules

Table 4.1 summarizes the rules designed during this process. The rules are described individually in the remainder of this section.

Each rule is presented as a pair of Haskell expressions separated by an arrow ($\longrightarrow$). The expressions consist of stream operators connected by standard symbolic Haskell operators such as function composition (.), which reads right-to-left. Operator parameters are presented as single characters, representing free variables.

	filter	map	filterAcc	scan	window	expand	join	merge
filter	1, 10	-	2	-	W1	-	-	R1
map	11	5	12	6	21	-	14	16
filterAcc	3	-	4	-	W2	-	-	-
scan	-	-	-	-	22	-	15	-
window	-	-	-	-	-	-	-	-
expand	9	17	13	18	-	20	-	R2
join	-	-	-	-	-	-	-	-
merge	R3	19	-	-	-	R4	-	7, 8, R5

Table 4.1:
Operator pairs that yielded rewrite rules. The row indicates the first operator in a pairing and the column the second.
Arabic numbers correspond to semantically-preserving rules.
R-prefix correspond to re-ordering rules from Section 4.5.6.
W-prefix correspond to window reshaping rules from Section 4.5.7.
"-" means no rule was discovered.

4.5.1 Operator Fusion

All four combinations of `streamFilter` and `streamFilterAcc` can be replaced by a single filtering operator, i.e., fused together.

1.
```
streamFilter q . streamFilter p
  ⟶ streamFilter (\e -> p e && q e)
```

2.
```
streamFilterAcc f a p . streamFilter q
  ⟶ streamFilterAcc
    (\a' v -> if q v then f a' v else a)
    a
    (\x a' -> q x && p x a')
```

3.
```
streamFilter q . streamFilterAcc f a p
  ⟶ streamFilterAcc f a (\x a' -> p x a' && q x)
```

4.
```
streamFilterAcc g b q . streamFilterAcc f a p
  ⟶ streamFilterAcc
    (\ (x,y) v -> (f x v, if p v x then g y v else y))
    (a, b)
    (\x (y,z) -> p x y && q x z)
```

5.
```
streamMap f . streamMap g ⟶ streamMap (f . g)
```

6. ```
   streamScan g a . streamMap f
     ⟶ streamScan (flip (flip g . f)) a
   ```
7. ```
   streamMerge (s1 ++ [streamMerge s2]) ⟶ streamMerge (s1 ++ s2)
   ```

   For a pairing of `streamMerge`s with this right-handed topology, fusing the operators preserves the ordering of events. For other topologies, see Rule R5.

4.5.2 Operator Elimination

8. ```
   streamMerge [s] ⟶ s
   ```

   For the case where there is exactly one incoming `Stream` to `streamMerge`, the operator can be eliminated.

4.5.3 Operator Re-Ordering

Rules which move filtering closer to source can result in a reduction of the volume of data flowing through the stream-processing program. This can have a positive impact on non-functional requirements such as minimizing the data transmitted over a network link.

9. ```
   streamFilter p . streamExpand
     ⟶ streamExpand . streamMap (filter p)
   ```
10. ```
    streamFilter p . streamFilter q
      ⟶ streamFilter q . streamFilter p
    ```

    A pair of adjacent `streamFilter`s are commutative.

A mapping-type operator preceding a filtering-type operator can be swapped by composing the mapping operation `f` with the filter predicate `p`. (The reverse is not true, see Section 4.5.8).

The size of the lists received by `streamExpand` is reduced by the preceding `streamMap`.

11. ```
    streamFilter p . streamMap f
      ⟶ streamMap f . streamFilter (p . f)
    ```
12. ```
    streamFilterAcc g a p . streamMap f
      ⟶ streamMap f . streamFilterAcc g a (p . f)
    ```
    ```

13.
```
streamFilterAcc f a p . streamExpand
  ⟶ streamExpand
    . streamMap (reverse.fst)
    . streamScan (\(_,b) a' -> filterAcc f b p a') ([], a)
```

This rewrite moves stateful filtering inside a window.

`streamFilterAcc`, which operates on a potentially infinite stream, is replaced by `filterAcc`, an implementation for finite lists. The implementation of `filterAcc` is provided by *StrIoT* [79, `src/Striot/FunctionalProcessing.hs#219`] .

The accumulator value needs to be carried over between invocations of `filterAcc`. This is achieved with `streamScan`. Finally we discard the accumulator (and reverse the output lists to reflect their proper ordering) as a final clean-up, via `streamMap`.

The next two rules involve `streamJoin` as the second operator.

Moving a `streamMap` or `streamScan` downstream of `streamJoin` means handling wrapping and unwrapping the original parameters to operate on one side of the tuple. The other side of the tuple is passed through unmodified.

14.
```
streamJoin s . streamMap f
  ⟶ streamMap (\(x,y) -> (x, f y)) . streamJoin s
```

15.
```
streamJoin s . streamScan f a
  ⟶ streamScan (\c (x,y) -> (x, f (snd c) y)) (undefined, a)
    . streamJoin s
```

16.
```
streamMerge [(streamMap f s),(streamMap f t)]
  ⟶ streamMap f $ streamMerge [s,t]
```

17.
```
streamMap f . streamExpand
  ⟶ streamExpand . streamMap (map f)
```

18.
```
streamScan f a . streamExpand
  ⟶ streamExpand
    . streamScan (\b a' -> tail $ scanl f (last b) a') [a]
    . streamFilter (/=[])
```

This rule follows the basic pattern of Rule 17: lifting a stream-mapping function that operates on a potentially infinite stream into an analogous function operating on finite lists. In this case, the list function analogous to `streamScan` is `scanl`,

which behaves slightly differently: it returns the accumulator seed value as the first value, without applying the accumulator function to it. We account for this by applying the accumulator function to the seed value that we supply to `scanl`.

`streamExpand` silently ignores empty lists. For the rewritten rule, this is achieved with a `streamFilter` inserted before the re-ordered pair.

19.
```
streamMap f $ streamMerge [s, t]
  ⟶ streamMerge [streamMap f s, streamMap f t]
```

The dual of Rule 16.

4.5.4 Other Semantically-Preserving Rules

The following rule does not belong to the established categories of stream-processing optimisations.

20.
```
streamExpand . streamExpand
  ⟶ streamExpand . streamMap concat
```

4.5.5 Type-Constrained Rules

If the first operator in a pair is a mapping-type operator which receives a stream of type a and emits a Stream of type b, and the second is `streamWindow`, the pair can be swapped if the `streamWindow`'s type permits receiving `Events` of type a as well as b. This is possible in two circumstances:

- If a and b are the same concrete type: i.e., the mapping-type operator does not change the type of the stream. For example `streamMap (+1)` receives and emits the same type.
- If the `streamWindow` has a polymorphic type, which is true for the four window-makers provided by *StrIoT*. For example, `streamWindow (chop 3)` accepts streams of any type.

Since an arbitrary window-maker could be defined in terms of the values of data in the stream (See Section 4.2.5), the following two rules only apply when this is not the case.

21.
```
streamWindow w . streamMap f
  ⟶ streamMap (map f) . streamWindow w
```

22.
```
streamWindow w . streamScan f a
  ⟶ streamScan (\b a' -> tail $ scanl f (last b) a') [a]
    . streamWindow w
```

4.5.6 Stream Re-Ordering Rules

Five promising-looking rules discovered during the process, all involving the `streamMerge` operator, are not semantically-preserving, as they cause the ordering of events to change. However, as discussed in Section 4.2.4, in real-world scenarios precise ordering may be non-deterministic or unimportant to the stream processing.

Rather than reject such rules, we instead collected them separately to the main rule set.

For some of these rules, it may be possible to design additional operators to re-order the stream back to match the behaviour prior to applying the rewrite. This is discussed in more detail in Section 7.3.1.

R1
```
streamMerge [streamFilter p t, streamFilter p s]
  ⟶ streamFilter p (streamMerge [t, s])
```

R2
```
streamMerge [streamExpand s, streamExpand t]
  ⟶ streamExpand (streamMerge [s, t])
```

R3
```
streamFilter p $ streamMerge [s, t]
  ⟶ streamMerge [streamFilter p s, streamFilter p t]
```

R4
```
streamExpand $ streamMerge [s, t]
  ⟶ streamMerge [streamExpand s, streamExpand t]
```

R5
```
streamMerge (a ++ streamMerge b : c)
  ⟶ streamMerge (concat [a, b, c])
```

Rule 7, applicable to a pair of merges arranged in a right-handed topology, is order-preserving. As described in Section 4.2.3, merging pairs of `streamMerge` for other topologies will alter the order of events. This rule represents those topologies. The variables a, b and c represent lists of incoming streams, which could be empty.

4.5.7 Rules That Reshape Windows

As described in Section 4.2.5, the behaviour of `streamWindow` might depend upon the values of events, making them very versatile but also hard to reason about in the

general case. When it can be determined that a particular `streamWindow` does not depend upon the values within the stream, there are some opportunities to apply rewrite rules.

The following rules are applicable when the window-maker argument is known not to depend upon the data of stream events, such as all of the window-makers provided with *StrIoT* (Section 2.5.2). Similar to the `streamMerge` rules described earlier, these rules are not strictly semantically-preserving: although they preserve all of the stream data, the exact composition of the windows is changed.

W1
```
streamWindow w . streamFilter p
  ⟶ streamMap (filter p) . streamWindow w
```

Consider when the `streamFilter` predicate is `odd`, selecting only odd values, and the window-maker is `chop 3` (described in Section 2.5.2). An input stream of `[0..11]` would result in an output of `[[1,3,5], [7,9,11]]`.

Moving the filtering after the windowing means that the lists emitted by the `streamWindow` operator will include elements that are later rejected. The rejection will cause the output lists to vary in length. For example, the input `[0..11]` would result in initial windows of `[0,1,2]`, `[3,4,5]`, `[6,7,8]`, `[9,10,11]`, before being transformed by `streamMap` into `[1]`, `[3,5]`, `[7]`, `[9,11]`. Notice that in both cases the values within the lists are the same, but the *shape* of the enclosing lists are different.

W2
```
streamWindow w . streamFilterAcc f a p
  ⟶ streamMap (reverse.fst)
    . streamScan (\ (_,b) a' -> filterAcc f b p a') ([], a)
    . streamWindow w
```

This is a variation of Rule W1 with the same caveats. The translation of `streamFilterAcc` into `filterAcc` uses the same approach as described in Rule 13.

4.5.8 Combinations That Do Not Yield Rules

Whilst we considered all 64 combinations of pairs of operators during this exercise, we were not attempting to exhaustively determine every possible transformation: if, after a reasonable interval, we could not devise a rule, we moved onto the next pairing.

However, during analysis, we were able to determine that for certain pairings, no transformation was possible. In many of these cases, we were able to establish this due to Haskell's strong type-system.

Filter-Type Operators Followed By Mapping-Type Operators

Consider a stream of Integers, a filter which selects numbers, and an incrementing mapping function:

```
streamMap (+1) . streamFilter even
```

An input stream of $[1, 2, 3, 4, 5]$ applied to this pairing would result in an output of $[3, 5]$.

If we swapped the operators, the stream would be first transformed to $[2, 3, 4, 5, 6]$ before being filtered down to $[2, 4, 6]$, which differs from the original behaviour.

In Rule 11, we achieved the opposite re-ordering by composing the filter predicate and the map parameter. In this case, we need a function which performs the reverse of the map operation, such as `streamMap (-1)` for the previous example. In the general case, we cannot construct a reverse operation for an arbitrary map parameter.

Pairs Beginning With `streamJoin`

`streamJoin` receives two streams, of types a and b, and emits a stream of tuples (a, b). The second operator in all eight pairings beginning with `streamJoin` therefore receives a tuple type.

This is not the same type as required by `streamExpand` (a list), and so `streamExpand` cannot occur after `streamJoin`. Similarly, a pairing ending in `streamMerge` must emit a list: since `streamJoin` doesn't emit a list, a pairing of `streamMerge` and `streamJoin` cannot be re-ordered.

Filter predicates, mapping-type operator parameters, and user-supplied window-makers may be defined in terms of either or both sides of the tuple. To re-order these operators prior to `streamJoin`, the operator would need to be placed on one of its two incoming streams. The operator would then not have access to the data from the other incoming stream, and so we cannot reason about its behaviour.

Mapping Before `streamExpand`

Since the input type to `streamExpand` must be a list, we know that a preceding `streamMap` or `streamScan` operator must produce a list. However, the input and output types of mapping operators are not necessarily the same (type `a -> b`) so we cannot infer that type `a` is also a list. We therefore cannot re-order the operators such that `streamExpand` consumes type `a`.

For example, `streamMap (x-> take x (repeat x))` has input type `Stream Int`, which is not an appropriate input for `streamExpand`.

Operators With Memory

`streamScan` carries state between invocations, so past `Events` can influence the value of `streamScan` for future `Events`. Changing the order of events (including by filtering some out) may alter the results. We cannot therefore move a mapping or filtering operation from after a `streamScan` to before it.

This is in contrast to `streamFilterAcc`, where in many cases (Rules 2 to 4) we were able to re-order and fuse filters because we could compose new filter accumulator update functions independently of the output value of the function. With `streamScan`, the accumulator is the output of the function, so we cannot take the same approach.

For the same reason, we cannot move or duplicate a `streamScan` or `streamFilterAcc` up or downstream of a `streamMerge`: the exact events and their order could be important.

As described in Section 4.2.5, the versatility of `streamWindow`'s window-maker parameter means the exact order of incoming `Events` may affect the output it produces. We therefore cannot move `streamWindow` downstream from a `streamMerge`, as this would result in a different event ordering.

Operators Downstream Of Windows

An operation downstream of `streamWindow` could depend upon properties of the lists it generates. For example, the output of `streamMap length` depends upon the length of lists received as input. These properties are only apparent after the `streamWindow` has taken place, so we could not generally move downstream operators to before the window is created.

If we were to consider only the data within a `Stream`, then `streamWindow` followed by `streamExpand` looks like a candidate for *operator elimination*. However, `streamExpand` does not simply reverse the effect of a prior `streamWindow`: the metadata of the `Events` arriving at the `streamWindow` is discarded prior to the `streamExpand`, and so cannot be re-created. As described in Section 4.2.5, the window-maker may also have transformed, duplicated or omitted incoming data.

streamFilter, streamExpand; streamFilterAcc, streamExpand

In these combinations, since `streamExpand` requires a list, we can infer that the input stream to the filter-type operator must be a list. We do not have any further information about the data-type. In order to move the `streamFilter` after `streamExpand`, the data-type within the list must itself also be a list, but we do not know whether that is the case.

streamFilter, streamJoin; streamFilterAcc, streamJoin

Moving the filtering operator would potentially result in events that would have been rejected instead reaching the join operator. The tuples emitted would thus differ from the original pairing.

4.6 Tools and Assurance

As discussed in Section 4.4.2, an incorrect rewrite rule could have serious consequences for the correct operation of a stream-processing program. We wanted to have assurance that the rules were correct. One approach would have been to write out full formal correctness proofs for each rule, which would have been a very time consuming process and still subject to human error. Instead, we leveraged the Haskell tool QuickCheck [12] to provide assurance for each rule.

The QuickCheck user defines *properties* that should hold about expressions in their program. A QuickCheck property is a parameterised boolean expression that should evaluate to `true`. QuickCheck generates a large volume of appropriately-typed test data, and calls the expression for each datum, before reporting whether the expression was `true` for all inputs, or for which inputs it failed.

To provide assurance that our rewrite rules were correct, we translated them into QuickCheck properties. To give an example, consider Rule 1, `streamFilter` fusion. The corresponding QuickCheck property (provided in Listing 4.3) is a function which accepts an arbitrary stream (supplied by QuickCheck during operation). We compare expressions corresponding to the left and right-hand sides of the rewrite rule using boolean equivalence (`==`).

```
prop_filterFilter s = (streamFilter q . streamFilter p) s
                   == (streamFilter (\x -> p x && q x)) s
```

Listing 4.3: QuickCheck property for rewrite Rule 1

The full set of properties corresponding to the rewrite rules we designed pass when applied to QuickCheck. They are provided in Chapter C.

4.6.1 Variables

Variables are frequently used in our rewrite rules to represent the parameters to stream-operators, such as filter predicates, which would be written by the end-user in a stream-processing program. In Rule 1, `p` and `q` are such variables.

When we translate rules into QuickCheck properties, we need to replace variables with concrete expressions so the property can be evaluated. Most of the stream-operator parameters are functions. Although QuickCheck can be used to synthesise functions [11], this is more complicated than synthesising data, and has a number of drawbacks.

Instead, we have written a set of stand-in utility functions that can be used as the parameters to stream operators. Listing 4.4 illustrates examples of predicates for `streamFilter`. The full set of utility functions is provided in Chapter C.

```
p = (>= 'a')
q = (<= 'z')
```

Listing 4.4: Example stand-in parameters for QuickCheck properties

4.6.2 Additional Rules

For rules involving `streamScan` or `streamFilterAcc`, we implemented some additional QuickCheck properties. The extra rules used alternate utility functions as parameters to the stream-processing functions, to provide additional assurance that the rewrite rule was being exercised in as many situations as practical.

4.7 Machine-Assisted Rule Derivation

QuickSpec [70] is a tool for machine-assisted rule discovery within a formal system. We were interested in whether QuickSpec could synthesise rewrite rules that we had not discovered. As an initial goal, we experimented to see if QuickSpec could discover the same rules as we designed in Section 4.5.

4.7.1 Initial operator encoding

QuickSpec requires the user to specify all the functions that it should consider for rule discovery. In our case that means at least the eight functional operators. By describing just the 8 stream-processing operators to QuickSpec, it discovered the two rules depicted in Listing 4.5.

The first rule is a syntactic variation of Rule 10, expressing the commutativity of `streamFilter`. The second is a form of filter elimination that we had not discovered: two filters with the same predicate can be simplified down to one.

```
streamFilter p (streamFilter q xs)
 = streamFilter q (streamFilter p xs)

streamFilter p (streamFilter p xs) = streamFilter p xs
```

Listing 4.5: QuickSpec-discovered filter properties

4.7.2 Anonymous functions

Many of the rewrites rules we designed relied upon anonymous functions on the right-hand side (e.g. Rule 1). QuickSpec cannot synthesise anonymous functions, so it is not capable of discovering rules of this form.

As an experiment we defined a labelled function equivalent to the anonymous function from the right-hand side of Rule 1 (`both p q e = p e && q e`) and provided it to QuickSpec. This was all that was needed for QuickSpec to discover the same rule (Listing 4.6).

```
streamFilter (both p q) xs = streamFilter p (streamFilter q)
```

Listing 4.6: A variation of Rule 1

4.7.3 Composition

Observe that in Listing 4.5, the two operators in each rule are connected together by *function application*. By contrast, in the rest of this chapter we have used *function composition* (`.`).

QuickSpec would not discover rules that use or depend upon composition without being explicitly provided with a definition of composition. Similarly, it will not consider other common combinators, such as `&&` (used by Rule 1), constructs such as `if`/`then`/`else` (Rule 2), or common functions such as `map` (Rule 17), `filter` (Rule 9) or `flip` (Rule 6).

To have any hope of QuickSpec discovering the same rules as we did, or new rules of a similar complexity, we need to add definitions of all such common functions to QuickSpec's set of functions under consideration.

In our early experiments, expanding QuickCheck's universe with a single additional function — composition — greatly increased its computational demands, rapidly outstripping my local resources.

If we supply composition along with only one or two stream-processing operators, QuickCheck discovers rules such as map fusion (Rule 5) and its dual (Listing 4.7).

Limiting the functions described to QuickCheck reduces the chance of finding rules involving several operators or of greater complexity than we have designed with manual pair-wise comparison.

```
streamMap (f (.) g) xs = streamMap f (streamMap g xs)

streamMap f (.) streamMap g = streamMap (f (.) g)
```

Listing 4.7: Map rules discovered in isolation from other operators

4.8 Implementation

We now describe the design and implementation of *StrIoT*'s Logical Optimiser. We first consider the type of functions implementing rewrite rules.

4.8.1 Rewrite Rule Encoding

Both Stream-processing programs and subsets of them are encoded as `StreamGraphs`, as described in Section 3.2.4. A rewrite-rule function requires the sub-program to which it is being applied, thus they will require `StreamGraph` as an input argument.

For a given rewrite rule and a candidate sub-program, we need to determine that the rule is applicable. For example, filter fusion (Rule 1) is applicable only to a pair of `streamFilter` operators. We opted to make this the responsibility of rewrite-rule functions. We signal whether a rule was applicable using Haskell's `Maybe` type, where `Nothing` indicates that the rule was not applicable.

In the event that a rewrite rule is applicable, the function needs to enact the transformation on the stream-processing program. The rewrite-rule's argument is a sub-set of the full program, but the transformation may require changes to other portions of the program due to the nature of the `Graph` type (see Section 3.2.4). Our rewrite-rules produce a transformation function that can be applied to the full program by the calling code.

The final type for our rewrite-rule functions is presented in Listing 4.8 [79, `src/Striot/LogicalOptimiser/RewriteRule.hs#16`] .

```
type RewriteRule = StreamGraph -> Maybe (StreamGraph -> StreamGraph)
```

Listing 4.8: Type definition for rewrite-rule functions

For debugging purposes, it is useful to be able to identify instances of rewrite rules. Functions are not easy to inspect: they cannot, for example, provide an instance of `Show`. We define the type `LabelledRewriteRule` (Listing 4.9) which pairs a `RewriteRule` with a string for this purpose. [79, `src/Striot/LogicalOptimiser/RewriteRule.hs#19`]

```
data LabelledRewriteRule = LabelledRewriteRule
    { ruleLabel :: String
    , rule :: RewriteRule }
```

Listing 4.9: Type definition for labelled rewrite-rules

Implementation

We use Haskell pattern-matching to encode the left-hand side of the rewrite rules. The pattern is written in terms of the relevant constructors of the `Graph` type (`Connect`,`Vertex`) and our own Vertex type (`StreamVertex`).

The function definition for the matching case describes the graph operations that are required to perform the rewrite (e.g. `removeEdge`, `mergeVertices`, `replaceVertex`).

For the scenario when the rule does not match, we use the catch-all pattern _ and the value `Nothing`.

Our implementation of filter fusion [79, `src/Striot/LogicalOptimiser.hs#166`] is illustrated in Listing 4.10.

```
filterFuse :: RewriteRule
filterFuse (Connect
  (Vertex a@(StreamVertex i (Filter sel1) (p:_) _ _ s1))
  (Vertex b@(StreamVertex _ (Filter sel2) (q:_) _ _ s2))) =
    let c = a { operator    = Filter (sel1 * sel2)
              , parameters  = [[| (\p q x -> p x && q x) ($p)(q) |]]
              , serviceRate = sumRates s1 sel1 s2
              }
    in Just (removeEdge c c . mergeVertices (`elem` [a,b]) c)

filterFuse _ = Nothing
```

Listing 4.10: *StrIoT* implementation of filter fusion

4.8.2 Applying Rewrite Rules

The first step in applying rewrite rules to a candidate program is to traverse the program to find a point at which a given rule is applicable. Our function `firstMatch` [79, `src/Striot/LogicalOptimiser.hs#87`] is provided in Listing 4.11.

It's possible that the rule is not applicable to any point of the program, which we again indicate with the `Maybe` type.

```
firstMatch :: StreamGraph -> RewriteRule
           -> Maybe (StreamGraph -> StreamGraph)
firstMatch g r = case r g of
        Just f -> Just f
        _      -> case g of
            Empty       -> Nothing
            Vertex _    -> Nothing
            Overlay a b -> tryLeftThenRight a b r
            Connect a b -> tryLeftThenRight a b r
        where
            tryLeftThenRight a b r = case firstMatch a r of
                                Just f  -> Just f
                                Nothing -> firstMatch b r
```

Listing 4.11: Implementation of `firstMatch`

During development, it proved useful to be able to determine the provenance of a rewritten program: what rules were applied in order to derive it? To capture this information, we define a data-type `Variant` [79, `src/Striot/LogicalOptimiser.hs#67`] which collects together a rewritten program with its parent program and the last rule which was applied. This type is provided in Listing 4.12.

```
data Variant = Variant { variantGraph  :: StreamGraph
                       , variantRule   :: String
                       , variantParent :: Variant
                       }
             | Original { variantGraph :: StreamGraph }
               deriving (Show, Eq)
```

Listing 4.12: The `Variant` data-type

We now combined all the pieces to write the principal routine. Given a list of rewrite rules and a stream-processing program, we attempt to apply every rule to the program individually to build a list of matching rewrite rules, which we in turn apply to get a list of variant programs. We then recursively apply the function again to each of those variants.

A set of rewrite rules may produce a variant program that is equivalent to one of its ancestors. We need to ensure that the recursive application eventually terminates. We opted to apply a depth limit to recursion: each invocation tests the supplied limit and terminates if the value is less than one. We decrement the limit for subsequent recursive calls.

Our implementation [79, `src/Striot/LogicalOptimiser.hs#106`] is in Listing 4.13.

```
applyRules :: [LabelledRewriteRule] -> Int -> Variant -> [Variant]
applyRules lrules n v =
    if n < 1 then [v]
    else let
        sg = variantGraph v
        vs = map (\(l,f) -> Variant (f sg) l v)
           $ mapMaybe (\(LabelledRewriteRule l r) ->
                          fmap ((,) l) (firstMatch sg r))
             lrules
        in v : vs ++ (concatMap (applyRules lrules (n-1)) vs)
```

Listing 4.13: `applyRules` function to apply rewrite rules

The list of variant programs may include duplicates, derived by a different set of rules, or differing order of application. These can be identified and removed with standard Haskell functions, e.g. `nubBy (on (==) variantGraph)`. See Chapter B for a description of these functions.

4.9 Chapter Summary

There are 64 pairings of the eight stream-processing operators. By performing a systematic analysis of these pairings, we created 22 rewrite rules that could be used in a term-rewriting system for transforming programs without altering their semantics.

The process also produced a series of rules which do alter the semantics of programs but might nonetheless be interesting or useful: 5 which caused re-ordering of the stream events and 2 which caused reshaping of aggregate windows (collections of data) within events.

Table 4.2 summarizes the total number of rules we designed, grouped by their category of stream-processing optimisation [38].

Optimisation Category	Number of Rules
fusion	7
elimination	1
operator re-ordering	11
other semantically preserving	1
type-constrained	2
total semantically-preserving	22
stream re-ordering	5
window re-shaping	2
grand total	29

Table 4.2: Summary of rewrite rules, grouped by optimisation category

We considered all 64 combinations of pairs of operators during this exercise. However our intention was not to exhaustively determine every possible transformation. There may exist further semantically-preserving transformations for pairs of operators.

We did not attempt to devise rules that applied to larger expressions involving more than two operators. However, the rules we have created could be repeatedly applied to a larger stream-processing program. For example, a program featuring a series of `streamFilter` operators could be reduced down to one by repeatedly applying Rule 1.

We performed an initial exploration of tools to automatically derive rewrite rules (Section 4.7). We were able to derive several of the rules we had created manually in our initial experiments as well as one new rule. Whilst promising, this work was not on the critical path for our research and so we did not pursue it further.

By leveraging purely-functional semantics, we have demonstrated a unique advantage of purely-functional programming for the design and implementation of this aspect of the declarative architecture.

Chapter 5. Cost Models

The Evaluator component of the declarative architecture (Section 2.4) is responsible for calculating a score for each of a list of possible deployment plans, with reference to relevant non-functional requirements, and choosing the best-scoring plan for deployment.

In this chapter we first explore some of the non-functional requirements which are relevant to stream-processing systems and summarize some of the approaches that could be used in the design of the Evaluator in order to compare deployment plans.

We then describe in detail the application of Queueing Theory to model distributed stream-processing systems and calculate estimates for properties to be used in costing.

5.1 Non-Functional Requirements

A distributed stream-processing program may be subject to several non-functional requirements at once: the overall financial cost of operating the system; real-time constraints on the time taken to produce results; limitations on the quantity or complexity of computation that can take place on particular deployment nodes; limitations on the bandwidth available for transmitting data through the system. Some of these constraints may interact with each other: for example the quantity and specification of rented Cloud nodes impacting both the speed and complexity of processing that can take place, as well as the operational cost.

In designing *StrIoT* we wished to support the use of multiple criteria for evaluating deployment plans. For the initial implementation, and in order to narrow the scope to make evaluation practical, we opted to start with two: utilisation and bandwidth.

5.1.1 Energy usage

Michalák et al modelled energy use with their cost model [58]. As part of their work they carefully measured the energy consumption of different functions of a specific "smart watch" device. The shelf-life of modern smart appliances can be very short. Their chosen device was discontinued shortly after their research.

The work of this thesis was completed part-time. This extended time span meant an increased exposure to the risk of given devices becoming discontinued or unavailable. We opted to avoid depending upon a non-functional requirement that exposed us to this risk.

5.1.2 Performance

We could model performance of a stream-processing program, such as operator throughput. This would complement a Logical Optimiser which functioned similarly to a classic source code compiler, replacing algorithms with functionally-equivalent but better-performing alternatives, e.g. by reducing the number of operators.

Performance is a popular metric for stream-processing systems and industrial-strength stream-processing systems have received a lot of attention and optimisation to improve performance. We therefore felt it would be very difficult to demonstrate a performance advantage with a research prototype and it would be better to focus on other metrics.

5.1.3 Utilisation

We define utilisation as the ratio of the arrival rate of events to a system over the rate at which the system can service events: $\rho = \frac{\lambda}{\mu}$. When the arrival rate is larger than the service rate, a queue of events to be processed will grow. If the arrival rate is consistently larger, the system will never complete processing and is unviable.

It is very hard to assess, ahead of time, the exact load requirements of a distributed stream-processing system. Since the consequence of under-provisioning is an unviable system, practitioners tend to over-provision to avoid this risk [68].

There is a relationship between the service time of a system and cost: a more capable and consequently expensive system may be able to process a higher rate of incoming events. To manage cost, it is therefore desirable to not over-provision a deployment.

It would be valuable for an Evaluator to accurately assess the utilisation of a system in order to choose plans which best avoided both under and over-provisioning.

There are drawbacks to optimising in terms of utilisation. It is a one-dimensional metric, whereas — for some problems — there can be several relevant and inter-acting parameters.

For example, a system which is under-provisioned in terms of memory (RAM) may result in the operating system swapping memory pages to and from disk, resulting in an increase in CPU and IO utilisation. The most effective change could be to increase the available memory, but an overall utilisation metric does not directly point towards that solution.

Similarly, poorly-written code, such as an unnecessarily busy loop, can result in high CPU utilisation. Increasing the specification of the available CPU will not resolve the root problem: a busy loop will keep a high specification CPU just as busy.

5.1.4 Bandwidth

In a distributed stream-processing system, data must be transmitted from its source (e.g. sensors) through the deployment nodes which perform data processing (including field gateways, smart phones, and cloud computing instances).

There may be different costs for the transmission of data through this heterogeneous system.

Different aspects of data transmission may involve costs which the user may wish to minimise.

A field gateway device may connect onwards to the public Internet using a mobile broadband technology (4G, 5G) etc. This would require a modem and a contract with a mobile broadband provider. Consumer mobile broadband contracts are typically metered, with either a capped data allowance within a period (e.g monthly) or a metered data rate. Depending on the availability of mobile broadband connectivity, there will be physical limits on the total available bandwidth.

Inter-cloud data transfers can be at cost. Cloud providers may charge different rates for data ingress or egress to their systems from the public Internet, between components within their Cloud (such as between Regions or Services).

The approach to serialising data for transmission can have an impact on cost. Establishing a TCP/IP connection requires some overhead. Maintaining and re-using an established connection results in lower overheads, but other constraints (such as power/energy/battery life) may prevent this.

Noisy or lossy links may result in lost or corrupt data packets, which may require re-transmission.

5.2 Initial Approach

In the early stages of this work, we implemented a simple place-holder cost model that ranked plans on their quantity of operators. This ranking favoured program transformations that reduced the number of operators in a program, such as fusion (Section 4.5.1).

McSherry et al [55] observed that the overheads of inter-node communication introduced by parallel architectures are often overlooked when assessing systems. By optimising for a reduced quantity of operators, plans preferred by the Evaluator will tend towards fewer nodes, with a corresponding lower overhead for inter-node communication.

With functional programming it is common to pass lists of data between functions which operate on the items within the list (such as filtering with `filter` and transforming with `map`). As discussed in Section 2.2.2, the repeated de-construction and re-construction of enclosing list structures can be very inefficient. Reducing the number of operators also reduces instances where intermediate lists would be

de-constructed and re-constructed.

We discuss operator fusion within a distributed environment further in Section 6.3.1.

5.3 Queueing Theory

In order to cost plans on other non-functional requirements, we explored modelling stream-processing programs using Queueing Theory: the study of systems which feature a series of jobs which are serviced by one or more servers, such that jobs may have to wait in a queue before being processed [59]. Queueing theory has been studied extensively since the 1950s and queueing models have proven useful for predicting the behaviour of distributed systems, including stream-processing systems [29].

5.3.1 Queueing Systems

A queueing system is a model consisting of jobs (possibly of different *types*) that arrive, are processed by one or more servers, and leave the system.

Queueing systems are often categorised according to a short-hand scheme which summarizes some of their key properties as a triplet [45]: a categorisation of the arrival rate, followed by the service time, both described as one of M (Markovian), D (deterministic) or G (generalised); followed by the number of parallel servers (e.g. 1).

For example the designation *M/M/1* denotes a queueing system with a memoryless arrival process (Poisson), a memoryless service time (exponential), and a single server processing jobs.

A useful law applying to queueing systems is the *utilisation law*, which states "*the average number of servers busy with jobs of type i is equal to the offered load of type i*"

For the simplified case of 1 server and a single job type, "*the average busyness of the server is equal to the total average demand for service per unit time.*"

The Utilisation Law $\rho = \frac{\lambda}{\mu}$ allows us to determine the utilisation of a server from the ratio of job arrivals and the average rate at which jobs are serviced. The derivation of the Utilisation Law relies upon Little's theorem [52], the full details of which we elide here.

Little's Theorem holds when the system is in *steady-state*: informally, when the probability of observing any particular state of the system is no longer a function of time.

This implies the following properties:

1. the arrival rate into the system is the same as the departure rate
2. the average number of jobs in the system (either being served or waiting to be) is static
3. the average interval between arrival and departure is static

5.3.2 Queueing Networks

Queueing networks are composed of inter-connected queueing systems.

A queueing network can be open, where jobs arrive enter and leave the system, or closed, where jobs solely circulate internal to the network.

Open queueing networks are often referred to as Jackson Networks, due to a popular theorem [42].

A queueing network is formally described by the average arrival rate of jobs into the system and a routing matrix which describes the probabilities of jobs moving from one server to another upon completion (or leaving the network).

Jackson's theorem states that there is a closed-form solution to the traffic equations describing the mean arrival rate of jobs (both internal and external) at each server. From this result, many useful properties of the network, such as average sojourn time or queue length, can be determined.

Jackson's theorem depends upon some properties of the network holding: external arrival rates must have a Poisson distribution and the distribution of service rates for all servers must be exponential. For many real systems, these requirements may not be met, or it may be impractical to determine whether or not they hold [14].

5.4 Alternative Approaches

Queueing theory models are relatively simple to construct and cheap to evaluate. However, as mentioned above, the models are only valid if their preconditions (listed in Sections 5.3.1 and 5.3.2) are met, and this may be difficult or impossible to establish.

Another approach is to evaluate and measure simulations of the final deployment. These could involve executing the real stream-processing program, with real inputs substituted for synthetic alternatives, and the deployment environment approximated or simulated, for example running all the nodes on the user's workstation rather than distributed across the Cloud.

An advantage of such an approach would be the ability to measure the true performance of running operators, which has been shown to be valuable for managing run-time performance [44]. A drawback is this is much more complicated. The model must approximate the real-world deployment sufficiently for the properties being

measured to be accurate. The properties of concern differ for each application. For example synthetic input data may need to match the arrival rate or distribution of the true inputs, depending on exactly how it is processed.

For the work in this thesis, we have chosen to explore queueing theory and its suitability for this purpose.

5.5 Mapping Queueing Theory to StrIoT

We model *StrIoT* operators as individual servers in a queueing network. For every operator, we need to know a mean average of its service time (μ). As the model is considered to be in steady state, the rate that most operators emit events is not influenced by the service time, and matches the arrival rate (λ). We note where this does not hold in the following sections.

For the majority of operators, there is exactly one incoming stream of events.

Figure 5.1 illustrates these properties.

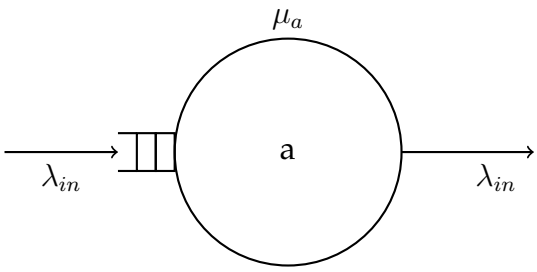


Figure 5.1: Queueing network representation of a stream operator.

This representation is sufficient to model `streamMap` and `streamScan`. In the following sections we describe additions and variations to this representation required for the remaining operators.

5.5.1 *streamMerge*

`streamMerge` reads from multiple incoming streams in turn and emits the events it receives. Its output rate is simply the sum of the arrival rates of its inputs. This is illustrated in Figure 5.2.

In queueing theory terms, there is nothing unusual about `streamMerge`: it can be modelled as an ordinary server.

5.5.2 *streamFilter*, *streamFilterAcc*

StrIoT's filtering operators reject a proportion of incoming events. To calculate the emission rate we need to know the average filter selectivity, which we denote f, and assign a probability value in the range $0-1$ that an event is accepted. The emission rate is then determined by the product $f \cdot \lambda_{in}$. These properties are depicted in Figure 5.3.

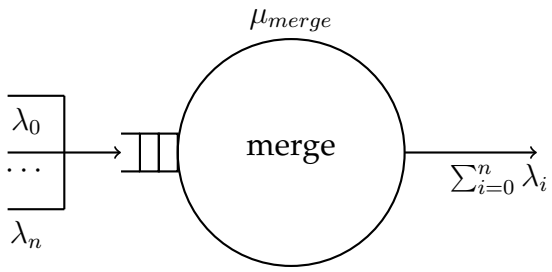


Figure 5.2: Queueing network representation of `streamMerge`

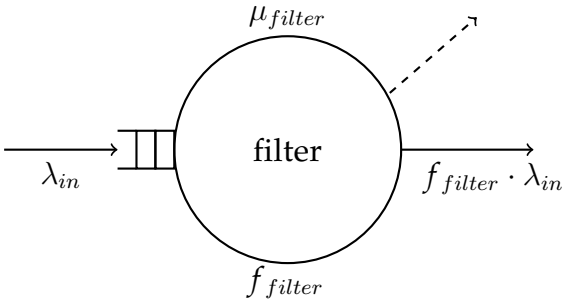


Figure 5.3: Queueing network representation of `streamFilter` and `streamFilterAcc`.
The dotted line represents rejected events leaving the system.

5.5.3 *streamWindow*

The exact behaviour of the `streamWindow` operator depends upon the semantics of the user-defined window-maker and so we cannot reason about the operator independently.

StrIoT provides four preset window-makers to perform common aggregations, described in Section 2.5.2. We now analyse their behaviour from a queueing theory perspective.

5.5.4 window-maker *chop*

The `chop` window-maker aggregates lists of events of a fixed, defined length. The output rate from this operator is therefore a fixed function of the input rate: If the operator is configured to collect together lists of 5 events, and one event arrives every second, then the operator will emit a list of 5 events every 5 seconds.

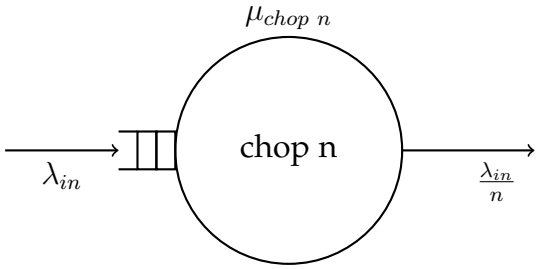


Figure 5.4: Queueing network representation of `streamWindow chop`.
window-maker chop with a parameter size n will emit events at a rate of $\frac{\lambda_{in}}{n}$.

5.5.5 window-maker *sliding*

The `sliding` window-maker emits overlapping lists of a fixed size. For example, for a list size of 2, the input events $1, 2, 3, 4, 5$ would result in the output lists $[1, 2], [2, 3], [3, 4], [4, 5]$.

The initial list emission would be delayed until sufficient input events had been received. However, once the system reaches steady-state, each arriving event would trigger a corresponding output event, thus, $\lambda_{in} = \lambda_{out}$. This is depicted in Figure 5.5.

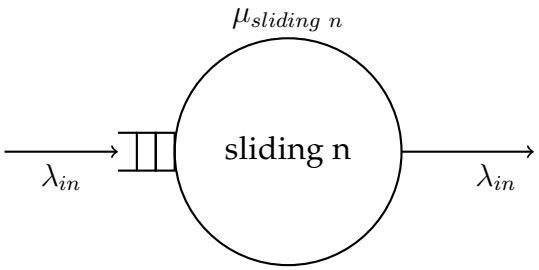


Figure 5.5: Queueing network representation of `streamWindow sliding`. Events are emitted at the same rate they are received.

5.5.6 window-maker *chopTime*

StrIoT supports `Events` without a timestamp. The `chopTime` window-makers filter such events. For the analysis which follows, we assume all events under consideration have timestamps.

The `chopTime` window-maker outputs lists of events where the included `Event`'s timestamps fall within a fixed interval.

When the first event arrives at the operator, `chopTime` calculates the upper-bound timestamp for events to include in the current batch. When an event arrives featuring a timestamp later than the current upper-bound, `chopTime` emits the current batch, starts a new batch containing the latest event, and calculates the next upper-bound from that event's timestamp.

If no subsequent `Event` arrives beyond the upper-bound, `chopTime` won't emit further `Events`. In practise, the programmer can prevent this outcome from occurring by defining a stream source which produces a steady stream of timed events and merging this source into the main stream.

The output rate is $\frac{\lambda}{n}$ (Figure 5.6).

5.5.7 window-maker *slidingTime*

`slidingTime` emits overlapping lists of events within a fixed time interval. `slidingTime` is depicted in Figure 5.7.

Similar to `chopTime`, `slidingTime` filters out received events without a time-stamp. For the analysis which follows, we assume all events under consideration have times-tamps.

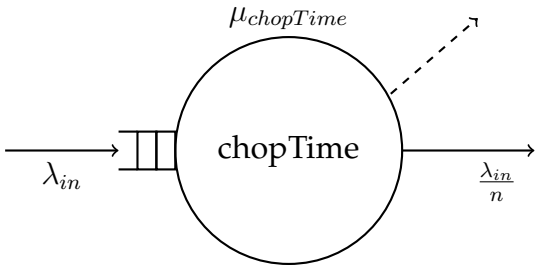


Figure 5.6: Queueing network representation of `streamWindow` `chopTime`. The dotted line represents `Events` lacking a timestamp rejected by `chopTime`.

Consider a series of events arriving at intervals after a common starting time. Here we represent the events solely as their arrival interval: $0, 1, 2, 3, 4, 5, 6$. `slidingTime` configured with a size of 2 emits $[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]$.

Each output list is emitted once the current time interval being collected closes. This is determined when the first event arrives *after* the interval.

In the above example, the event $[2, 3]$ cannot be emitted until the event 4 arrives, indicating that no further events in the interval $2 - 4$ will occur. Without any further inputs, the event $[5, 6]$ will never be emitted.

Unlike `sliding` (described above), under steady-state operation, `slidingTime` is not guaranteed to emit an event for each arrival.

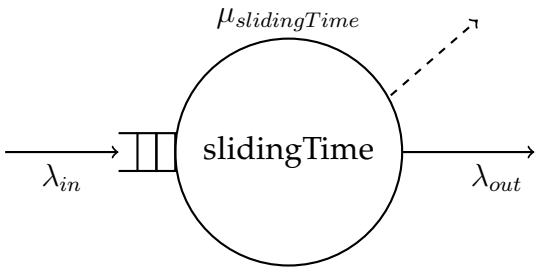


Figure 5.7: Queueing network representation of `streamWindow` `slidingTime`. The dotted line represents rejecting arriving `Events` lacking a timestamp.

5.5.8 *streamExpand*

`streamExpand` receives `Events` consisting of lists and emits an `Event` for each item in the list. If the received `Event` list is empty, `streamExpand` doesn't emit anything. The average output rate of `streamExpand` is the product of the average arrival rate and the average length of lists within the arriving `Events`. This is depicted in Figure 5.8.

5.5.9 *streamJoin*

`streamJoin` reads events from two input streams in turn and emits them as pairs. Since `streamJoin` must wait for an event on both input streams before it can emit the pair, its output rate matches that of its slowest input.

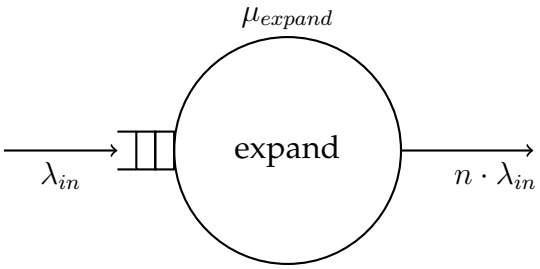


Figure 5.8: Queueing theory representation of `streamExpand`. n is the average length of lists in the arriving `Events`.

However, if the input rates are not the same, the queue of unprocessed events arriving from the faster stream will grow indefinitely. In practice this would not be viable, thus `streamJoin` could only be used in scenarios where the average arrival rates for the input streams were balanced (Figure 5.9).

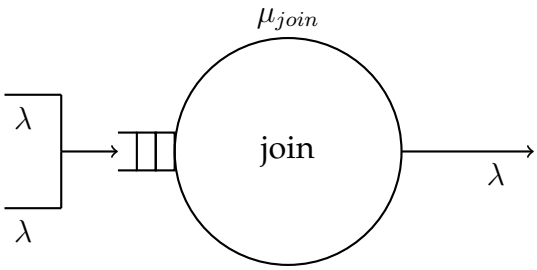


Figure 5.9: Queueing network representation of `streamJoin`

5.6 The Impact of Rewrite Rules on the Model

The rewrite rules introduced in Chapter 4 have implications for the queueing theory model of the stream-processing program. In this section we collect some observations about certain rules and groups of rules and how they affect the modelling.

5.6.1 Filtering and Mapping Fusion

The 7 rules which fuse together pairs of filtering and mapping operators replace the two original operators with one new operator. The rule needs to determine the queueing theory properties of the new operator.

In Section 5.2 above, we noted the potential performance impact of reducing the number of stream-processing operators by reducing the corresponding amount of "book-keeping" work required to marshal data between them.

A reduction in the work required could be reflected in a reduction in the average time required to service events. However, in our model we have defined the service time for operators to be the *internal* time required by the operator, i.e, the work performed by the user-supplied parameters to the operator. In contrast, the list construction and de-construction work is external, and so not reflected in the service time values for operators.

Since fusing operators has no obvious effect on the total internal service time, we define the service time of the new operator to be the sum of the service times of the original operators.

Consequently, fusion rules do not reduce the overall time required to service events in the system, nor the overall system utilisation.

In the case of fusion rules which operate on filtering operators, the rule also needs to specify the selectivity of the new operator. We chose to define this as the product of the original two selectivities.

For example, if two adjacent filter operators had selectivities of $\frac{9}{10}$ and $\frac{1}{2}$ respectively, filter fusion would result in a new filter operator with a selectivity of $\frac{45}{100}$.

5.6.2 Swapping Operators

The rules 11 and 12 reverse the order of adjacent map and filtering operators. For each event, the predicate of the newly-positioned filtering operator performs the work of the original predicate and `streamMap`'s higher-order function. Therefore, we defined the service time of the newly-positioned filter operator to be the sum of the original two operators.

These rules result in the sum of service times across the operators increasing.

5.6.3 Moving Filters Upstream

The four rules 9, 13, W1 and W2 each move a filtering operation upstream, such that it operates on a list of `Events`. To achieve this, it is replaced with a `streamMap`, lifting the filter predicate to operate across lists.

Since we have encoded filter selectivity as a property of the `Filter` constructor of the `StreamOperator` type (described in Section 5.7.1 below), by replacing it with `streamMap`, we lose the ability to encode the selectivity.

For filtering operators, selectivity influences the departure rate for events. For the new mapping operator, the departure rate matches the arrival rate, as per the discussion in Section 5.5: for every arriving list, a list will be emitted, even if all the enclosed events are rejected by the predicate.

The missing selectivity information would pose a problem if the lists are later unpacked by another operator such as `streamExpand`. As noted in Section 5.5.8, the average output rate of `streamExpand` is dependent on the average length of arriving lists.

5.6.4 Lifted Operators and Service Times

Several rewrite rules (those in the previous section, as well as Rule 21, Rule 18 and Rule 17) result in the ”lifting” of work performed per-item into lists of items.

Choosing a service time for the new operator is difficult. The service time for the original operator is defined as per arriving event. The aggregate operator will perform

this work for each item within the arriving lists. For steady-state, we could define it as the product of the cost per item and the average length of arriving lists. However, the rewrite rule does not have this information.

5.6.5 Creating New Filters

Several rules create new filter operators, for which we need to determine appropriate selectivities.

Rule 18 introduces a new filter to remove empty lists prior to `streamScan`. This is necessary to preserve the semantics of the operators before and after rewriting: before, the `streamExpand` performed this work.

In the general case, with real-world stream-processing programs, we believe it is relatively unlikely for `streamExpand` to receive empty lists. It's unlikely for an application programmer to wish to generate empty lists with windowing operations. However it is not impossible: as described above in Section 5.5.6, the `chopTime` window-maker will emit a series of empty lists corresponding to windowed time intervals if the interval between received events exceeds its window size. For the rewrite rule, we have initially opted to pick an arbitrary selectivity of 50%.

For the service time, we opted to pick a value of 0, since it is replicating the previously external work of the `streamExpand` operator, and we have defined the service time to reflect the internal work performed by operators.

Rule R3 removes a `streamFilter` operator originally downstream from a `streamMerge` and creates copies of it for each incoming stream. For the rewrite rule implementation, we opted to choose the same service time and selectivity values for the new filters as the original.

The dual rule Rule R1 removes a set of `streamFilter` operators on the incoming streams to a `streamMerge` and inserts a new filter immediately afterwards. We opted to define this rule such that it would only match if the original filters had identical predicates and selectivities. The new filter was created with the same values.

If we had permitted the rule to match filters with varying selectivities, we would need to pick an appropriate selectivity for the new operator. One choice would be an average of the original selectivities.

5.7 Implementation

We now describe the approach taken to implement the two cost models within *StrIoT*.

5.7.1 Utilisation

To support building a queueing theory model for evaluation, we needed to extend *StrIoT* with the properties required by the model (described in Section 5.5).

We considered extending the `GenerateOpts` data-type (See Section 3.3.5) which is already used to describe properties of the program not captured by the `StreamGraph` data-type, for example which rewrite rules for the Logical Optimiser to consider. However, in order to match the queueing theory properties back to operators in the stream-processing program, we would need to encode them with a form of reference, such as to the `StreamOperator vertexId` value. We felt that this would be unwieldy and risk introducing errors: if the stream-processing program was altered, care would be needed to ensure the references were updated accordingly.

To remove this risk, we opted to extend the definition of the `StreamGraph` data-types (described in Section 3.2.4) with the required data-types directly. Listing 5.1 illustrates the parameter added to `StreamVertex` and Listing 5.2 to `StreamOperator`, respectively.

```
data StreamVertex = StreamVertex
  { vertexId      :: Int
  , operator      :: StreamOperator
  , parameters    :: [ExpQ]
  , intype        :: String
  , outtype       :: String
{+, serviceRate :: Double+}
  }
```

Listing 5.1: `StreamVertex` type extended with `serviceRate` property

```
data StreamOperator = Map
                    | Filter {+Double+}
                    | Expand
                    | Window
                    | Merge
                    | Join
                    | Scan
                    | FilterAcc {+Double+}
                    | Source {+Double+}
                    | Sink
    deriving (Show,Ord,Eq)
```

Listing 5.2: `StreamOperator` type extended with filter selectivities and source arrival rates

Aggregate utilisation

Modelling utilisation solely at the level of operators does not allow the user to model deployment nodes being overwhelmed by having a large set of operators assigned to them in a deployment plan. The individual operators may not be over-utilised, but the aggregate load of a set could be beyond what a particular deployment node could support.

To address this, In addition to filtering operators which are determined to be over-utilised, we also implemented support for a user-specified aggregate utilisation limit, to be applied for each node in a deployment.

This does not take into account any specific properties of a given deployment node. For example we cannot differentiate between a low powered Edge device in the field, and a well-specified cloud instance. As discussed in Section 3.1.1, our model of nodes is homogeneous.

To configure the Evaluator for this scenario, the user can provide a per-node limit on the sum of operator utilisations that can be accepted. To choose a sensible value for this limit, the user should consider the least-capable node in their deployment, for example an edge device, and either estimate or perform some measurements of the performance of that node with an appropriate workload.

Future work could develop a Run-time Monitor to measure the performance of individual nodes in a deployment and determine reasonable threshold values. This is discussed further in Section 7.3.3.

5.7.2 Bandwidth

Bandwidth is a measure of data over time, typically measured in bits (or kilobits, megabits, etc.) per second.

The queueing theory model provides us with the rate of events arriving at each operator within the stream-processing program. In order to estimate the bandwidth between operators, we need to combine this with an estimate of the size of individual `Events`.

Our implementation of the *StrIoT* runtime includes routines for serialising and de-serialising `Event` streams for transmission over a TCP/IP network (Described in [8, Section 3.3.2]).

To measure the size of individual `Events`, we use the same method as the Deployer to serialize an instance of a given `Event` type (e.g. `Event _ Int`) and then count the number of bytes used in the representation.

In order to account for the cost of the overhead of TCP/IP transmission, our bandwidth model also applies a per-event weighting to the calculated bandwidth. This is a per-event overhead to represent the size of typical TCP and IP packet headers.

The Evaluator takes a user-supplied bandwidth limit and compares it to the calculated estimated bandwidth at the link between the first and second nodes in a deployment plan. In an Edge-to-Cloud configuration, this link would correspond to the uplink connecting Edge devices to a gateway device.

5.7.3 Applying Cost Models

We define `Cost` as `Maybe Int`. The `Maybe` component models whether the plan being evaluated is outright rejected. The `Int` component allows for ranking accepted plans.

The function `planCost` [79, `src/Striot/Orchestration.hs#120`] is responsible for evaluating plans. This function is provided in Listing 5.3.

We construct a Queueing Theory model corresponding to the stream-processing program, via `calcAllSg`. This model is used for the utilisation-based checks that follow. `planCost` performs three checks to accept or reject the plan. The first test is whether any individual operator is determined to be over-utilised (Section 5.5). This is performed by `isOverUtilised`. The second test calculates the sum of operator utilisations for nodes in the deployment plan. The plan will be rejected if any node has a sum of operator utilisations exceeding a user-supplied threshold. (See Section 5.7.1 for advice on choosing an appropriate threshold). The third test, performed by `overBandwidthLimit`, calculates the weighted bandwidth required between the first and second nodes in the deployment plan (Section 5.7.2) and rejects the plan if this is above the user-supplied threshold.

```
planCost :: GenerateOpts -> Plan -> Cost
planCost opts plan@(Plan sg pm) = let
    oi = calcAllSg sg
    in if   isOverUtilised oi
         || any (> maxNodeUtil opts) (totalNodeUtilisations oi pm)
         || overBandwidthLimit plan (maxBandwidth opts)
       then Nothing
       else Just (length pm)
```

Listing 5.3: Implementation of `planCost`

If the plan survives these three tests, we return the number of nodes in the partition plan as the score. This results in plans requiring fewer nodes to have a lower cost than plans requiring more nodes.

5.7.4 Advice for Practitioners

The queueing theory model predicts the behaviour of the program under steady state operation. The model's parameters — rate of arrival into the system; service time per event; filter selectivities — are expressed as averages. During operation, the actual arrival rate may exceed the model's average for intervals, so long the average rate remains accurate. If the model's parameters do not reflect reality, then we cannot extrapolate meaningfully from the model.

Since the consequences of over-utilisation — an unviable deployment, with queues of events building up and never being processed — are more serious than those of under-utilisation (wasted resources), the practitioner is advised to provide conservative estimates for the model parameters: over-estimating arrival rates and under-estimating service rates.

5.8 Limitations

Our approach to cost modelling and our translation of those models into *StrIoT* suffer from a number of limitations.

5.8.1 Utilisation

The theory underpinning our queueing theory model relies upon preconditions about the distribution of arrival rates and service times (Section 5.3) that might not hold [14].

The transformations performed by the Logical Optimiser can result in losing useful information for the model, such as when a `streamFilter` is transformed into a `streamMap` (Section 5.6.3).

These problems could be addressed by development of the Run-time Monitor component of the distributed stream-processing architecture. Run-time measurements of arrival rates, filter selectivities and operator service times could be used to adjust model parameters.

As noted in Section 5.5.3, the flexible semantics of `streamWindow` make modelling it in the general case very difficult. We have provided analysis of its behaviour with the window-makers provided with *StrIoT*, subject to further pre-conditions about the data (presence or absence of time-stamps). There are open questions to resolve about the impact of aggregation on the distribution of emitted events.

We derive the model from the description of the stream-processing program and not from the combined program and mapping of program's operators to deployment nodes (a plan). This means we can detect when an individual operator is likely over-utilised, but not the situation where a deployment node is over-utilised by having too much work assigned to it.

5.8.2 Bandwidth

The approach used to estimate bandwidth (Section 5.7.2) has a number of technical limitations.

We do not have enough information to know whether the arriving `Events` include timestamps, which can have a significant influence on their size. For example the size of an `Event` consisting of a data payload of an Integer (`Int`) is 26 bytes without a timestamp, and 44 bytes (69% larger) when a timestamp is present. For our implementation we assume that all `Events` include timestamps.

In general we are unable to estimate the size of `Events` containing variable length data, such as lists. We do not have enough information to measure or estimate the average length of the arriving lists. We have implemented one exception: for `streamWindow` using the `chopTime` window-maker supplied with *StrIoT*, we are able to calculate the average output list length by considering the supplied window size and the average arrival rate.

There are a number of future enhancements which could help to address this limitation: an implementation of the Run-time Monitor would provide an opportunity to estimate list lengths based on past data; adjusting the data types (similar to adding queueing theory properties in Section 5.7.1) to allow for the user to provide `Event` size estimates for some or all operators, and for list-length estimate information of other operators to be derived from known values earlier in the stream.

Our initial implementation makes the assumption that the link between the first and second nodes, corresponding to the interface between Edge and Gateway devices, is the most bandwidth-constrained and where we should apply the bandwidth model. Future work could enhance the bandwidth element of our cost model to consider other links, and to apply different bandwidth thresholds to separate links. This could follow from developing the catalogue component of the declarative stream-processing model.

5.9 Chapter Summary

In this chapter we have explored some of the non-functional requirements which are of key concern in modern problem domains, in particular node utilisation and bandwidth which we have chosen as the focus of our initial implementation.

We outlined approaches that can be used for build an Evaluator. We have explored queueing theory in more detail and explained why we have chosen this approach for cost modelling.

We have discussed how we map queueing theory concepts on the architecture of *StrIoT* and performed an analysis of how some of the design choices for our system, namely the choice of data-types and design of the Logical Optimiser, impact the model.

Chapter 6. Evaluation

In this chapter we explore the design and performance of *StrIoT*.

We began this thesis with a proposal for a declarative stream-processing architecture, outlined in Section 2.4, from which we have focussed on three components: Deployer (Chapter 3), the Logical Optimiser (Chapter 4) and the Evaluator (Chapter 5). In the second part of this chapter we explore these specific components in more detail.

6.1 Methodology and Goals

Recapping our original research aims from Section 1.4: to provide further evidence as to the viability of the declarative stream-processing architectural model (RQ1) and purely-functional programming for building a system of this complexity (RQ2); to identify advantages and disadvantages of purely-functional programming for this domain (RQ3).

Completing the development of *StrIoT* required the design of the Logical Optimiser (Chapter 4) and Evaluator (Chapter 5). In addition to the objectives above, we will evaluate these specific components.

6.1.1 Methodological approach

Our methodological approach is to first encode existing, real-world stream-processing programs from the literature as stream-processing programs in *StrIoT*. These are discussed in Section 6.2.1 and Section 6.2.2, below.

Achieving this will provide evidence to support RQ1 and RQ2. For RQ3, we must compare the behaviour of pure purely-functional programming implementation with a prior implementation, which we do in Section 6.6.

To evaluate the Logical Optimiser, we first discuss the effects of applying our catalogue of rewrite rules in Section 6.3, before exploring the individual cost models. For utilisation, firstly at the level of logical operators in Section 6.4, and then at the plan level in Section 6.5. We introduce and evaluate the performance of the bandwidth cost model in Section 6.6.

6.2 Functional Stream-Processing

Michalák et al demonstrated the viability of the declarative stream-processing model (Section 2.4) by implementing solutions to several real-world problems [56], [57], [58].

Their approach used the relational algebra as the method of declaratively describing the computation.

In contrast, we have implemented a declarative-stream processing system, *StrIoT*, where stream-processing is described using purely-functional programming. With *StrIoT*, a user describes the computation to take place using Haskell and library functions provided by *StrIoT*, which itself is written with Haskell.

By completing *StrIoT* and using it to implement several solutions to real-world stream-processing problems, we have further demonstrated the viability of the declarative architecture, and shown that purely-functional programming is suitable for this purpose.

We have published the source code to *StrIoT*, including these example programs, as open-source (See Section A.1) We now explore two of those example programs in detail.

6.2.1 DEBS 2015 Grand Challenge

Since 2010, the annual ACM International Conference on Distributed and Event-based Systems (DEBS) has featured a "Grand Challenge": a practical problem for which participants can submit solutions. The winners of the challenge are announced during the conference.

The Grand Challenge for 2015 was based on a scenario of trip information for Taxi journeys in New York City [43]. The challenge posed two problems, the first of which was to identify the 10 most frequently-repeated routes completed within the last 30 minutes of data. A solution should continually update the top 10 as more data is received.

We implemented a solution for this challenge using *StrIoT* [79, `examples/taxi`] , illustrated in flowchart-format in Figure 6.1. Each operator in the flowchart is labelled with a number. We now describe this solution step-by-step, referencing each operator by label.

Stream Source and Event Times

In a real-world scenario, individual taxis submit live trip data at or near to the time of the corresponding events. In our implementation, a single stream source (operator 1) simulates the fleet of 10,867 taxis by reading aggregated trip data from an on-disk file. Details of the format of the data-set and how to obtain it are provided in Section A.2.

The implementation of the source function is simple: we read saved records of historic events sequentially and emit a corresponding `Trip` data structure (described in Table 6.1). This simplicity enables a straightforward implementation that is easy to reason about and can be executed on a single host. Unfortunately this simplification introduces a discrepancy from the real-world. The `nodeSource` function provided by the run-time (described in Section 2.5.3) emits events with a `timestamp` field corresponding

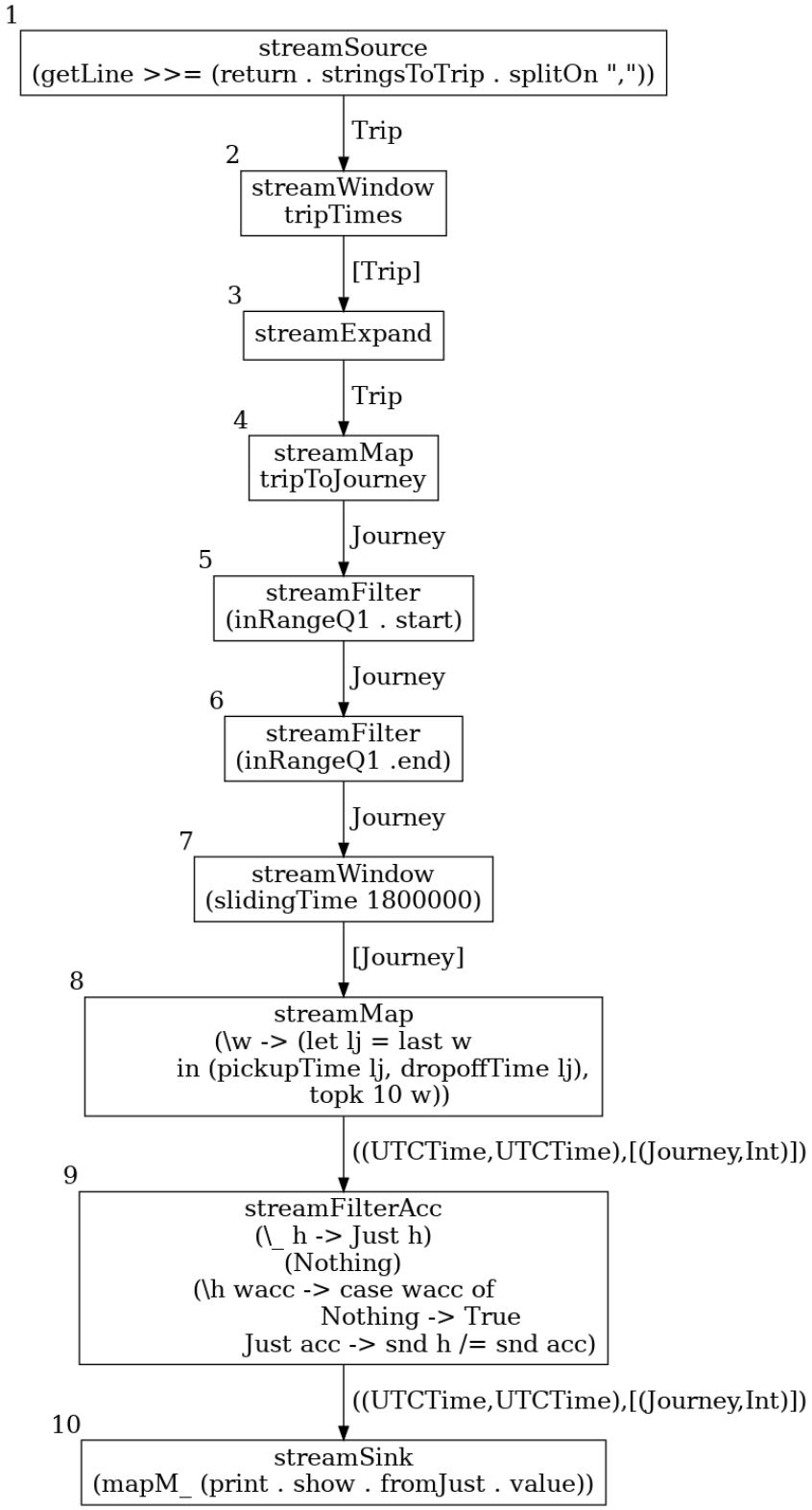


Figure 6.1: A *StrIoT* solution to the DEBS '15 Grand Challenge: profitable Taxi routes

to the generation time, rather than the time of the trip that the data represents. `Events` are emitted as soon as they are read from disk, so the rate of emission and distribution do not correspond to that of the trip data.

For example, the sample dataset we used during development describes Taxi journeys that took place in January 2013, including separate journeys taking place at the same time. Running the simulation in 2025 would result in `Event` `timestamp` fields of 2025, in a strictly sequential, ascending order.

We address this with operator 2, using the `tripTimes` [79, `examples/taxi/Taxi.hs#400`] window-maker which exploits the fact that it (in contrast to the other operators) emits instances of `Event`, and so can generate `Events` with `timestamp` values taken from the data-set.

We do not attempt to adjust the streaming rate or distribution to match the source data, and are careful in our following analysis not to derive conclusions from measurements of these properties in the simulation.

Operators 2-3 result in an adjusted stream of `Events` with the `timestamp` field corrected to match the start time of the journey they represent.

Data Filtering and Reduction

The following three operators (4-6) are used to reduce the stream down to only the data required for this specific scenario.

The first (operator 4) applies `tripToJourney`, converting the `Trip` data type to the smaller, more specific `Journey` type detailed in Table 6.2.

The following two operators (5-6) filter the data to remove journeys that start or end outside of the geographic area of concern, as specified in the problem description.

Top-k and Stateful Operations

The seventh operator applies the `slidingTime` window-maker (described in Section 2.5.2) to collect together `Events` that take place within 30 minute intervals and emit them as a single aggregate `Event` that holds a list of the `Journeys` in that interval.

The eighth operator applies a top-k algorithm to determine the 10 most frequently occurring journeys from the events in each window.

The ninth operator is a stateful filter that compares the output of the previous operator against the previously emitted value. When a new batch of events arrive, and the top 10 results are unchanged from the previous batch, this operator ensures the program does not repeat the previous result.

This operation illustrates that stateful operations are not incompatible with purely-functional programming. We achieve this using `streamFilterAcc`, described in Section 2.5.2.

Trip			
	medallion	::	Medallion
	hacklicense	::	MD5Sum
	pickupdatetime	::	UTCTime
	dropoffdatetime	::	UTCTime
	tripTimeInSecs	::	Int
	tripDistance	::	Float
	pickup	::	Location
	dropoff	::	Location
	PaymentType	::	PaymentType
	fareAmount	::	Dollars
	surcharge	::	Dollars
	mtaTax	::	Dollars
	tipAmount	::	Dollars
	tollsAmount	::	Dollars
	totalAmount	::	Dollars

Location			
	lat	::	Degrees
	long	::	Degrees

PaymentType
card
cash

Medallion	::	MD5Sum
Degrees	::	Float
MD5Sum	::	String

Table 6.1: `Trip` data-type definitions

Journey			
	start	::	Cell
	end	::	Cell
	pickupTime	::	UTCTime
	dropoffTime	::	UTCTime

Cell			
	clat	::	Int
	clong	::	Int

Table 6.2: `Journey` data-type definitions

6.2.2 Path2IOT

For our second example problem, we re-implemented the "Wearable" solution from *PATH2iot* [58] using *StrIoT* and provide it as an example program within the *StrIoT* source [79, `examples/wearable`] .

This program simulates a user wearing a smart device featuring an accelerometer. Readings of the accelerometer measurements are periodically taken from the device and used to calculate an estimate of the user's step count. These counts are aggregated into windows of a fixed time interval. The lengths of these windows are calculated and forwarded to the cloud for further processing.

A flowchart illustration of our implementation is in Figure 6.2. As before, the operators in the flowchart are numbered and we describe the solution in detail below, referencing each operator by number.

Source

The `streamSource` (operator 1) simulates the arrival of data from a wearable sensor. The data format reflects the fields that are emitted by a Pebble smart watch , and are listed in Table 6.3.

PebbleMode60	=	(Accelerometer, Vibe)
Accelerometer	=	(AccelVal, AccelVal, AccelVal)
AccelVal	=	Int
Vibe	=	Int

Table 6.3: PebbleMode60 data-type definitions

Filtering

Operator 2 filters out smart watch events where the vibration sensor was active. These values are not necessary for our data processing.

Processing

Operators 3-5 implement a step-counting algorithm [83].

Operators 3 and 4 calculate the Euclidean distance from the vector expressed in cartesian coordinates within the data packet.

In the original *PATH2iot*, the Euclidean distance operation was decomposed into separate multiplication and square-root calculations. This allowed for deployment plans which placed the multiplication calculations on an Edge device and the more complex and energy-intensive square-root operation in the Cloud. We have preserved this decomposition in our re-implementation.

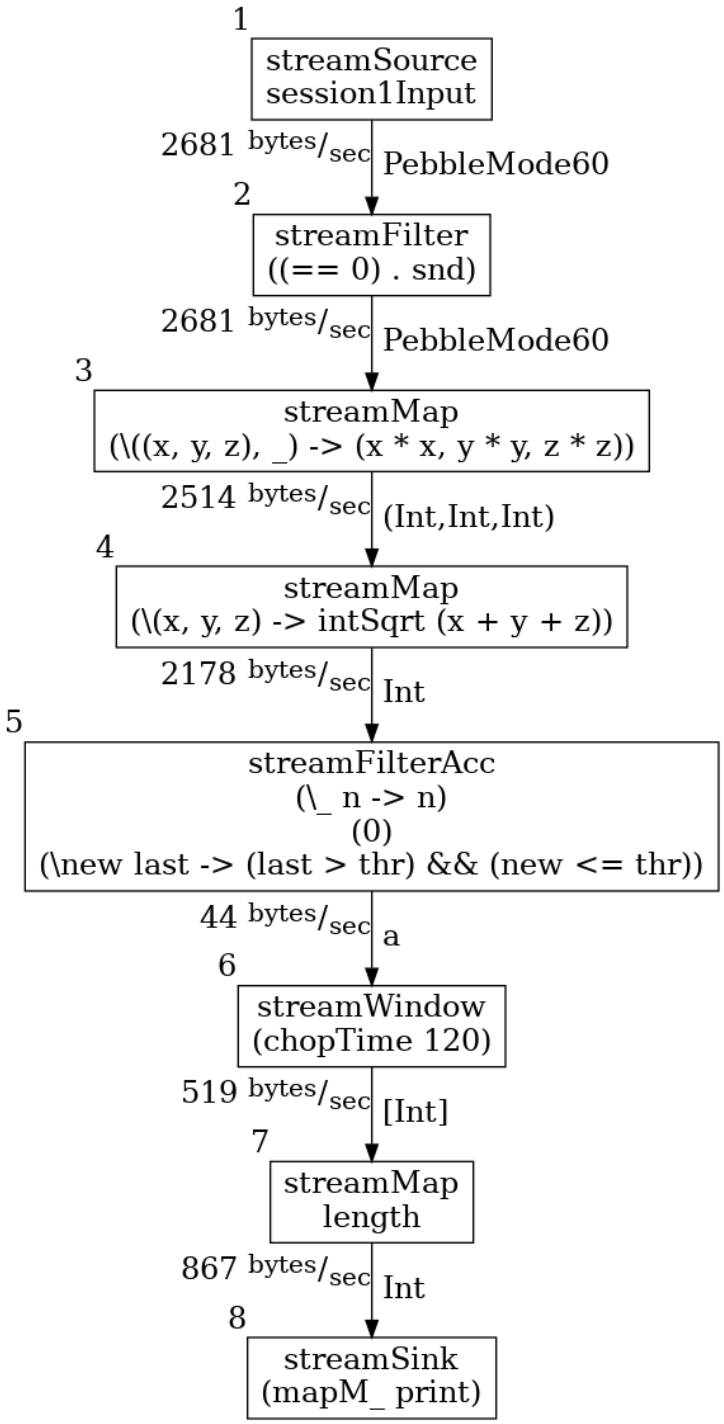


Figure 6.2: A *StrIoT* implementation of the wearable program from *PATH2iot*

The result of this processing is a stream of integers representing the magnitude of a physical movement detected by the sensor. The next operator (5) completes the step-counting algorithm by performing a stateful filtering, comparing the current and prior values against a fixed threshold value. We use the stateful `streamFilterAcc` operator, described in Section 2.5.2.

The final processing that took place in the original *PATH2iot* calculated the number of qualifying measurements that took place within a given time interval. The final two operators in our implementation calculate this by first using a time-based window (operator 6) and then measuring the length of each window (operator 7).

6.2.3 Summary

By successfully implementing solutions to two real-world problems: the DEBS 2015 Grand Challenge (Section 6.2.1) and the Path2IOT's Wearable (Section 6.2.2); we have demonstrated that purely-functional programming can be used to build a functional stream-processing system supporting real-world applications.

Both the Taxi [79, `examples/taxi`] and Wearable [79, `examples/wearable`] examples are included in the *StrIoT* source distribution as full problem solutions that can be partitioned, deployed to containers and run, using the lightweight orchestration tool Docker Compose [21] (or an equivalent). The deployment features of *StrIoT* are fully described in Section 3.3.

6.3 Logical Optimiser

We now explore those aspects of the declarative stream-processing architecture which are the focus of this thesis, starting with the Logical Optimiser.

In Chapter 4 we presented a set of 29 semantically-preserving rewrite rules as well as two sets of rules which altered the precise order of stream events, but we argued were nonetheless useful: 5 rules which caused re-ordering of incoming `Events` and 2 which reshaped windows.

In Section 4.3 we identified many of these rules as belonging to one of five established categories of stream-processing optimisations [38]. In this section we revisit those categories and discuss them in the context of a distributed stream-processing system.

6.3.1 Operator Fusion

7 rules enact *operator fusion*: replacing a pair of operators with a single, combined operator performing the work of the original pair. Operator fusion can be a useful transformation as it reduces the complexity of the stream-processing program. It can also reduce the computation required for executing the program, in particular, by removing the need for intermediate list data-structures to be constructed and traversed

at the ingress/egress of each operator [33], [75]. This problem is explored in more detail in Section 2.2.2.

As described in Chapter 3, When a stream-processing program is deployed, the Haskell source code describing the sub-programs for each deployment node are generated and compiled in isolation from one another as stand-alone computer programs. When such a program features a pair of candidate operators for fusion, this could be identified and achieved using established technologies such as GHC rewrite rules [66] and other compile-time optimisations.

We have therefore determined that there is little value-add in *StrIoT* performing this work prior to code generation. It could even have a negative effect: by combining two operators, the result can only be assigned to one deployment node, ruling out potential deployment plans where one of the operators is assigned to a deployment node earlier in the stream than the other, potentially reducing the size or rate of data.

6.3.2 Operator Re-Ordering

11 rules implemented *operator re-ordering*: swapping the order of the pair of operators (and in some cases, also changing the type of one or both operators).

Operator re-ordering is interesting in the context of a distributed stream-processing system because it can involve moving an operator between different nodes in a deployment plan. This enables at least three types of optimisation:

1. by moving a pair of operators to be newly adjacent, the pair may end up assigned to the same deployment node, opening up the potential for the compiler operating on that node to perform optimisations such as operator fusion (described above).
2. moving operators towards the source of the stream-processing program can enable the assignment of more work to earlier nodes in a deployment plan, corresponding to Edge devices. This can increase the utilisation of Edge devices, and reduce the total number of nodes required in a deployment plan. We provide an example of this in Section 6.5, below.
3. moving mapping operators which reduce the `Event` payload size or filtering operators which reduce the rate of `Events` cause a reduction in the required bandwidth downstream from those operators. This is another advantage to moving operations towards the Edge. We explore bandwidth in more detail in Section 6.6.

6.4 Rejecting Over-Utilised Operators

StrIoT is designed to flexibly support multiple optimisation criteria within the Evaluator. To demonstrate, we have implemented two within the proof-of-concept: utilisation

and bandwidth. Their design and implementation are discussed in Chapter 5.

StrIoT's Evaluator accepts as input a plan: a pairing of stream-processing program and mapping of its operators to nodes in a deployment. However, the queueing theory model (described in Chapter 5) is derived solely from the stream-processing program, without the mapping of operators to nodes. This limitation is discussed in Section 5.8.1.

Prior to ranking plans, the Evaluator rejects those that are determined to be unsuitable for deployment. The queueing theory model is used to reject plans where an individual operator is determined to be over-utilised: events arrive at the operator at a higher rate than it is capable of processing them.

We demonstrate this outcome with the example user-supplied program illustrated in Figure 6.3. The operators have been annotated with the user-supplied event arrival rates (λ), filter selectivities (f) and service rates (μ); as well as their utilisation (ρ) as calculated by the Evaluator's cost model.

The supplied program has been determined unviable for deployment: the filter operator (shaded in red) is over-utilised ($\rho > 1$). Attempting to deploy this stream-processing program would result in an ever-growing queue of `Events` waiting to be processed at the filter operator.

The Logical Optimiser derives 4 rewritten programs from that depicted in Figure 6.3. Illustrations of all 4 programs are provided in Chapter D.

One variant, depicted in Figure 6.4, has been rewritten by applying Rule R3. The `streamFilter`, originally downstream from the `streamMerge`, has been removed and new `streamFilter` operators inserted on each incoming stream to `streamMerge`.

Applying the Evaluator to this rewritten program shows that none of its operators are over-utilised. It would not be rejected at this stage of evaluation, and deployment plans based on this program would be ranked by the Evaluator for potential deployment.

The rewritten program features more operators than the original program. A simpler cost model, such as one that minimised the total number of operators (described in Section 5.2) would not prefer the rewritten program and would select an option that was worse in practice.

6.5 Node Utilisation Threshold

StrIoT permits the user to define a maximum node utilisation as one of the parameters supplied alongside the stream-processing program. This value is used by the Evaluator to limit the amount of work that can be assigned to any node within a deployment plan. This is described in detail in Section 5.1.3.

Figure 6.5 depicts an example program consisting of a set of temperature sensors, combined with `streamMerge`, followed by a series of highly-utilised `streamMap` operations. For numerical convenience, the event arrival rates and operator service times

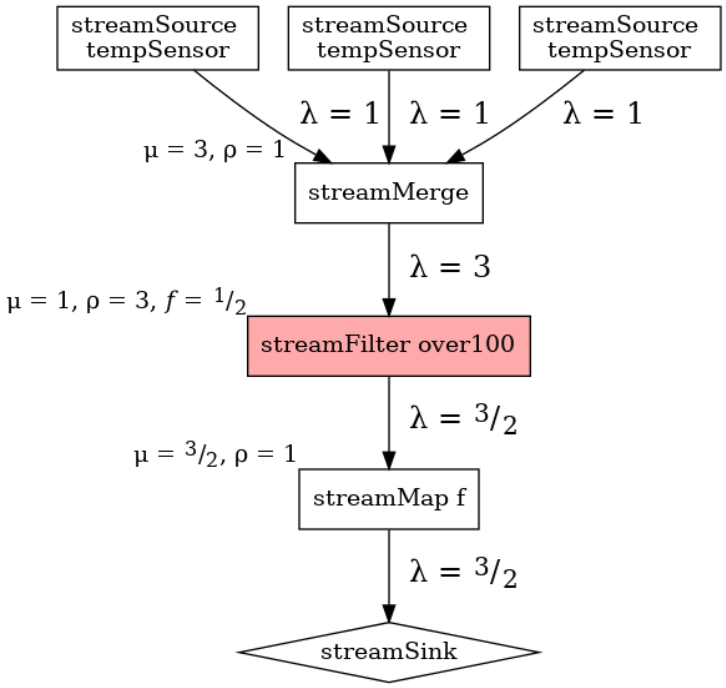


Figure 6.3: Example program with over-utilised operators shaded red

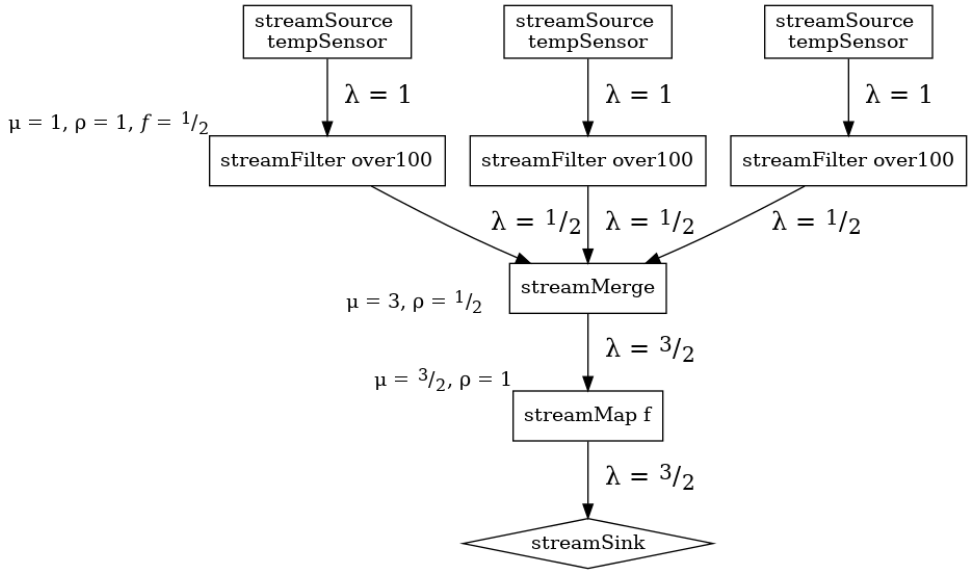


Figure 6.4: Rewritten program from Figure 6.3 with no over-utilised operators

have been chosen such that the `streamMap` operators are maximally utilised ($\rho = 1$).

The user-provided parameters are provided in Table 6.4. Maximum node utilisation is specified as 300%, meaning a maximum of three fully-utilised operators could be assigned to any given node.

Parameter	Value
maxNodeUtil	300%
maxBandwidth	*unspecified*
rules	defaultRewriteRules

Table 6.4: User-supplied parameters for the Utilisation example

We now explore the performance of *StrIoT* on the user-provided program followed by the results of applying the Logical Optimiser.

6.5.1 Original Program Performance

Without considering maximum node utilisation (i.e. when the user has not provided it), the only constraints on the system are the limitations imposed by the Deployer, detailed in (Section 2.5.3): source and sink operators cannot share a node; `streamMerge` can only be the first operator in a node. Operating under only these constraints, for the program in Figure 6.5, The Evaluator would choose plans consisting of 3 nodes.

In our example, each `streamMap` operator within the program is fully utilised: $\rho = 1$, or 100%. Considering the maximum node utilisation of 300%, any deployment plan that assigns more than 3 `streamMap` operators to a node is ruled out by the Evaluator. The lowest-possible number of nodes is now 4.

Figure 6.5 depicts one of the best-scoring deployment plans under this constraint, requiring 4 nodes.

6.5.2 Logical Optimiser Performance

In an edge-to-cloud configuration, a deployment typically consists of a fixed number of Edge devices, connected (via gateways) to a variable amount of Cloud computing resource. Traditionally the majority of processing would take place on cloud nodes, with Edge devices limited to gathering and forwarding data such as sensor data.

Where it is possible to perform some computation on Edge devices, the quantity of cloud resource required to complete the computation is reduced, resulting in cost savings.

The Logical Optimiser produces 3,524 variants of this program. One of the best-scoring is depicted in Figure 6.6. By applying Rule 19 (See Section 4.5.4), some of the stream processing has been moved upstream, next to the sources, prior to the `streamMerge`. This has increased the utilisation of the nodes at the Edge. The remaining

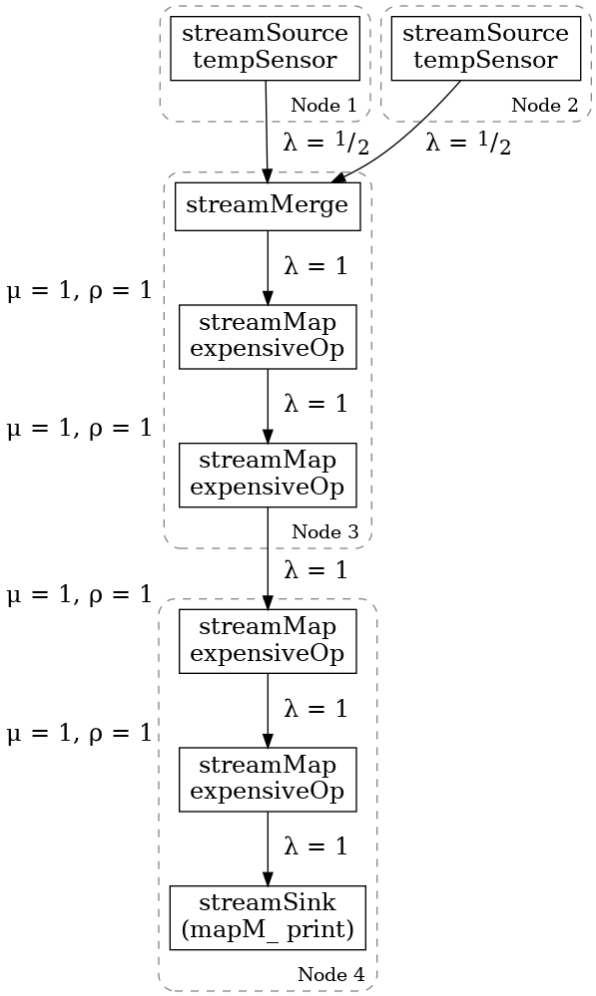


Figure 6.5:
A deployment plan for a stream-processing program with 4 consecutive highly-utilised operators (each $\rho = 1$). *maximum node utilisation* is specified as 300%. Consequently no more than three of these operators can be assigned to a single deployment node. The chosen plan requires 4 nodes.

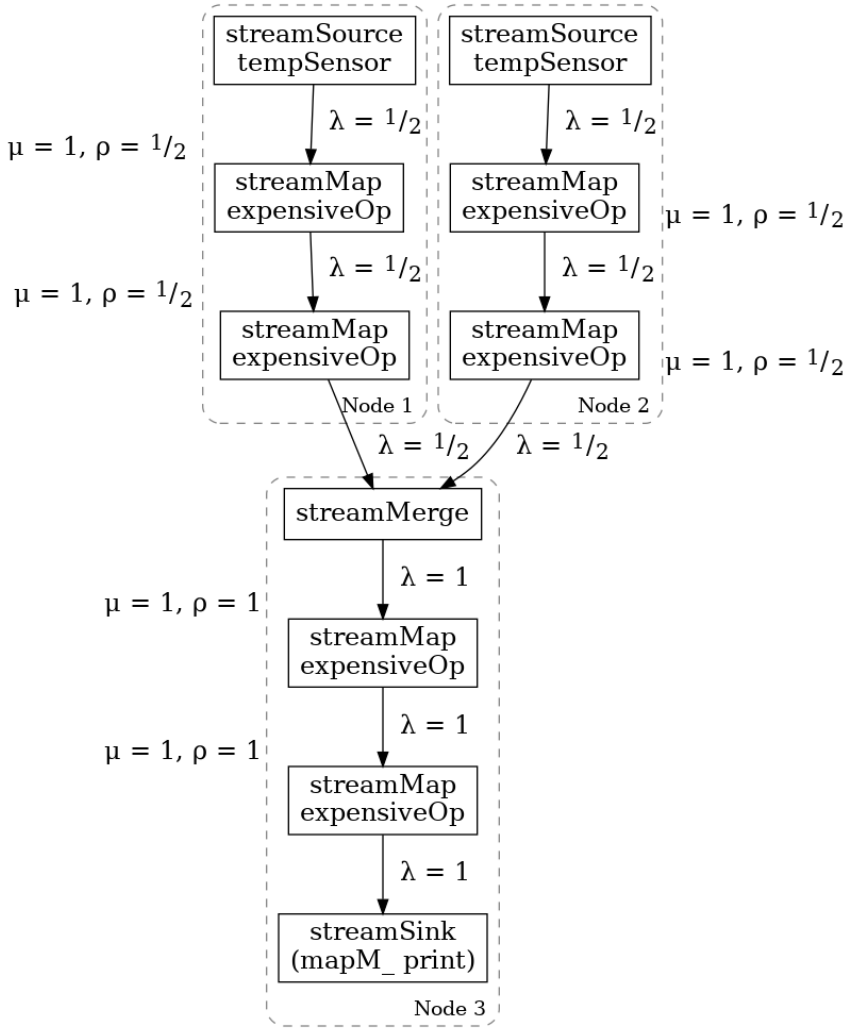


Figure 6.6: A deployment plan featuring a variant of the program in Figure 6.5, derived by the Logical Optimiser. This plan requires only 3 nodes, one fewer than Figure 6.5.

operators can be assigned to a single node, resulting in a deployment plan of only 3 nodes: an improvement over the 4 required by the original program.

This example has demonstrated that by considering maximum node utilisation, *StrIoT* can better model a real-world constraint. By applying the Logical Optimiser, we are able to derive a program variant which moves some computation earlier in the stream, increasing the utilisation of edges nodes in an edge-to-cloud scenario, and allowing the Evaluator to select plans requiring fewer resources.

6.6 Utilisation and Bandwidth

We now explore the performance of both the Utilisation and Bandwidth aspects of the Evaluator combined by applying them to the Wearable example originally introduced in Section 6.2.2.

6.6.1 Estimating Model Parameters

Along with the stream-processing program, the cost models used by the Evaluator also require the user to supply information describing the mean average arrival rate of data into the system; the mean average service times for each operator and the average selectivity for each filtering operator (See Section 5.7.1).

To determine reasonable values for these parameters, we wrote short stream-processing programs [79, `examples/wearable/WearableStreams.hs`] to process a sample data-set provided by the *PATH2iot* authors and measure the required properties. This sample-data was recorded from a "Pebble" smart-watch. A description of the data-set and how to obtain it is provided in Section A.3. These measurements are summarized in Table 6.5. The programs used to measure each parameter are described below.

Parameter	Measurement
data arrival rate	20.947 Hz
vibeFilter selectivity	$9.999\,8 \times 10^{-1}$
stepEvent threshold	1,250
stepEvent selectivity	$2.027\,2 \times 10^{-2}$

Table 6.5: Data arrival rate and filter selectivities for the Wearable program from Figure 6.2.

Arrival Rate

The arrival rate of data into the system is straightforwardly calculated directly from the sample data using a rolling average [79, `examples/wearable/WearableStreams.hs#97`]

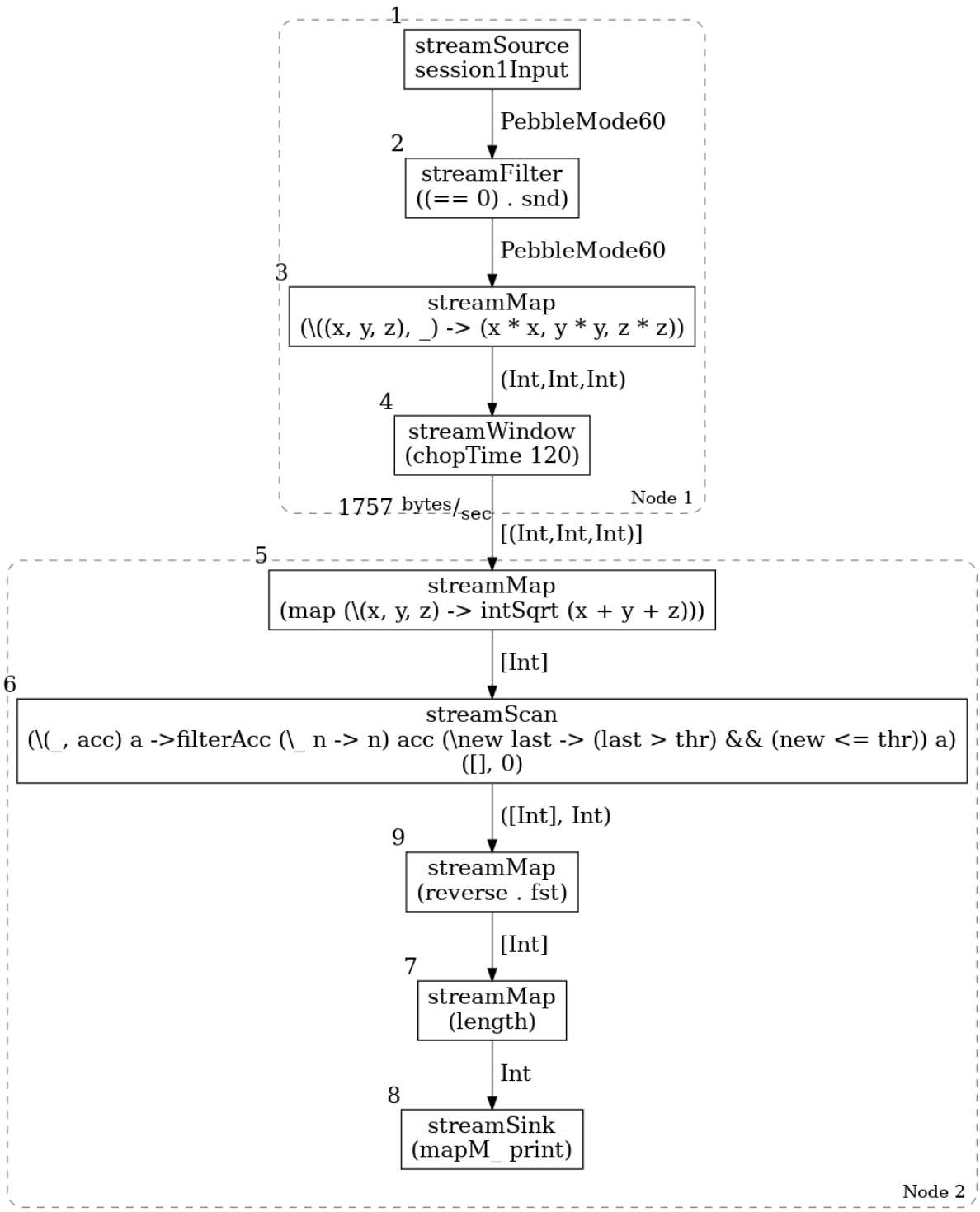


Figure 6.7: The rewritten Wearable program from Figure 6.2.
The chosen deployment plan is indicated with dashed lines.

```
arrivalRate s = s
              & streamWindow (chopTime 1000)
              & streamMap length
              & streamScan (\(count,sum,_,_) n -> let
                count' = count+1
                sum' = sum+n
                avg' = (fromIntegral sum') / (fromIntegral count')
                in (count',sum',n,avg')) (0,0,0,0.0::Double)
              & streamMap (\(_,_,_,x) = x)
```

Listing 6.1: Code to calculate the average arrival rate of pebble-watch data

`vibeFilter` Selectivity

The first filter in the program removes events where the vibration field is set. We estimate average selectivity by calculating the actual selectivity over our data-set using `vibeSelectivity` [79, `examples/wearable/WearableStreams.hs#161`] :

```
vibeSelectivity file = let
  total  = (fromIntegral . length . lines) file
  accept = (fromIntegral . vibeCount) file
  in accept / total

vibeCount =
  length . unStream . edEvent . streamMap snd . pebbleStream'
```

Listing 6.2: Code to calculate the selectivity of `vibeFilter`

`stepEvent` Selectivity

We use a similar technique to measure the actual selectivity of the `streamFilterAcc` operator which completes the step-count algorithm. The threshold value of 1250 is taken directly from [58]. We calculate the selectivity with `stepSelectivity` [79, `examples/wearable/WearableStreams.hs#134`] :

```
stepSelectivity file = let
  edEvents  = file & pebbleStream' & streamMap snd & edEvent
  edEvLen   = edEvents & length & fromIntegral
  stepEvLen = edEvents & stepEvent 1250 & length & fromIntegral
  in stepEvLen / edEvLen
```

Listing 6.3: Code to calculate the selectivity of `stepEvent`

6.6.2 Operator Service Times

To determine average service times for each operator, we performed a benchmarking experiment using the Criterion package [63].

Criterion is designed to support the user writing "micro-benchmarks": individual benchmarks of specific functions. Criterion repeatedly evaluates the provided functions and records the time taken to evaluate them. It estimates the time taken for a single iteration using a ordinary least-squares regression (OLS) model. This provides a more accurate estimate than a simple mean average, as it accounts for measurement overhead.

Criterion provides an R^2 goodness-of-fit to measure how accurately the OLS model fits the observations. The Criterion authors describe R^2 values of between 0.99 and 1 as an "excellent fit". All of our measurements have R^2 values within this range.

Our benchmarking code is provided with *StrIoT* [79, `examples/wearable/Criterion.hs`].

Hardware Selection

To control for the impact of hardware performance on our benchmarks, we performed the experiments on a single, dedicated machine. It was not practical to attempt to execute *StrIoT* on an edge device of a similar calibre to the Pebble Watch used in *PATH2iot*: at the time of our experiments, the device was no longer in production. It would also have been a significant undertaking to build a cross-compilation environment in order to target the architecture of the Pebble Watch CPU (STM32, 32-bit ARMv7-M architecture).

Instead, we selected a single-board computer designed for IoT deployments which used the same CPU architecture as conventional desktop computers (amd64). This simplified the development of the test suite, as we could iteratively develop the suite on regular, high-performance machines, whilst executing them in the lower-power, controlled environment, without needing to build a cross-compiler.

The specific configuration of our test machine was a 4 core Intel Atom X5-z8350 CPU at 1.92GHz and 4GB RAM. The data-sheet is available at [74].

Benchmark implementation

The general pattern for each micro-benchmark is to call the code for each operator via Criterion's `bench` and `whnf` functions. These ensure that the function is evaluated to weak-head normal form (whnf) prior to supplying the final datum for each run of the benchmark. By wrapping the function to be measure in `whnf`, Criterion ensures that it has not been optimised away to a constant value by GHC.

To ensure we measure both fast and slow code-paths, we benchmark each operator with a selection of input values. For example, Listing 6.4 lists two measurements for `vibeFilter`, one for each possible value for the `vibe` field (`0` and `1`).

```
bench "vibeFilterYes" $ nf vibeFilter ((0,0,0),0)
bench "vibeFilterNo"  $ nf vibeFilter ((0,0,0),1)
```

Listing 6.4: Example Criterion benchmark code for `vibeFilter`

Benchmarking Results

The average measured service times are summarized in Table 6.6. The full report generated by Criterion is provided in Chapter E.

ID	Operator	Average Service Time (s)
2	vibeFilter	7.73×10^{-7}
3	squares	9.19×10^{-7}
4	intsqrt	3.4×10^{-6}
5	filterAcc	1.6×10^{-6}
6	chopTime	1.24×10^{-6}
7	length	1.5×10^{-7}

Table 6.6: Measured average service times for the operators in the program from Figure 6.2.

User-supplied parameters

Parameter	Value
maxNodeUtil	$1.110\,2 \times 10^{-2}$
maxBandwidth	1760 bytes/s
rules	defaultRewriteRules ++ reshapingRules

Table 6.7: User-supplied parameters for the Wearable example

In addition to the program description itself, the user supplies *StrIoT* with configuration parameters which include thresholds for the Evaluator. The user has customised the set of rewrite rules to apply by adding reshaping rules (See Section 4.5.7) to the default set. The parameters supplied in this example are shown in Table 6.7.

We have scaled the maximum node utilisation to adjust for the difference in performance between the equipment used for benchmarking the operators (Section 6.6.2) and the Pebble Watch used in the original *PATH2iot* research.

6.6.3 Original Program Performance

First, we explore the behaviour of the Evaluator on the original program supplied by the user.

The Partitioner generates 127 potential deployment plans for the input program.

The Evaluator is then applied to each pairing of the program and deployment plan. First, plans are rejected if they breach either of the calculated maximum node utilisation or bandwidth limits.

Consider first the maximum node utilisation. 31 plans are rejected, leaving 96. Next we apply the bandwidth filter (described in Section 5.1.4) which rejects all the remaining plans.

No viable plans remain for deployment. In other words, the original program has been determined to be impossible to deploy and operate under the constraints specified by the user.

Without the Logical Optimiser, the user must either alter their program or relax the operating constraints to proceed: for example, purchase or rent larger, more expensive cloud computing resources.

6.6.4 Logical Optimiser Performance

We now consider the situation where we first apply the Logical Optimiser to generate a set of new but functionally equivalent versions of the program submitted by the user.

66 variants are derived from the original program. Once paired with all possible deployment plans, there are 6,989 Plans to be filtered and ranked by the Evaluator.

This time, the maximum node utilisation filter removes 4,673 options, and the bandwidth filter removes a further 2,061, leaving 255 viable plans for ranking. The distribution of costs calculated for these plans is illustrated in Figure 6.8.

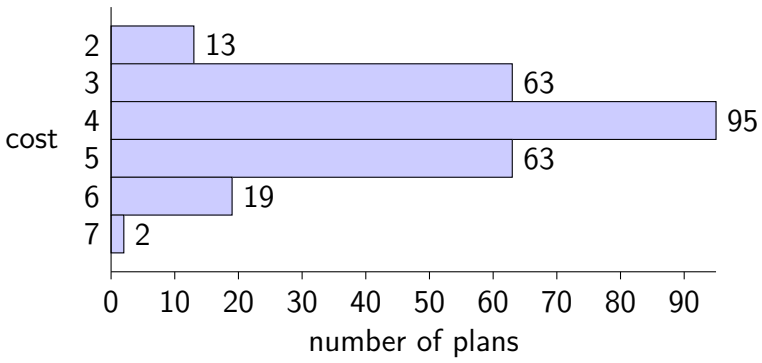


Figure 6.8: Distribution of costs calculated for deployment plans

Of the 13 lowest-scoring (best) plans, the variant depicted in Figure 6.7 is chosen by the Evaluator. This program was derived from the original by first applying the window reshaping Rule W2 (See Section 4.5.7) which moved the `streamWindow` one position earlier in the program, followed by applying Rule 21 (Section 4.5.4) twice in succession: each time moving the `streamWindow` another position earlier.

The bandwidth model of the Evaluator calculates the bandwidth at the boundary between the first and second deployment nodes. In the original program (for plans

not already rejected by the maximum node utilisation) the events flowed over this boundary at the rate of the source node, and the calculated bandwidth was determined to be too high for all plans (between $2{,}178\ \frac{\text{B}}{\text{s}}$ and $(2{,}681\ \frac{\text{B}}{\text{s}})$.

In the rewritten program, the events flowing over the first node boundary are the result of applying `streamWindow`, and are flowing at the rate specified by the window-maker, which is much lower. The resulting bandwidth ($1{,}757\ \frac{\text{B}}{\text{s}}$) is calculated to be below the required threshold ($1{,}760\ \frac{\text{B}}{\text{s}}$).

The Logical Optimiser has therefore produced a deployable program, unlike the original program provided by the user.

6.7 Chapter Summary

In this chapter we have evaluated the design of *StrIoT*, our reference implementation of a purely-functional distributed stream processing system, by first addressing the research questions posed earlier in this thesis, before exploring in further detail the facets of the design which are the focus of this work: the Logical Optimiser, cost models and Evaluator.

We have implemented several solutions to real-world stream-processing problems using *StrIoT*, adding evidence to support the viability of the declarative stream-processing model and showing that purely-functional programming is a viable approach for such systems.

We have discussed where we have shown particular advantages or disadvantages of our design, as a consequence of purely-functional program, in contrast to existing stream-processing systems.

We have demonstrated the value of generating program variants as well as the use of a cost model to filter and rank deployment plans, by finding viable deployment plans when the original user-supplied program is unviable.

6.7.1 Key Insights

We note in Section 6.3 that some of the rewrite rules we designed can be directly classified in terms of established categories of stream-processing optimisations. However, we observe in Section 6.3.1 that some optimisations, including *operator fusion*, when performed at this level of abstraction can actually thwart more efficient transformations, by removing opportunities for more effective transformations such as operator placement.

The application of term rewriting to purely-functional stream processing programs can generate a large number of program variants: in our examples, the number of variants is too great feasibly assessed and ranked by a human. For the wearable example (Section 6.6), 6,989 plans were generated. Allowing 15 minutes to evaluate a plan, a human would require 218 working days of 8 hours per day to evaluate them all.

This clearly shows that machine assistance for filtering and ranking program variants is essential.

The result achieved in Section 6.6.4 relied upon the selection of rewrite rules that modified the program's semantics: specifically rules which permitted the "reshaping" of windows (detailed in Section 4.5.7). This result demonstrates that there is value in developing and applying rewrite rules which modify a program's semantics, so long as those semantics are not relevant to the ultimate results calculated by the program. We discuss this further in Section 7.3.1.

Chapter 7. Conclusion

7.1 Contributions

I have demonstrated, via an end-to-end proof of concept implementation, that the architectural vision for declarative stream-processing (described in Section 2.4) is viable. I successfully implemented several solutions to real-world stream processing problems, including re-implementing an example from an earlier relational system (Section 2.4.1).

A drawback of the earlier relational system was reliance upon user-defined functions (UDFs), which prevented the Optimiser from reasoning about their semantics [58]. In contrast, *StrIoT* restricts users to a small set of operators with well-understood semantics. Our example programs show that this restriction does not prevent solving real-world stream-processing problems. I were able to design program transformations for our Logical Optimiser involving all of the operators.

Purely-functional programming is used for both our implementation and the language we present to the user for authoring stream-processing programs. Whilst concepts from purely-functional programming have been adopted in existing distributed stream-processing systems, we believe *StrIoT* to be the first such system to adopt it end-to-end.

I have not only shown that purely-functional programming is a practical method to build a system of this complexity, but that its properties – namely pure functions and lazy evaluation – combined with the restricted operators – enable the use of powerful reasoning methods (equational reasoning, term rewriting) in the design of the Logical Optimiser.

Program optimisations are traditionally semantically-preserving. I have discovered that it can be advantageous to alter some aspects of their behaviour in well-defined ways, such as re-ordering streaming data. I believe there is value in controlling which aspects of the semantics of a program are important, and which can be altered. (See Section 7.3.1 for further discussion).

My approach to the Logical Optimiser is not specific to stream-processing and could be applied to other domains utilising dataflow models, including e-Science workflows, large language models (LLMs) and graphical rendering pipelines.

7.1.1 Open Science and reproducibility

This work has been produced under the principles of open and reproducible science. Full-text pre-prints are available for the associated publications [9], [24]. We have developed *StrIoT*, the research software underpinning this work, as open source software [79]. To support reproducible science, the data sets used for the experiments in Chapter 6 have been archived in the Newcastle University Research Repository, alongside a copy of the *StrIoT* source code. Full details of the precise version of the source code used for the experiments and a description of the data-sets and their formats are provided in Section A.1.

7.2 Thesis Summary

The motivation for this research was set out in in Chapter 1: designing and operating modern stream-processing systems is complex, due to the demands of high velocity data and the range of different technologies and expertise required for them. A declarative architecture was proposed [57] to address these problems. Michalák explored the architecture using a relational approach [56], whilst Watson and Woodman produced an initial prototype using a functional approach [78].

Chapter 2 provided a summary of the relevant foundational concepts and research in both stream-processing and functional programming. I then thoroughly explored the proof-of-concept functional prototype that we build upon with our work, including its data-types (Section 2.5.1), stream-processing operators (Section 2.5.2), containers and initial run-time (Section 2.5.3). The prototype lacked the Deployer, Logical Optimiser and Evaluator components of the architecture.

Chapter 3 describes the work I undertook to design and implement the Deployer, to divide a stream-processing program into sub-programs, and the Partitioner, to generate mappings of program operators to deployment nodes. I describe the data-types and algorithms that I designed and implemented for these components.

Chapter 4 explores term rewriting for transforming stream-processing programs. I designed a library of rewriting rules and categorised them according to established categories of stream-processing optimisations. I initially approached program rewriting as strictly semantically preserving before realising the opportunities of rewrite rules which alter some aspects of program behaviour. I detail the implementation of rewrite rules and the design and implementation of the Logical Optimiser to apply them.

Chapter 5 discusses the design of cost models, using queueing theory to model the utilisation of stream-processing operators, and bandwidth constraints for data propagating through the distributed system, followed by a description of their implementation and the algorithm used to apply them in the Evaluator.

In Chapter 6 I evaluate the performance of the completed *StrIoT* by re-implementing

solutions to real-world stream-processing problems, applying the Logical Optimiser to generate alternative plans, the Evaluator to cost those plans, and discussing the outcomes.

7.3 Future Work

Throughout this thesis I have identified several opportunities for future work, described below.

7.3.1 Semantic-Modifying Program Transformations

In Chapter 4, I designed a set of rewrite rules for stream-processing programs which were semantically-preserving: the functional behaviour of the rewritten program is equivalent to that of the original. Preserving semantics is a fundamental axiom of term rewriting, the formal method upon which rewrite rules are based.

During the course of this work I discovered that there are some aspects of the semantics of a stream-processing program that were unimportant in some contexts: for example, preserving the ordering of stream data might not impact the work performed by the program. I developed two sets of rewrite rules which were not semantically preserving: one set which re-ordered streams (Section 4.5.6) and another which reshaped windows (Section 4.5.7). These rules are provided as a separate collection within *StrIoT*, allowing users to opt-into using them.

In Section 6.6, in a scenario where the user-supplied program was not deployable under the operating constraints, I determined that window reshaping rules enabled the Logical Optimiser to derive a deployable plan. It is clear that there is value in rewrite rules which alter some aspects of the semantics of a stream-processing program.

Future work could categorise the different facets of program semantics and explore ways in which each could be specified as important (or otherwise) by the end-user.

Regarding stream order: In *StrIoT* the `Stream` type is modelled as a list, which is an inherently ordered data-type. Alternative representations could include unordered types such as sets. For situations where order is important for some portion of stream-processing and not others, an exploration could be made of more complex schemes, such as dependent types [6], to represent both circumstances in the same program.

Rather than require the end-user to identify and specify where different semantics are important, further research could explore to what extent this could be automatically determined by program analysis or modelling.

Stream Re-Ordering

Further analysis could be performed to explore the re-ordering rules described in Section 4.5.6, in particular to establish whether the original stream order could be preserved or recovered after the transformations. For some of the rules, the extent of

stream re-ordering is externally determined, such as Rule R1, where it is determined by the user-supplied filter predicate. For others, the re-ordering is determined by internal factors, such as the number of incoming streams and the size of windows for Rule R2. In some circumstances, there may be enough information to design re-ordering rules which can restore the stream order to match that prior to the rewrite.

For example, Listing 7.1 is a variation of Rule R3 which preserves ordering:

```
streamFilter p $ streamMerge [s, t]
  ⟶ streamMap fromJust
    $ streamFilter isJust
    $ streamMerge [ streamMap (filterMaybe p) s
                  , streamMap (filterMaybe p) t ]
```

Listing 7.1: an order-preserving variant of Rule R3

In this variation, instead of rejecting filtered events with `streamFilter`, `streamMap` is used to replace accepted events with `Just x`, and rejected events with `Nothing`. This preserves stream ordering, yet reduces its data-size, if `Nothing` serialises to a smaller representation than the original data-type, which is likely.

Once the streams have been merged, two new operators are used to remove the events consisting of `Nothing` and unwrap the original data-type from `Just`. It may be possible to design order-preserving equivalents of the other rules in Section 4.5.6 using similar techniques.

Alternatively, altering the stream operators themselves could be explored. Recall that the `Event` type (Section 2.5.1) already wraps the payload in `Maybe`. The definition of `streamFilter` could be adjusted such that, when rejecting an `Event`, it emitted a corresponding `Event` consisting of `Nothing` as the payload. Such a change would require a full analysis of the new semantics and likely alterations to the definition of the other operators.

7.3.2 Catalogue

The original declarative stream-processing architecture included a catalogue to describe the available devices (sensors, gateways, cloud instances) available within the deployment environment and their properties, such as availability, capability and cost (Section 2.4). We did not implement a catalogue during this research.

I implemented two cost models in the Evaluator: utilisation (Section 5.5) and bandwidth (Section 5.7.2). Without a catalogue we made a number of simplifications. For utilisation, maximum node utilisation is a threshold provided by the user as an alternative for modelling the actual limits of deployment nodes. The bandwidth model assumes that the connection of interest, to which the limit should be applied, is between the first and second nodes in the deployment, which would correspond to the interface between the Edge and Gateway in an Edge-to-Cloud topology. These assumptions could be replaced with a richer description of the operating environment.

Without the catalogue, we considered deployment nodes as homogeneous and modelled them as a simple list. We did not consider different types of deployment nodes, such as varying costs or capabilities.

Interesting characteristics of the deployment environment that could be considered in both cost models and deployment plans include the relative capabilities of nodes (e.g. number of CPU cores, core clock speed, available memory and storage); the availability of particular features (GPUs, support for floating point operations); operating constraints such as estimated battery life, for edge devices; and rental cost, in the case of cloud computing instances.

7.3.3 Run-time

We did not explore the run-time monitor component of the declarative architecture in this research. Cattermole investigated run-time adaptation in [8]. Further development of the run-time aspect of *StrIoT* could support several improvements.

Our current cost models rely upon the user providing estimates of several properties of their program and its environment: the average data arrival rate, average service times for each operator and average selectiveness of each filter (Section 5.7.1). A run-time could instead calculate values for these properties based on the actual performance of the system. Together with further work to support re-deployment, this could allow for replacing a deployed stream-processing program with a new plan which scores better for the applicable cost models according to re-calculated properties.

7.3.4 Machine-assisted Rewrite Rule Generation

My initial experiments with QuickSpec (Section 4.7) demonstrated that tool-assisted rule discovery has the potential to discover laws about the stream-processing operators which can be converted into rewrite rules.

I quickly discovered that the computer memory requirements to explore a set of operators grew rapidly as the number of operators and supporting functions in scope for analysis was increased. I was unable to run QuickSpec against the full set of operators and a reasonable set of supporting functions on conventional machines.

Further work could explore executing QuickSpec in the cloud, on virtual machine instances provisioned with very large available memory, to see whether it can discover further rules or rules of a complexity beyond those I designed via pairwise comparison.

7.3.5 Representation

StrIoT's stream-processing operators (Section 2.5.2) were designed to be simple and easy to use. They are defined as standard Haskell functions with the intent that the end-user describes the stream-processing to take place in terms of these functions in a regular Haskell program.

In contrast, the Logical Optimiser and Deployer components of *StrIoT* are required to manipulate the stream-processing program as structured data. To ease the design and implementation of these components, we defined a distinct set of data-types tailored to their requirements, described in Section 3.2.4. In comparison to the pure functional operators, writing stream-processing applications in terms of these data-types is awkward and unnatural.

I opted to build upon an algebraic Graph type `Algebra.Graph` [60] for this alternative representation. This allowed us to leverage an existing library of functional graph algorithms. For the implementation of rewrite rules (Section 4.8), I was able to use pattern-matching in the function definitions. This aided in their translation from the initial abstract representation, which also used pattern-matching. There were some drawbacks of this choice.

Topology. The topology of the program was entirely encoded at the level of the `Graph` type provided by the library, and was not represented at the level of our vertex type (`StreamVertex`). In order to ensure that two otherwise-indistinguishable vertices were not incorrectly unified by the library, I needed to add a unique ID field to `StreamVertex` (Section 3.2.4).

Locality. A given vertex may occur repeatedly within an instance of a graph data-structure. This meant graph transformations could not be performed on subsets of a stream-processing program (such as the subset that matches a rewrite rule pattern) as they might miss an instance in the wider program. Rewrite rules therefore had to return another function to perform the necessary transformation, which had to be collected and applied by the code which called the rewrite rules. This complicated the implementation.

`Algebra.Graph` provided an interface consisting of total functions that prevents the construction of invalid graphs. However, the structure of our stream-processing programs is actually the more restricted type *tree*. It is possible to construct instances of `StreamGraph` which are valid graphs, but not trees, such as an operator connecting onwards to more than one downstream operator.

Several promising extensions to the Haskell language could help to define an alternative, unified data-type addressing these problems: Template Haskell; typed Templated Haskell and Generalised Algebraic Data Types.

Template Haskell (TH) is a meta-programming framework that permits the user to write Haskell code which is evaluated at compile-time. This code can itself generate Haskell which is combined with regular code. I used TH in Section 3.2.4 to improve the encoding of operator parameters in the `StreamVertex` type. This ensured that the provided parameters were valid Haskell expressions, preventing one class of possible programming errors.

By default, TH is untyped, so nothing prevents the user from accidentally providing a Haskell expression with an invalid type. Listing 7.2 is an example of an instance

of `StreamVertex` which is expected to receive and emit a stream of type `String`, but features a parameter describing a numerical operation which cannot be applied to it. This does not generate a type error when the program is compiled, and instead will provoke a run-time error when the resulting program is deployed.

```
StreamVertex 3 Map [[| (+1) |]] "String" "String" 1
```

Listing 7.2: An example of a type error in an instance of `StreamVertex`

Adopting Typed Template Haskell [32] could potentially catch these errors. If the `parameters` field of `StreamVertex` was typed, then we could potentially remove the `intype` and `outtype` fields. This would improve the legibility of the representation and further close the gap between these data-types and the original stream-processing functions.

Generalised Algebraic Data Types (GADTs) are an extension to algebraic data-types that allow separate constructors to return different types. Figure 7.1 demonstrates an initial translation of *StrIoT*'s stream-processing operators into a GADT. Consider the snippet in Listing 7.3 consisting of two operators:

```
streamFilter even . streamMap (+1)
```

Listing 7.3: Simple example of two *StrIoT* operators

The equivalent snippet realised with the GADT type is in Listing 7.4. This closely resembles the snippet in Listing 7.3, and so constructing programs using this type should not be more onerous for the end-user, and the resulting data-type can be evaluated, traversed, inspected and de-constructed by a higher-level program such as a Logical Optimiser.

```
StreamFilter [|| even ||] . StreamMap [|| (+1) ||]
```

Listing 7.4: The example from Listing 7.3 formulated using the GADT type in Figure 7.1

```
type StrExp a = Code Q a -- typed Template Haskell expression

data StreamProgram o where
  StreamSource    :: StrExp o
                  -> StreamProgram (Stream o)

  StreamSink      :: StrExp (i -> o)
                  -> StreamProgram (Stream i)
                  -> StreamProgram o

  StreamMap       :: StrExp (i -> o)
                  -> StreamProgram (Stream i)
                  -> StreamProgram (Stream o)

  StreamScan      :: StrExp (o -> a -> o)
                  -> StrExp o
                  -> StreamProgram (Stream a)
                  -> StreamProgram (Stream o)

  StreamFilter    :: StrExp (i -> Bool)
                  -> StreamProgram (Stream i)
                  -> StreamProgram (Stream i)

  StreamWindow    :: StrExp (Stream o -> [Stream o])
                  -> StreamProgram (Stream o)
                  -> StreamProgram (Stream [o])

  StreamExpand    :: StreamProgram (Stream [o])
                  -> StreamProgram (Stream o)

  StreamMerge     :: [StreamProgram (Stream o)]
                  -> StreamProgram (Stream o)

  StreamJoin      :: StreamProgram (Stream a)
                  -> StreamProgram (Stream b)
                  -> StreamProgram (Stream (a,b))

  StreamFilterAcc :: StrExp (b -> i -> b)
                  -> StrExp b
                  -> StrExp (i -> b -> Bool)
                  -> StreamProgram (Stream i)
                  -> StreamProgram (Stream i)
```

Figure 7.1: An example translation of *StrIoT* operators into a GADT

Appendix A. Research Artefacts

A.1 StrIoT Source Code

StrIoT has been developed and is distributed as open-source software. The canonical location for the *StrIoT* source-code is GitHub [79].

The version of the software at the time of this thesis was 0.2.1.0.

A copy of the software as of 0.2.1.0 has been deposited as an artefact in the Newcastle University Research Repository [25].

A.2 NYC Taxi Data

[22] is an anonymised excerpt of New York City Taxi trip data for the month of January 2013, in CSV format. This data is used for the example stream-processing program in Section 6.2.1. The fields are described in Table A.1.

This data-set is an anonymised derivation of one produced by the DEBS 2013 Grand Challenge authors [43] which is no longer available. It, in turn, was an excerpt (1,999,999 records) from a larger data-set obtained by Chris Whong via Freedom Of Information Law (FOIL) [80]. The NYC Taxi and Limousine Commission now publish this data themselves [10].

A.3 Pebble Watch accelerometer data

[23] is 918,150 samples of accelerometer data recorded from a Pebble Smart Watch. The data is in CSV format. This data was used for our re-implementation of the stream-processing example from *PATH2iot* [58], described in Section 6.2.2.

The first field of each line is a time-stamp in UNIX epoch format with millisecond resolution. There then follows ten readings, each consisting of four fields: the X, Y and Z axis values, and the vibration sensor value (0 for for off, 1 for on). The ten readings are followed by 11 fields consisting only of zeroes.

The unit for the accelerometer values is thousandths of the gravitational constant G, with minimum/maximum values of -4000/4000 (range -4G to 4G).

The data was captured by Peter Michalák for his own PhD work and was graciously shared with me for mine. Peter has given permission for the data to be distributed under the terms of the Creative Commons Zero license [15].

Field	Description	Type	Notes
1	medallion ID	MD5 hash	32 zeroes for all records
2	license ID	MD5 hash	32 zeroes for all records
3	pick-up time	timestamp	Format `%Y-%-m-%-d %H:%M:%S`
4	drop-off time	timestamp	Format `%Y-%-m-%-d %H:%M:%S`
5	trip time	integer	Seconds
6	trip distance	fixed	Miles
7	pickup latitude	float	Degrees
8	pickup longitude	float	Degrees
9	drop-off latitude	float	Degrees
10	drop-off longitude	float	Degrees
11	payment type	text	`CRD` for card
12	fare amount	fixed	US dollars
13	surcharge	fixed	US dollars
14	MTA Tax	fixed	US dollars
15	Tip amount	fixed	US dollars
16	Tolls amount	fixed	US dollars
17	Total amount	fixed	US dollars

Table A.1: Description of the fields in the anonymised NYC Taxi trip data-set [22]

Appendix B. Haskell

Here we provide a brief description of the Haskell language, focussing on the aspects that are presented in this thesis. For more thorough treatments, we recommend [5], [62].

B.1 Functions

Functions are defined by declaring their name, any arguments, and an equals sign, followed by their definition. A function's type may be optionally declared with a type signature, prefixed by the function's name and `::`. The argument types are separated by `->` and terminated by the function's return type. Listing B.2 illustrates the definition and type signature for a simple 2-argument function.

```
f :: a -> b -> c
f x y = x + y
```

Listing B.1: A 2-argument function definition and type signature

The application of argument `x` to function `f` is written as `f x`. Function application reads right-to-left. Brackets can be used to adjust precedence, e.g. in `f (x y)`, `y` is applied to `x`, and the result to `f`.

Functions can be *partially applied*: Applying a single argument to a function of type `a -> b -> c` results in a new function of type `b -> c`.

Functions are *composed* with the `.` operator. A function `f` of type `a -> b`, composed with a function `g` of type `b -> c`, results in a function of type `a -> c`. It is common for several functions to be composed in sequence, e.g. `h . g . f`. The resulting order of evaluation reads right-to-left, i.e., `f` is applied first, followed by `g`, followed by `h`.

`&` is used as an *application* operator. It takes a left-hand value and applies it to the right-hand function. A sequence of applications reads left-to-right. E.g., `v & f & g & h` first calls `f` with the value of `v`, and then passes the result to `g`, followed by `h`.

A function with an arity of 2 can be written infix by surrounding it with back-quotes, e.g. `x `f` y` is equivalent to `f x y`.

An anonymous function is defined beginning with a backslash (chosen as the closest ASCII approximation to a lambda), followed by its arguments, the separator `->` and then the function body. E.g. `\x y -> x + y`.

B.2 Expressions

The expression `if x then y else z` evaluates `x`, of type `Bool`, and if the result is true, evaluates `y`, otherwise, `z`.

Expressions can be *bound* to a local name with `let`, e.g. `let two = 2 * x in (two, two)`.

A `case expression` is used to return different expressions based on a pattern. Patterns can match number literals. The catch-all pattern _ matches anything:

```
case v of
 1 -> "one"
 2 -> "two"
 _ -> "something␣else"
```

Listing B.2: A 2-argument function definition and type signature

B.3 Types

B.3.1 Basic Types

Basic Haskell types include `Int` (Integer); `Float` and `Double` (floating point numbers of different precisions); `Char` for text characters (denoted by single quotes, e.g. `'a'`) and `String`, a list of `Chars`, denoted by double-quotes, e.g. `"hello"`.

The special name `undefined` has type `a`, which means it can occur anywhere, but attempting to evaluate it results in a runtime error. For example, `map undefined []` returns the value `[]`, but attempting to apply `map undefined` to a non-empty list will provoke a runtime error.

B.3.2 Polymorphic Types

A polymorphic type is parameterized in terms of one or more other types.

The `Maybe a` type has two constructors: `Nothing`; `Just a`. It is used to represent computations that might fail.

Tuples, of which the most common variant is pairs, consist of two or more elements of different types. The expression `(1,'c',[])` represents a three-element tuple consisting of a number, a character and a list.

B.3.3 Lists

Haskell's lists are singly-linked lists. The empty list is denoted `[]` and a symbolic constructor `:` is used to inductively define a list featuring a head element and a tail list (which may be the empty list).

List literals may be defined using shorter comma-based syntax: `[1,2,3]` is equivalent to `1 : 2 : 3 : []`.

Common List Functions

`head` returns the first element of a list and `tail` everything except the first element. `last` returns the last element, and `init` all but the last. All four throw a runtime exception if they are called with the empty list.

`map f` applied to a list evaluates to a new list with the same number of elements, each element of which is the result of applying `f` (of type `a -> b`) to the corresponding element in the source list.

`filter p`, where p is of type `a -> Bool`, is applied to a list and evaluates to a new list of possibly-differing length, consisting of only those elements from the source list for which the predicate evalutes to `True`.

Folds such as `foldl` reduce a list (or other traversible data structure) to produce an output value.

`++` joins two lists together. `concat` joins a list of lists together. E.g. `concat ["hello ","world"]` returns `"helloworld"`.

`sort` returns a sorted list. `nub` returns a list with no duplicate items. The variants `sortBy` and `nubBy` require a higher-order function to use as the comparator.

B.3.4 Defining Types

An alias to an existing type is created with the `type` keyword,
E.g. `type Stream a = [Event a]`.

More complex types are introduced with the `data` keyword that has two forms: Sum-type syntax:

```
data StreamOperator = Map | Filter | Expand -- …
```

Listing B.3: Sum type syntax

Record syntax:

```
data StreamVertex = StreamVertex
{ vertexId   :: Int
, operator   :: StreamOperator
, parameters :: [ExpQ]
} -- etc.
```

Listing B.4: Record type syntax

In both cases the term on the left of equals defines a *type constructor* whilst the terms on the right define *data constructors*. The data constructors defined in the above examples are `Map`, `Filter`, `Expand` and `StreamVertex`. Type and data constructors all begin with capital letters.

For record syntax, the constituent fields define *accessor functions* which are used to retrieve the value from an instance of the parent type. For example, the above definition defines an accessor function `vertexId` with the type `StreamVertex -> Int`.

B.4 Pattern-Matching

Pattern matching is used to deconstruct instances of a data-type, to write function definitions that are specific to inputs matching a given pattern, and to bind portions of a data-type to variables that can be referenced in a function definition.

For example, consider the definition of a function `safeHead`:

```
safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead [] = Nothing
```

Listing B.5: Demonstration of pattern-matching

Here we have defined the function twice. The first definition is scoped to a pattern `(x:_)`, defined over the constructor for lists `(:)`. This definition will be used when an input is supplied which matches the pattern, in other words, only for non-empty lists. The pattern binds the variable `x` to one of the constructor's arguments, and that variable is referenced in the function definition on the right-hand side of the equals.

The second function definition is scoped to the pattern `[]`, the other list constructor. This will only match inputs which are the empty list.

As-patterns are used to label sub-components within patterns with variables. For example, the pattern in `wmInspect (e@(Event _ (Just i)):s)` labels the `Event` type with the variable `e`, which the right-hand side could reference in the same way as `x` in the previous example.

Appendix C. Supporting Code

This chapter collects together implementations of QuickCheck properties (See Section 4.6) to gain assurance that the rewrite rules designed in Section 4.5 are sound.

C.1 Preamble Code

```
{-# OPTIONS_GHC -F -pgmF htfpp #-}

module Main where

import Data.List (sort)
import Data.Char (isAscii)
import Test.Framework
import Striot.FunctionalIoTtypes
import Striot.FunctionalProcessing
import Striot.Simple
import Data.Maybe
import Data.Function ((&))
import Data.Time (UTCTime (..), secondsToDiffTime, Day (..))

main = htfMain htf_thisModulesTests
```

Most of the QuickCheck predicates receive an incoming stream as a parameter and compare the result of applying it to two expressions using the equivalence operator (==). We provide the following convenience operator ==== to encapsulate the common behaviour.

```
(====) :: Eq b => Show b => (a -> b) -> (a -> b) -> a -> Property
(====) f g s = f s === g s
infixl 7 ====
```

C.2 Sample operator parameters and input streams

Definitions of stream operator parameters, sample input streams, etc., used to support the QuickCheck implementations.

C.2.1 filter predicates

```
p = (>= 'a')
q = (<= 'z')
```

sample arguments for streamFilterAcc

```
-- increasing values only
accfn1 _ v = v
acc1 = '\NUL'
pred1 = (>=)
-- alternating values only
accfn2 _ v = v
acc2 = '\NUL'
pred2 = (/=)
-- even indices only
accfn3 acc _ = acc + 1
acc3         = 0
pred3  _ acc = even acc

-- pred/succ with wrapping
next :: (Eq a, Bounded a, Enum a) => a -> a
next a = if a == maxBound then minBound else succ a
prev :: (Eq a, Bounded a, Enum a) => a -> a
prev a = if a == minBound then maxBound else pred a
```

C.2.2 mapping parameters

```
f :: Char -> Char
f = next
```

examples of a streamScan arguments

```
counter = \c v -> c+1
scanfn  = counter
scanInit = 0 -- :: Int

scanfn2 b a = if a > b then 1 else if b == a then 0 else -1
scanAcc2 = 0 :: Int
```

C.2.3 sample window makers

```
wm = chop 3
```

C.2.4 test streams of characters

```
sA = [Event Nothing (Just i)|i<-['a'..]]
sB = [Event Nothing (Just i)|i<-['0'..]]
sC = [Event Nothing (Just i)|i<-['A'..]]
sI = [Event Nothing (Just i)|i<-[0..]]
sW = streamWindow (chop 2) sB
sWW= streamWindow (chop 3) sW
sw1 = streamWindow wm sA
```

C.2.5 Utility functions

```
-- utility functions for mapFilterAcc
accfn acc _ = acc+1
accpred dat acc = even acc

filterMaybe p v = if   p v
                  then Just v
                  else Nothing
```

C.3 QuickCheck properties

C.3.1 Operator fusion

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

filterFilterAccPre  = streamFilterAcc accfn1 acc1 pred1 .
   streamFilter q
filterFilterAccPost = streamFilterAcc
        (\a v -> if q v then accfn1 a v else a)
        acc1
        (\x a -> q x && pred1 x a)
prop_filterFilterAcc = filterFilterAccPre
                  ==== filterFilterAccPost

filterAccFilterPre  = streamFilter q . streamFilterAcc accfn1 acc1
   pred1
```

```
filterAccFilterPost = streamFilterAcc accfn1 acc1
                        (\x a -> pred1 x a && q x)
prop_filterAccFilter = filterAccFilterPre
                  ==== filterAccFilterPost

filterAccFilterAccPre  = streamFilterAcc accfn2 acc2 pred2
                       . streamFilterAcc accfn1 acc1 pred1
filterAccFilterAccPost = streamFilterAcc
    (\(x,y) v -> (accfn1 x v, if pred1 v x then accfn2 y v else y))
    (acc1, acc2)
    (\x (y,z) -> pred1 x y && pred2 x z)
prop_filterAccFilterAcc = filterAccFilterAccPre
                    ==== filterAccFilterAccPost

-- XXX would prefer two different mapping functions here
mapMapPre   = streamMap f . streamMap f
mapMapPost  = streamMap (f . f)
prop_mapMap = mapMapPre ==== mapMapPost

mapScanPre   = streamScan scanfn scanInit . streamMap f
mapScanPost  = streamScan (flip (flip scanfn . f)) scanInit
prop_mapScan = mapScanPre ==== mapScanPost

-- right-balanced pairs of merges can be fused
prop_mergeMerge :: [Stream Char] -> [Stream Char] -> Bool
prop_mergeMerge ss1 ss2 = streamMerge (ss1 ++ [streamMerge ss2]) ==
   streamMerge (ss1 ++ ss2)
```

C.3.2 Operator elimination

```
prop_mergeElim :: Stream Char -> Bool
prop_mergeElim s = s == streamMerge [s]
```

C.3.3 Other semantically-preserving rules

```
expandFilterPre   = streamFilter p . streamExpand
expandFilterPost  = streamExpand . streamMap (filter p)
prop_expandFilter = expandFilterPre ==== expandFilterPost

filterFilterPre'   = streamFilter q . streamFilter p
filterFilterPost'  = streamFilter p . streamFilter q
prop_filterFilter' = filterFilterPre' ==== filterFilterPost'
```

```
mapFilterPre   = streamFilter p . streamMap f
mapFilterPost  = streamMap f . streamFilter (p . f)
prop_mapFilter = mapFilterPre ==== mapFilterPost

mapFilterAccPre   = streamFilterAcc accfn 0 accpred . streamMap f
mapFilterAccPost  = streamMap f
                  . streamFilterAcc accfn 0 (accpred . f)
prop_mapFilterAcc = mapFilterAccPre ==== mapFilterAccPost

expandFilterAccPre accfn acc pred =
  streamFilterAcc accfn acc pred . streamExpand

expandFilterAccPost accfn acc pred
 = streamExpand
 . streamMap (reverse.fst)
 . streamScan (\(_,acc') a -> filterAcc accfn acc' pred a) ([],acc)

prop_expandFilterAcc1 :: Stream [Char] -> Property
prop_expandFilterAcc1 = expandFilterAccPre  accfn1 acc1 pred1
                  ==== expandFilterAccPost accfn1 acc1 pred1

prop_expandFilterAcc2 :: Stream [Char] -> Property
prop_expandFilterAcc2 = expandFilterAccPre  accfn2 acc2 pred2
                  ==== expandFilterAccPost accfn2 acc2 pred2

prop_expandFilterAcc3 :: Stream [Char] -> Property
prop_expandFilterAcc3 = expandFilterAccPre  accfn3 acc3 pred3
                  ==== expandFilterAccPost accfn3 acc3 pred3

mapJoinPre   = streamJoin sA . streamMap f
mapJoinPost  = streamMap (\(x,y) -> (x, f y)) . streamJoin sA
prop_mapJoin = mapJoinPre ==== mapJoinPost

scanJoinPre  = streamJoin sA . streamScan scanfn scanInit
scanJoinPost = streamScan
                 (\c (x,y) -> (x, scanfn (snd c) y))
                 (undefined, scanInit)
             . streamJoin sA
prop_scanJoin  :: Stream Char -> Property
prop_scanJoin = scanJoinPre ==== scanJoinPost

prop_mapMerge s1 s2 = streamMerge [streamMap f s1, streamMap f s2]
```

```
                  == streamMap f (streamMerge [s1,s2])

expandMapPre   = streamMap f . streamExpand
expandMapPost  = streamExpand . streamMap (map f)
prop_expandMap = expandMapPre ==== expandMapPost

expandScanPre  scanfn scanInit
  = streamScan scanfn scanInit . streamExpand
expandScanPost scanfn scanInit
  = streamExpand
  . streamScan (\b a -> tail $ scanl scanfn (last b) a) [scanInit]
  . streamFilter (/=[])
prop_expandScan :: Stream [Char] -> Property
prop_expandScan = expandScanPre  scanfn scanInit
             ==== expandScanPost scanfn scanInit

prop_expandScan2 :: Stream [Int] -> Property
prop_expandScan2 = expandScanPre  scanfn2 scanAcc2
              ==== expandScanPost scanfn2 scanAcc2

expandExpandPre    = streamExpand . streamExpand
expandExpandPost   = streamExpand . streamMap concat
prop_expandExpand :: Stream [[Char]] -> Property
prop_expandExpand  = expandExpandPre ==== expandExpandPost

prop_mergeMap s1 s2 = streamMap f (streamMerge [s1, s2])
                  == streamMerge [streamMap f s1, streamMap f s2]

scanWindowPre  = streamWindow wm . streamScan scanfn scanInit
scanWindowPost = streamScan
                   (\b a -> tail $ scanl scanfn (last b) a)
                   [scanInit]
                 . streamWindow wm

prop_scanWindow :: Stream Char -> Property
prop_scanWindow = scanWindowPre ==== scanWindowPost
```

C.3.4 Rules for specific types

```
mapWindowPre  :: Stream Char -> Stream [Char]
mapWindowPre   = streamWindow (chop 2) . streamMap f
mapWindowPost  = streamMap (map f) . streamWindow (chop 2)
prop_mapWindow = mapWindowPre ==== mapWindowPost
```

C.3.5 Re-ordering rules

```
filterMergePre  s1 s2  = streamMerge [ streamFilter p s1
                                     , streamFilter p s2 ]
filterMergePost s1 s2  = streamFilter p $ streamMerge [s1, s2]
prop_filterMerge s1 s2 = sort (filterMergePre s1 s2)
                      == sort (filterMergePost s1 s2)

expandMergePre  s1 s2  = streamMerge [ streamExpand s1
                                     , streamExpand s2 ]
expandMergePost s1 s2  = streamExpand (streamMerge [s1, s2])
prop_expandMerge :: Stream [Char] -> Stream [Char] -> Bool
prop_expandMerge s1 s2 = sort (expandMergePre  s1 s2)
                      == sort (expandMergePost s1 s2)

mergeFilterPre  s1 s2 = streamFilter p $ streamMerge [s1, s2]
mergeFilterPost s1 s2 = streamMerge [ streamFilter p s1
                                    , streamFilter p s2 ]
prop_mergeFilter :: Stream Char -> Stream Char -> Bool
prop_mergeFilter s1 s2 = sort (mergeFilterPre  s1 s2)
                      == sort (mergeFilterPost s1 s2)

mergeFilterPost2 s1 s2 = streamMap fromJust
                       $ streamFilter isJust
                       $ streamMerge [
                           streamMap (filterMaybe p) s1,
                           streamMap (filterMaybe p) s2
                         ]
prop_mergeFilter2 s1 s2 = mergeFilterPre   s1 s2
                       == mergeFilterPost2 s1 s2


prop_mergeExpand :: Stream [Char] -> Stream [Char] -> Bool
prop_mergeExpand s1 s2 = sort (streamExpand (streamMerge [s1,s2]))
                       == sort (streamMerge [ streamExpand s1
                                            , streamExpand s2 ])
```

Right-balanced `streamMerges` are order-preserving and tested by `prop_mergeMerge`. The following tests the other combinations: $\geq$0 streams prior to the inner merge, and $\geq$0 after the inner merge.

```
mergeMergePre'   a b c = streamMerge (a ++ streamMerge b : c)
mergeMergePost'  a b c = streamMerge (concat [a, b, c])
prop_mergeMerge' :: [Stream Char]
```

```
                  -> [Stream Char]
                  -> [Stream Char]
                  -> Bool
prop_mergeMerge' a b c  = sort (mergeMergePre'  a b c)
                       == sort (mergeMergePost' a b c)
```

C.3.6 Rules that reshape windows

In order to check properties of rules that reshape windows, we must ignore the difference in windows sizes. We expand the windows using `streamExpand` and extract the data from the `Event` wrappers using `unStream` in order to ignore timestamp differences caused by the reshaping. This limits our testing to ensuring the data and ordering of data within the windows is unchanged.

```
filterWindowPre  = streamWindow wm . streamFilter p
filterWindowPost = streamMap (filter p) . streamWindow wm

prop_filterWindow = unStream . streamExpand . filterWindowPre
               ==== unStream . streamExpand . filterWindowPost

filterAccWindowPre  = streamWindow wm
                    . streamFilterAcc accfn3 acc3 pred3
filterAccWindowPost = streamMap (reverse . fst)
                    . streamScan
                      (\ (_,acc) a -> filterAcc accfn3 acc pred3 a)
                      ([],acc3)
                    . streamWindow wm
prop_filterAccWindow :: Stream Int -> Property
prop_filterAccWindow =
 unStream . streamExpand . filterAccWindowPre ====
 unStream . streamExpand . filterAccWindowPost
```

Appendix D. Utilisation Example Program Variants

Here we present illustrations of the 4 program variants produced by the Logical Optimiser as applied to the Utilisation example program provided in Section 6.4.

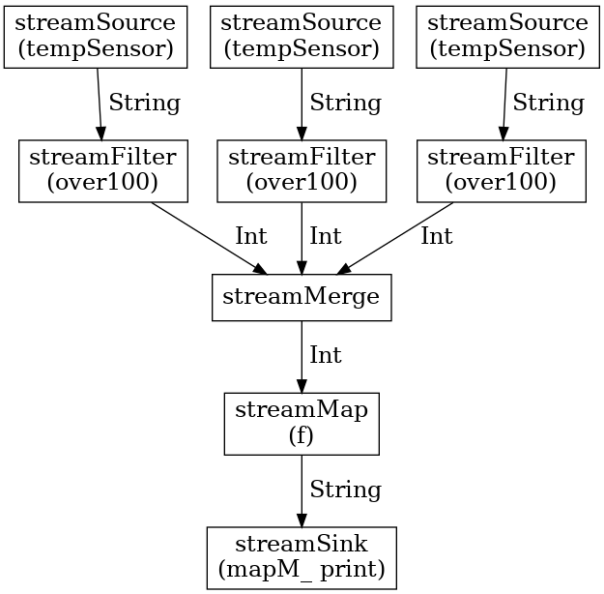


Figure D.1: utilVariants/1

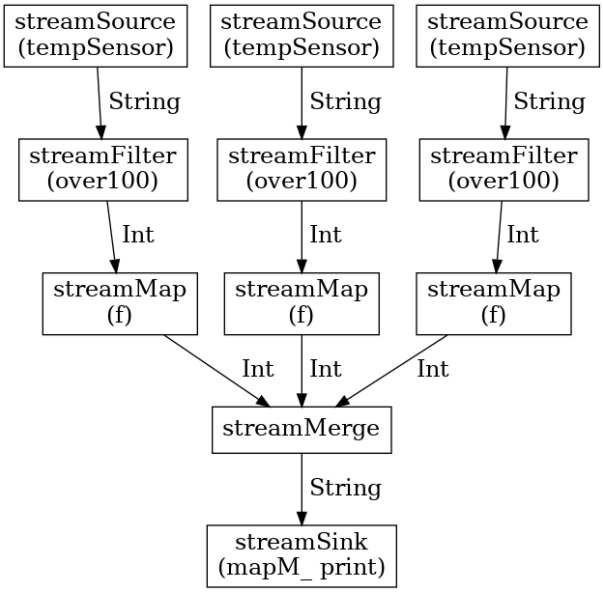


Figure D.2: utilVariants/2

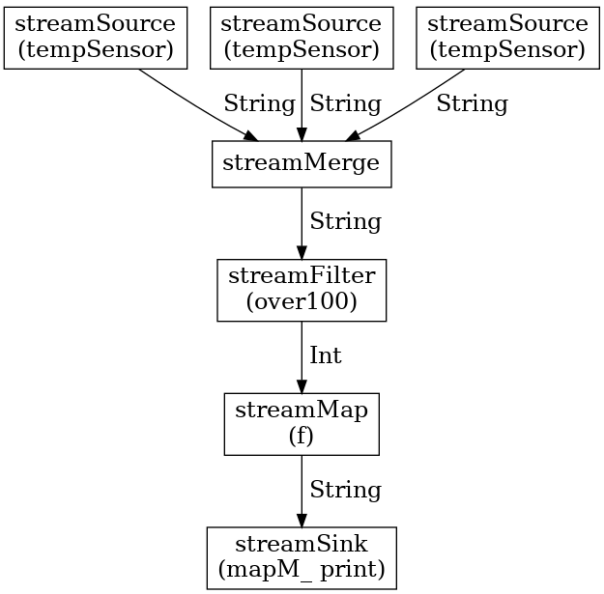


Figure D.3: utilVariants/3

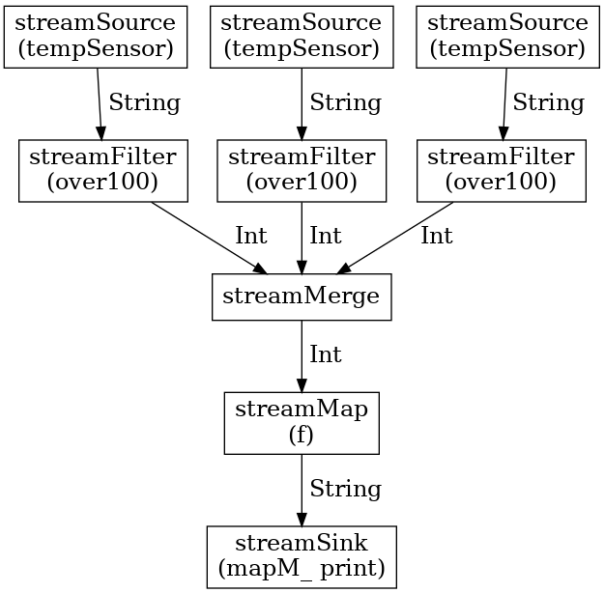


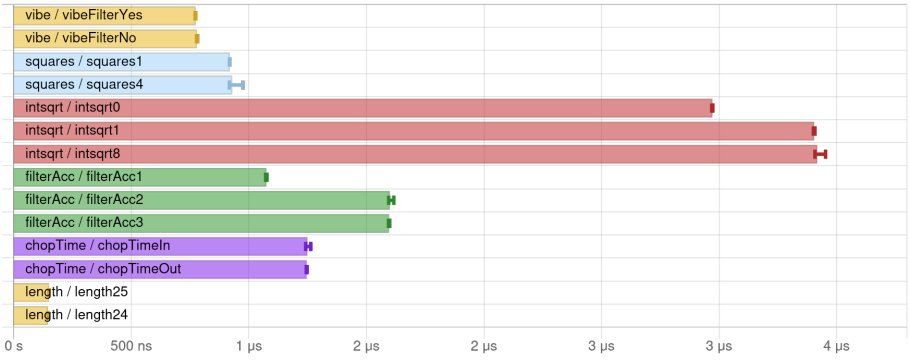
Figure D.4: utilVariants/4

Appendix E. Criterion Report

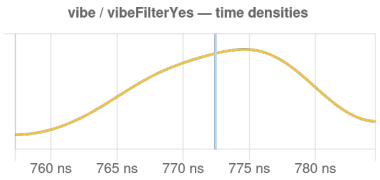
This is the report produced by Criterion [63] whilst benchmarking the operators within the wearable example (see Section 6.6.2).

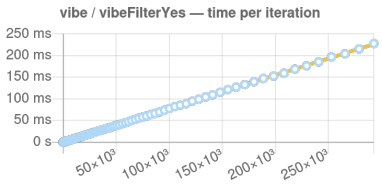
criterion performance measurements

overview



vibe / vibeFilterYes

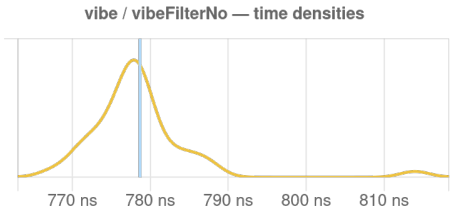


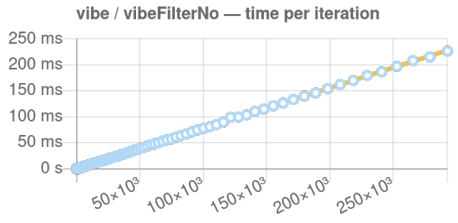


	lower bound	estimate	upper bound
OLS regression	771 ns	773 ns	775 ns
R^2 goodness-of-fit	1.00	1.00	1.00
Mean execution time	771 ns	772 ns	774 ns
Standard deviation	4.48 ns	5.33 ns	6.46 ns

Outlying measurements have no (0.461%) effect on estimated standard deviation.

vibe / vibeFilterNo

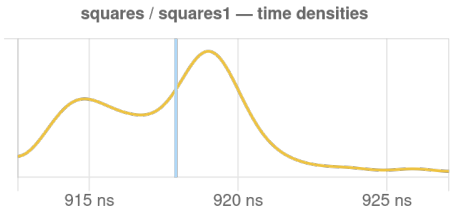


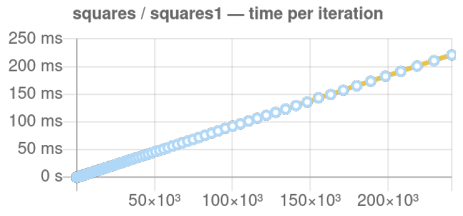


	lower bound	**estimate**	upper bound
OLS regression	776 ns	778 ns	780 ns
R^2 goodness-of-fit	1.00	1.00	1.00
Mean execution time	777 ns	779 ns	782 ns
Standard deviation	4.01 ns	7.04 ns	13.1 ns

Outlying measurements have a slight (6.11%) effect on estimated standard deviation.

squares / squares1

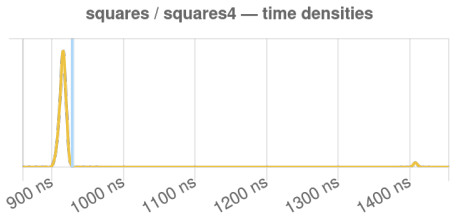


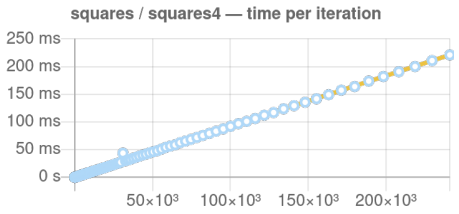


	lower bound	**estimate**	upper bound
OLS regression	918 ns	919 ns	920 ns
R^2 goodness-of-fit	1.00	1.00	1.00
Mean execution time	917 ns	918 ns	919 ns
Standard deviation	2.26 ns	2.72 ns	3.59 ns

Outlying measurements have no (0.469%) effect on estimated standard deviation.

squares / squares4

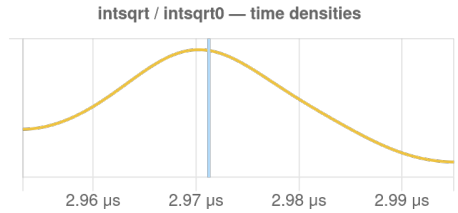


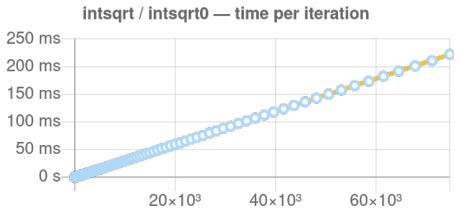


	lower bound	**estimate**	upper bound
OLS regression	916 ns	917 ns	919 ns
R^2 goodness-of-fit	0.998	0.999	1.00
Mean execution time	917 ns	929 ns	975 ns
Standard deviation	3.12 ns	75.4 ns	160 ns

Outlying measurements have a severe (84.1%) effect on estimated standard deviation.

intsqrt / intsqrt0

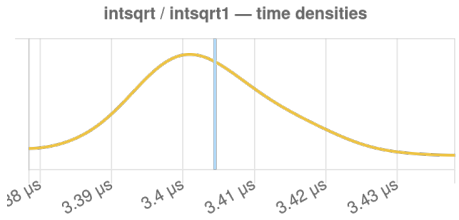


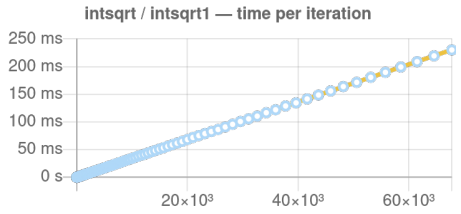


	lower bound	**estimate**	upper bound
OLS regression	2.97 µs	2.97 µs	2.97 µs
R^2 goodness-of-fit	1.00	1.00	1.00
Mean execution time	2.97 µs	2.97 µs	2.97 µs
Standard deviation	6.86 ns	8.25 ns	10.3 ns

Outlying measurements have no (0.529%) effect on estimated standard deviation.

intsqrt / intsqrt1

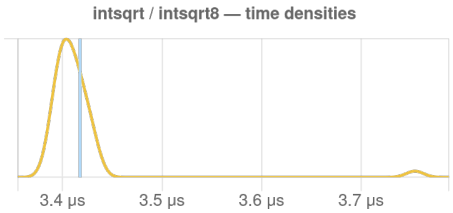


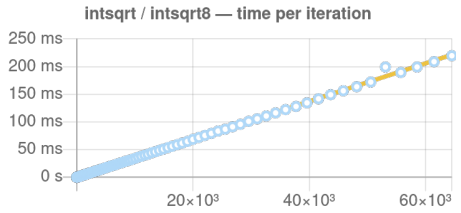


	lower bound	**estimate**	upper bound
OLS regression	3.40 µs	**3.40 µs**	3.40 µs
R^2 goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	3.40 µs	**3.40 µs**	3.41 µs
Standard deviation	9.30 ns	**11.3 ns**	14.7 ns

Outlying measurements have no (0.535%) effect on estimated standard deviation.

intsqrt / intsqrt8

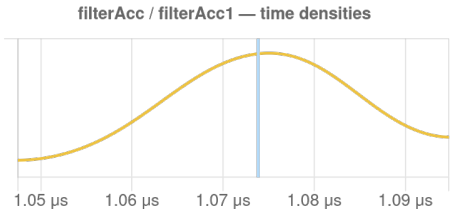


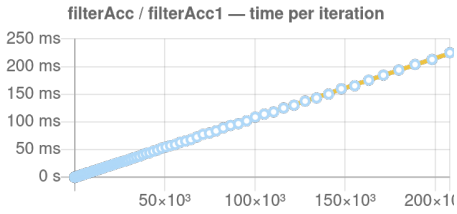


	lower bound	**estimate**	upper bound
OLS regression	3.40 µs	**3.43 µs**	3.48 µs
R^2 goodness-of-fit	0.998	**0.999**	1.00
Mean execution time	3.41 µs	**3.42 µs**	3.45 µs
Standard deviation	11.1 ns	**54.9 ns**	114 ns

Outlying measurements have a moderate (14.9%) effect on estimated standard deviation.

filterAcc / filterAcc1

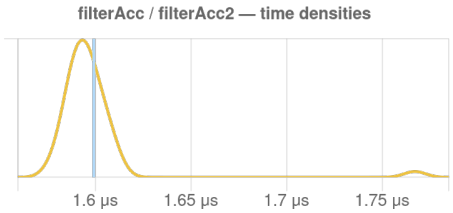


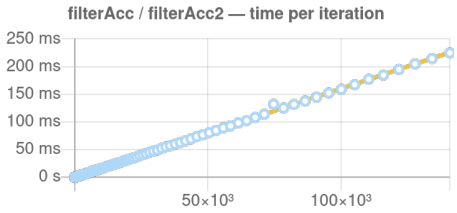


	lower bound	**estimate**	upper bound
OLS regression	1.07 µs	**1.07 µs**	1.08 µs
R^2 goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	1.07 µs	**1.07 µs**	1.08 µs
Standard deviation	7.52 ns	**9.20 ns**	11.4 ns

Outlying measurements have no (0.476%) effect on estimated standard deviation.

filterAcc / filterAcc2

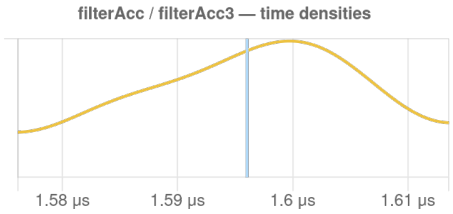


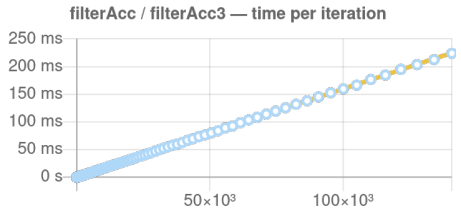


	lower bound	**estimate**	upper bound
OLS regression	1.59 µs	**1.60 µs**	1.61 µs
R^2 goodness-of-fit	0.999	**1.00**	1.00
Mean execution time	1.59 µs	**1.60 µs**	1.62 µs
Standard deviation	7.39 ns	**27.5 ns**	56.9 ns

Outlying measurements have a moderate (18.0%) effect on estimated standard deviation.

filterAcc / filterAcc3

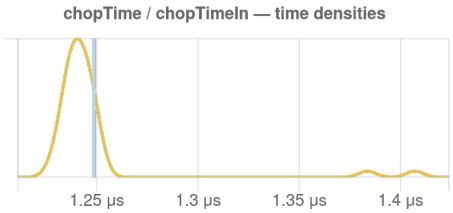


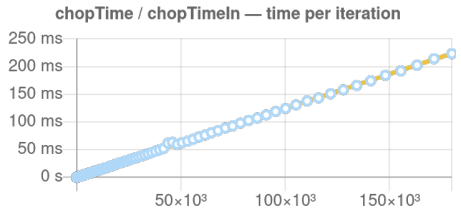


	lower bound	**estimate**	upper bound
OLS regression	1.59 µs	**1.59 µs**	1.60 µs
R^2 goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	1.59 µs	**1.60 µs**	1.60 µs
Standard deviation	7.28 ns	**8.37 ns**	9.92 ns

Outlying measurements have no (0.495%) effect on estimated standard deviation.

chopTime / chopTimeIn

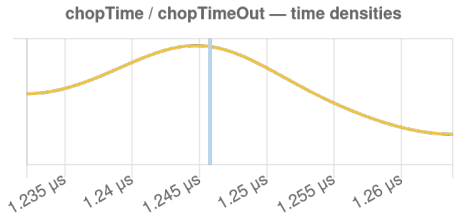


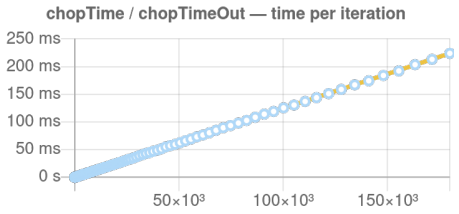


	lower bound	**estimate**	upper bound
OLS regression	1.24 µs	**1.24 µs**	1.25 µs
R^2 goodness-of-fit	0.999	**1.00**	1.00
Mean execution time	1.24 µs	**1.25 µs**	1.26 µs
Standard deviation	5.84 ns	**33.6 ns**	57.8 ns

Outlying measurements have a moderate (35.5%) effect on estimated standard deviation.

chopTime / chopTimeOut

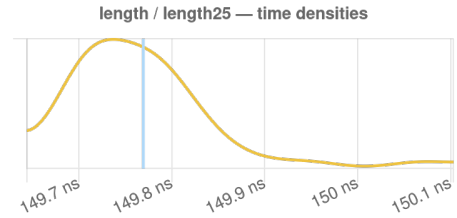


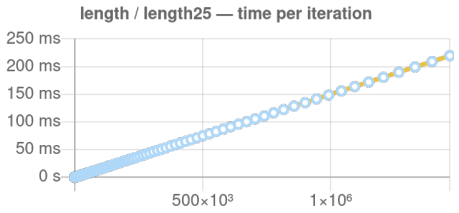


	lower bound	**estimate**	upper bound
OLS regression	1.24 µs	**1.24 µs**	1.25 µs
R^2 goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	1.24 µs	**1.25 µs**	1.25 µs
Standard deviation	5.78 ns	**6.92 ns**	8.43 ns

Outlying measurements have no (0.483%) effect on estimated standard deviation.

length / length25

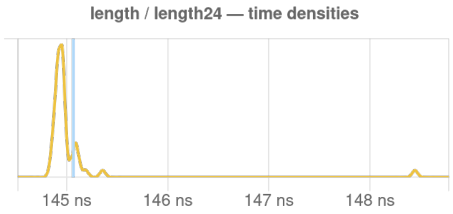


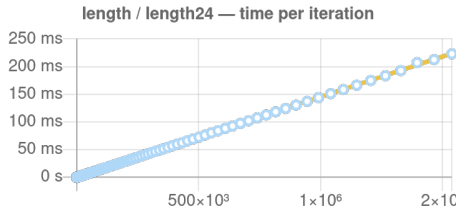


	lower bound	**estimate**	upper bound
OLS regression	150 ns	**150 ns**	150 ns
R^2 goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	150 ns	**150 ns**	150 ns
Standard deviation	52.9 ps	**74.4 ps**	120 ps

Outlying measurements have no (0.400%) effect on estimated standard deviation.

length / length24





	lower bound	**estimate**	upper bound
OLS regression	145 ns	145 ns	146 ns
R^2 goodness-of-fit	1.00	1.00	1.00
Mean execution time	145 ns	145 ns	145 ns
Standard deviation	88.0 ps	540 ps	1.13 ns

Outlying measurements have no (0.398%) effect on estimated standard deviation.

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x*-axis indicates the number of loop iterations, while the *y*-axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression estimate of execution time for a given number of iterations. Ideally, all measurements will be on (or very near) this line. The transparent area behind it shows the confidence interval for the execution time estimate.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R^2; goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R^2; should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of

the linear model.

- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be.

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, Second Edition. MIT Press, Jul. 2010, ISBN: 0262011530. [Online]. Available: https://web.mit.edu/6.001/6.037/sicp.pdf.

[2] C. Agrigoroae, "Lhc experiments are stepping up their data processing game," *CERN News*, Feb. 2022. [Online]. Available: https://home.cern/news/news/computing/lhc-experiments-are-stepping-their-data-processing-game.

[3] F. Baader and T. Nipkow, *Term Rewriting and all that*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.002.

[4] R. Bird, "A simple equational calculator," in *Thinking Functionally with Haskell*. Cambridge University Press, 2014, ch. 12. DOI: 10.1017/CBO9781316092415.

[5] R. Bird, *Thinking Functionally with Haskell*. Cambridge University Press, 2014, ISBN: 9781107087200. DOI: 10.1017/CBO9781316092415.

[6] E. Brady, *Type-Driven Development with Idris*. Shelter Island, NY: Manning Publications, 2017, ISBN: 9781617293023.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, pp. 28–38, 2015. [Online]. Available: https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1059537&dswid=-2208.

[8] A. Cattermole, "Run-time adaptation of a functional stream processing system," Ph.D. dissertation, 2022. [Online]. Available: https://theses.ncl.ac.uk/jspui/handle/10443/5761.

[9] A. Cattermole, J. Dowland, and P. Watson, "Run-time adaptation of stream processing spanning the cloud and the edge," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21, Leicester, United Kingdom: Association for Computing Machinery, 2021, ISBN: 9781450391634. DOI: 10.1145/3492323.3495627. [Online]. Available: https://eprints.ncl.ac.uk/280160.

[10] City of New York, *Tlc trip record data*. [Online]. Available: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[11] K. Claessen, "Shrinking and showing functions: (functional pearl)," in *Proceedings of the 2012 Haskell Symposium*, ser. Haskell '12, Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 73–80, ISBN: 9781450315746. DOI: 10.1145/2364506.2364516.

[12] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00, New York, NY, USA: ACM, 2000, pp. 268–279, ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. [Online]. Available: http://doi.acm.org/10.1145/351240.351266.

[13] G. Collins, *Io-streams*, 2013. [Online]. Available: https://hackage.haskell.org/package/io-streams.

[14] T. Cooper, "Performance modelling of distributed stream processing topologies," Ph.D. dissertation, School of Computing, Newcastle upon Tyne, UK, 2020. [Online]. Available: https://tomcooper.dev/files/Thomas_Cooper_Thesis.pdf.

[15] Creative Commons, *Cc0*. [Online]. Available: https://creativecommons.org/public-domain/cc0/.

[16] J. R. G. Cupitt, "The design and implementation of an operating system in a functional language (miranda)," AAIDX92465, Ph.D. dissertation, University of Kent at Canterbury, Canterbury, UK, 1989. [Online]. Available: https://kar.kent.ac.uk/94289/.

[17] D. Cutting, M. Cafarella, and B. Lorica, "The next 10 years of apache hadoop," *O'Reilly*, 2016. [Online]. Available: https://www.oreilly.com/content/the-next-10-years-of-apache-hadoop/.

[18] M. Dayarathna and S. Perera, "Recent advancements in event processing," *ACM Comput. Surv.*, vol. 51, no. 2, 33:1–33:36, Feb. 2018, ISSN: 0360-0300. DOI: 10.1145/3170432. [Online]. Available: http://doi.acm.org/10.1145/3170432.

[19] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA: USENIX Association, Dec. 2004. [Online]. Available: https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters.

[20] D. Dell'Aglio, E. D. Valle, F. V. Harmelen, and A. Bernstein, "Stream reasoning: A survey and outlook," *Data Sci.*, vol. 1, pp. 59–83, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:61155763.

[21] Docker Inc., *Docker compose*, 2013. [Online]. Available: https://docs.docker.com/compose/.

[22] J. Dowland, *Nyc taxi trip data 2013 - excerpt*, Mar. 2025. DOI: 10.25405/data.ncl.28691225. [Online]. Available: https://data.ncl.ac.uk/articles/dataset/NYC_Taxi_Trip_Data_2013_-_Excerpt/28691225.

[23] J. Dowland and P. Michalák, *Raw accelerometer data sampled from a pebble watch*, Mar. 2025. DOI: 10.25405/data.ncl.28691234. [Online]. Available: https://data.ncl.ac.uk/articles/dataset/Raw_accelerometer_data_sampled_from_a_Pebble_Watch/28691234.

[24] J. Dowland, P. Watson, and A. Cattermole, "Logical optimisation and cost modelling of stream-processing programs written in a purely-functional framework," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, ser. UCC '22, Vancouver, Washington, USA: IEEE, Dec. 2022. DOI: 10.1109/ucc56403.2022.00048. [Online]. Available: https://eprints.ncl.ac.uk/289943.

[25] J. Dowland, P. Watson, A. Cattermole, and S. Woodman, *Striot — functional stream processing for iot, Version 0.2.1.0*, Apr. 2025. DOI: 10.25405/data.ncl.28778102. [Online]. Available: https://doi.org/10.25405/data.ncl.28778102.

[26] M. Erwig, "Inductive graphs and functional graph algorithms," *Journal of Functional Programming*, vol. 11, no. 5, pp. 467–492, 2001. DOI: 10.1017/S0956796801004075.

[27] EsperTech, *Epl reference*. [Online]. Available: http://www.esper.espertech.com/release-5.2.0/esper-reference/html/epl_clauses.html#epl-intro.

[28] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *The VLDB Journal*, vol. 33, no. 2, pp. 507–541, 2024. DOI: 10.1007/s00778-023-00819-8. [Online]. Available: https://link.springer.com/article/10.1007/s00778-023-00819-8.

[29] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, *Drs: Dynamic resource scheduling for real-time analytics over fast streams*, 2015. arXiv: 1501.03610 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1501.03610.

[30] M. Garofalakis, J. Gehrke, and R. Rastogi, "Data stream management: A Brave New World," in *Data Stream Management: Processing High-Speed Data Streams*, M. Garofalakis, J. Gehrke, and R. Rastogi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 1–9, ISBN: 978-3-540-28608-0. DOI: 10.1007/978-3-540-28608-0_1. [Online]. Available: https://doi.org/10.1007/978-3-540-28608-0_1.

[31] GHC Team, *Glasgow haskell compiler*, 2022. [Online]. Available: https://www.haskell.org/ghc/.

[32] GHC Team, *Template haskell (ghc language extensions)*. [Online]. Available: https://downloads.haskell.org/ghc/9.0.1/docs/html/users_guide/exts/template_haskell.html.

[33] A. Gill, J. Launchbury, S. P. Jones, and S. Peyton Jones, "A short cut to deforestation," in *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, ISBN 0-89791-595-X, ACM Press, Jan. 1993, pp. 223–232. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-short-cut-to-deforestation/.

[34] G. Gonzalez, *Pipes: Compositional pipelines*, 2012. [Online]. Available: https://hackage.haskell.org/package/pipes.

[35] Graphviz authors, *Graphviz*, version 2.42.2-7, Mar. 18, 2025. [Online]. Available: https://graphviz.org/.

[36] P. Henderson and J. H. Morris Jr., "A lazy evaluator," in *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, ser. POPL '76, Atlanta, Georgia: ACM, 1976, pp. 95–103. DOI: 10.1145/800168.811543. [Online]. Available: http://doi.acm.org/10.1145/800168.811543.

[37] M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Akrivi Vlachou, "Stream processing languages in the big data era," *SIGMOD Rec.*, vol. 47, no. 2, pp. 29–40, Dec. 2018, ISSN: 0163-5808. DOI: 10.1145/3299887.3299892. [Online]. Available: https://doi.org/10.1145/3299887.3299892.

[38] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, Mar. 2014. DOI: 10.1145/2528412.

[39] *History of programming languages* (Association for Computing Machinery. ACM monograph series), eng. New York: Academic Press, 1981, ISBN: 0127450408. [Online]. Available: https://archive.org/details/historyofprogram0000hist/.

[40] P. Hudak, J. Hughes, S. Jones, and P. Wadler, "A history of haskell: Being lazy with class," English, in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, 2007, pp. 1–55. DOI: 10.1145/1238844.1238856.

[41] S. Hykes, *The future of linux containers, Pycon 2013 lightning talk*, 2013. [Online]. Available: https://www.youtube.com/watch?v=wW9CAH9nSLs.

[42] J. R. Jackson, "Jobshop-like queueing systems," *Management science*, vol. 10, no. 1, pp. 131–142, 1963, ISSN: 00251909, 15265501. [Online]. Available: http://www.jstor.org/stable/2627213.

[43] Z. Jerzak and H. Ziekow, "The debs 2015 grand challenge," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15, Oslo, Norway: ACM, 2015, pp. 266–268, ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2772598. [Online]. Available: http://doi.acm.org/10.1145/2675743.2772598.

[44] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 783–798, ISBN: 9781931971478.

[45] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *The Annals of Mathematical Statistics*, pp. 338–354, 1953.

[46] J. Kreps, "Questioning the lambda architecture," *O'Reilly Radar*, 2014. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/.

[47] J. Kreps, N. Narkhede, J. Rao, et al., "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[48] Kubernetes Inc, *Kubernetes*. [Online]. Available: https://kubernetes.io.

[49] S. Kulkarni et al., "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 239–250, ISBN: 9781450327589. DOI: 10.1145/2723372.2742788.

[50] H. Kumar, *Streamly: Beautiful streaming, concurrent and reactive composition*, 2017. [Online]. Available: https://hackage.haskell.org/package/streamly.

[51] P. J. Landin, "Correspondence between algol 60 and church's lambda-notation: Part i," *Commun. ACM*, vol. 8, no. 2, pp. 89–101, Feb. 1965, ISSN: 0001-0782. DOI: 10.1145/363744.363749.

[52] J. D. Little, "A proof of the queueing formula l= 2w," *Opns Res*, vol. 9, 1967.

[53] S. Marlow, "Haskell 2010 language report," The Haskell community, Tech. Rep., 2010. [Online]. Available: http://www.haskell.org/onlinereport/haskell2010/.

[54] N. Marz, *How to beat the cap theorem*, Accessed: 2025-03-07, 2011. [Online]. Available: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[55] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, ser. HOTOS'15, Switzerland: USENIX Association, 2015, p. 14. [Online]. Available: https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf.

[56] P. Michalák, "Automating computational placement for the internet of things," Ph.D. dissertation, 2020. [Online]. Available: https://theses.ncl.ac.uk/jspui/handle/10443/5596.

[57] P. Michalák, S. Heaps, M. Trenell, and P. Watson, "Automating computational placement in iot environments: Doctoral symposium," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 434–437.

[58] P. Michalák and P. Watson, "Path2iot: A holistic, distributed stream processing system," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, New Jersey, NJ, USA: IEEE, 2017, pp. 25–32. DOI: 10.1109/CloudCom.2017.35. [Online]. Available: https://eprints.ncl.ac.uk/242660.

[59] I. Mitrani, *Probabilistic Modelling*. Cambridge University Press, 1997. DOI: 10.1017/CBO9781139173087.

[60] A. Mokhov, "Algebraic graphs with class (functional pearl)," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2017, Oxford, UK: ACM, 2017, pp. 2–13, ISBN: 978-1-4503-5182-9. DOI: 10.1145/3122955.3122956. [Online]. Available: https://eprints.ncl.ac.uk/239461.

[61] S. A. Noghabi et al., "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017, ISSN: 2150-8097. DOI: 10.14778/3137765.3137770.

[62] B. O'Sullivan, J. Goerzen, and D. B. Stewart, *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008, ISBN: 978-0596514983. [Online]. Available: https://book.realworldhaskell.org/.

[63] B. O'Sullivan, R. Newton, and R. Scott, *Criterion*, version 1.5.13, Jun. 15, 2022. [Online]. Available: https://hackage.haskell.org/package/criterion.

[64] Open Container Initiative, *Open container initiative image format specification v1.0.0*, 2017. [Online]. Available: https://github.com/opencontainers/image-spec/releases/download/v1.0.0/oci-image-spec-v1.0.0.pdf.

[65] Open Source Initative, *The open source definition*. [Online]. Available: https://opensource.org/osd.

[66] S. Peyton Jones, A. Tolmach, and T. Hoare, "Playing by the rules: Rewriting as a practical optimisation technique in ghc," in *2001 Haskell Workshop*, ACM SIGPLAN, New York, NY, USA: ACM, Sep. 2001, pp. 203–233. [Online]. Available: https://www.haskell.org/haskell-workshop/2001/proceedings.pdf.

[67] R. Qasha, J. Cała, and P. Watson, "A framework for scientific workflow reproducibility in the cloud," in *2016 IEEE 12th International Conference on e-Science (e-Science)*, 2016, pp. 81–90. DOI: 10.1109/eScience.2016.7870888. [Online]. Available: https://eprints.ncl.ac.uk/232737.

[68] G. Rosinosky, D. Schmitz, and E. Rivière, *Streambed: Capacity planning for stream processing*, 2023. arXiv: 2309.03377 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2309.03377.

[69] T. Sheard and S. Peyton Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, New York, NY, USA: Association for Computing Machinery, Oct. 2002, pp. 1–16. [Online]. Available: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[70] N. Smallbone, M. Johansson, K. Claessen, and M. Algehed, "Quick specifications for the busy programmer," *Journal of Functional Programming*, vol. 27, Jul. 2017. DOI: 10.1017/S0956796817000090.

[71] M. Snoyman, *Conduit: Streaming data processing library*, 2011. [Online]. Available: https://hackage.haskell.org/package/conduit.

[72] M. Thompson, *Streaming: An elementary streaming prelude and general stream type*, 2015. [Online]. Available: https://hackage.haskell.org/package/streaming.

[73] A. Toshniwal et al., "Storm@twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 147–156, ISBN: 9781450323765. DOI: 10.1145/2588555.2595641.

[74] UP! Bridge the gap, *Up board datasheet v8.5*. [Online]. Available: https://up-board.org/wp-content/uploads/datasheets/UPDatasheetV8.5.pdf.

[75] P. Wadler, "Deforestation: Transforming programs to eliminate trees," in *ESOP '88*, H. Ganzinger, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 344–358, ISBN: 978-3-540-38941-5.

[76] J. Warren and N. Marz, *Big Data: Principles and best practices of scalable realtime data systems*. Manning, 2015, ISBN: 9781638351108.

[77] P. Watson, G. Lukyanov, and A. Cattermole, *Re-design of the event type, Issue discussion*, 2018. [Online]. Available: https://github.com/striot/striot/issues/22.

[78] P. Watson and S. Woodman, *Striot proof-of-concept*, 2017. [Online]. Available: https://github.com/striot/striot/blob/prior-jmtd/README.md.

[79] P. Watson, S. Woodman, J. Dowland, and A. Cattermole, *Striot*, 2025. [Online]. Available: https://github.com/striot/striot/.

[80] C. Whong, *Foiling nyc's taxi trip data*. [Online]. Available: https://chriswhong.com/open-data/foil_nyc_taxi/.

[81] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 423–438, ISBN: 9781450323888. DOI: 10.1145/2517349.2522737.

[82] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, San Jose, CA: USENIX Association, 2012, p. 2.

[83] N. Zhao, "Full-featured pedometer design realized with 3-axis digital accelerometer," *Analog Dialogue*, vol. 44, 2010. [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/pedometer-design-3-axis-digital-acceler.html.

Glossary Of Terms

bandwidth

A cost model in *StrIoT*'s Evaluator which estimates the bandwidth required for the size of the data-type and rate of emission between the first and second node within a deployment plan.

catalogue

The component of the declarative architecture (Section 2.4) describing the deployment environment and its properties.

container

A Linux application together with all of its software dependencies, packaged in a standarised format, such that a container runtime can start the application without requiring further software. Standardised by the Open Container Initiative (OCI) [64].

cost model

A mathematical model of a distributed stream-processing system, used by the Evaluator to measure properties of plans for filtering and ranking.

Deployer

The component of the declarative architecture (Section 2.4) responsible for deploying a plan and starting the stream-processing application.

eager

See eager evaluation.

eager evaluation

Also known as applicative-order evaluation: all arguments to procedures are evaluated when the procedure is applied [1].

equational reasoning

A technique for transforming functions through a process of substitution by applying laws or rules which express the equivalence of two expressions [5].

Evaluator

The component of *StrIoT* responsible for evaluating and ranking plans.

Event-Processing Language

Event-Processing Language (EPL) is an extended version of Structured-Query Language (SQL) where tables are replaced by streams [27].

functional programming

A declarative programming paradigm whereby the primary abstraction is that of the function, and programs are assembled by applying and composing functions.

GADT

Generalised Algebraic Data Types (GADTs) are a Haskell language extension permitting a data-type's individual constructors to have distinct types from each other.

GHC

The Glasgow Haskell Compiler (GHC) is the de-facto standard compiler for Haskell.

Haskell

A lazily-evaluated, purely-functional programming language. See Section 2.2.1.

lazy

See lazy evaluation.

lazy evaluation

An evaluation strategy where expressions are only evaluated at the point at which their value is required, such as when the value is to be printed to the screen [36]. Expressions which do not need to be evaluated during the execution of the program will not be: this allows the programmer to work with infinite expressions, such as the set of natural numbers. Contrast with eager evaluation.

Logical Optimiser

The component of the declarative stream-processing architecture (Section 2.4) responsible for optimising the stream-processing program. In *StrIoT*, the Logical Optimiser produces rewritten variant programs, to be costed by the Evaluator.

maximum node utilisation

A threshold for the total aggregated utilisation of all logical operators assigned to a node in a deployment plan.

node

In the context of *StrIoT*, a node is the term used to denote a computer instance, be that a physical or virtual machine, or an isolated container running within a container platform.

open-source

A model of software development whereby the source code to an application is available to consumers of the software. The source code to open-source is often freely available and open development is encouraged. The widely-accepted formal definition of open-source is the OSI Open-Source Definition [65].

partial functions

A function for which only a sub-set of the possible inputs (the range) are mapped to outputs (the domain). The behaviour of a partial function with an unmapped input is undefined. In practice, this may provoke a run-time error (e.g., when applying `head`, the function to return the first element of a list, to an empty list).

partition map

In the context of *StrIoT*, a mapping of operators from a stream-processing program to nodes within a deployment environment.

Partitioner

The component of *StrIoT* responsible for deriving the possible deployment plans for a stream-processing program. This component implements the Physical Optimiser from the original declarative stream-procesing model (Section 2.4).

Physical Optimiser

The component of the declarative stream-processing model (Section 2.4) responsible for producing the mapping of logical operators in a stream-processing program onto physical nodes in a deployment.

plan

In the context of *StrIoT*, a plan is a combination of a stream-processing program and a mapping of its constituent operators onto nodes for deployment (see partition map).

Prelude

A set of standard Haskell function and type definitions.

pure

Pure functions cannot perform *side-effects*, or, actions that cause a change of state.

purely-functional programming

A variant of functional programming featuring pure functions.

queueing network

In Queueing Theory, a queueing network is a collection of inter-connected servers.

Queueing Theory

the study of systems which feature a series of jobs which are serviced by one or more servers, such that jobs may have to wait in a queue before being processed [59].

rewrite rule

An equation asserting the equivalence of an expression on the left-hand side to that on the right. The left-hand term may be a pattern containing variables which serve as placeholders for arbitrary sub-expressions. The right-hand may contain references to the same variables. When applying the rewrite rule, those referernces are substituted for the bound values.

run-time

A packaging or collection of *StrIoT*'s functions and data-types included within deployed sub-programs. See also: Run-time Monitor.

Run-time Monitor

The component of the declarative stream-processing architecture responsible for collecting run-time information.

selectivity

The selectivity of a filter is a measure of the average proportion of accepted inputs.

server

In Queueing Theory, a server is an abstract representation of a device that provides a service.

service time

In Queueing Theory, service time is the mean average time required for a job to be serviced by a server.

steady state

In Queueing Theory, steady state describes the system where the number of jobs in the system is independent of the time.

Stream

A data-set which is possibly infinite and is never available all at once.

strongly typed

With a strongly-typed type system, every expression has a type and the compiler can reject invalid combinations of expressions based on incompatible types at compile time.

sub-program

A sub-program is a subset of a stream-processing program. A sub-program is defined when its constituent operators are assigned to the same node in a deployment.

term rewriting

Term rewriting is a technique for modifying a term by the successive application of rewrite rules.

total functions

A total function has a defined output for every possible input. Contrast with partial functions.

type system

A system of rules that assigns a property (type) to every term within a program. A type-checker can then ensure that the rules governing types hold. This can be used to catch programming mistakes, such as attempting to use a function defined for one type (such as addition for numerical types) on an term of a different type (such as a list).

window-maker

the parameter to `streamWindow` which describes the logic for producing each window.