

UNIVERSITÉ DE NAMUR  
Faculty of Computer Science  
Academic Year 2021–2022

**Data stream processing with quantitative  
regular expressions**

Romain Bruyère



Supervisor : \_\_\_\_\_ (Signed for Release Approval - Study Rules art. 40)  
Jean-Marie Jacquet

A thesis submitted in the partial fulfillment of the requirements for the degree of Master of Computer  
Science at the Université of Namur.



# Acknowledgments

I would like to thank my supervisor, Prof. Jean-Marie Jacquet. Thanks to him, I was able to work on a very interesting subject, which I really liked. He gave me great help and advices, whenever I needed and always at the right time, relying on his pedagogy and experience. Furthermore, his proofreading guided me very well throughout the work.

I would like to acknowledge and thank my family who, in addition to their encouragement during my thesis, supported me throughout all my studies.

Finally, I would like to thank my friends for the good times spent with them having fun and relaxing between these moments of intense concentration.



# Abstract

In a world in which IoT systems are more and more present, processing of data streams in an efficient way has become more and more important. Generic solutions are no longer sufficient to solve these problems, and specific models appear. Quantitative regular expressions, one of those, are standing out by a balance between language expressiveness and high efficiency. They use a flexible formal language for specifying complex numerical queries over data streams.

The purpose of this thesis is to analyze the solution exposed by this language in its context. The language evolution is detailed from its origin to its multiple implementations, explaining the theoretical concepts which had led to its conception. A new implementation is detailed. It provides flexibility in performance thanks to the use of a functional language.

## Keywords

Data streaming processing, Cost register automata, Quantitative regular expressions

# Résumé

Dans un monde où les systèmes IoT sont de plus en plus présents, le traitement optimisé des flux de données prend une place de plus en plus importante. Les solutions génériques ne suffisent plus à résoudre ces problèmes, et des modèles spécifiques prennent place. Parmi eux, les expressions régulières quantitatives se distinguent par un équilibre entre leur expressivité et une grande efficacité. Elles utilisent un langage formel flexible pour spécifier des requêtes numériques complexes sur des flux de données.

Le but de ce mémoire est d'analyser la solution exposée par ce langage dans son contexte. L'évolution du langage est détaillée depuis son origine jusqu'à ses multiples implémentations, en expliquant les concepts théoriques qui ont conduit à sa conception. Une nouvelle implémentation est détaillée. Elle offre plus de flexibilité au niveau des performances grâce à l'utilisation d'un langage fonctionnel.

## Mots-clés

Traitement de flux de données, Automate à registre de coûts, Expressions régulières quantitatives



# Contents

<b>Abstract</b>	<b>v</b>
<b>Résumé</b>	<b>v</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Listings</b>	<b>xii</b>
<b>Introduction</b>	<b>1</b>
<b>1 Context</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.2 Data streams and stream processing . . . . .	3
1.2.1 Streaming processing purpose . . . . .	4
1.2.2 Streaming data model . . . . .	4
1.2.3 Time management . . . . .	7
1.3 Road traffic monitoring . . . . .	8
1.4 Representative use case . . . . .	8
1.4.1 Theoretical capacity . . . . .	9
1.4.2 Peak hour factor . . . . .	9
1.4.3 Type of driver factor . . . . .	11
1.4.4 Heavy vehicle factor . . . . .	11
1.4.5 Practical capacity . . . . .	12
1.5 Testing strategy . . . . .	12
1.6 Road traffic monitoring use case: A naive solution . . . . .	13
1.6.1 Algorithm . . . . .	13
1.6.2 Performance . . . . .	14
1.6.3 Modifiability . . . . .	14
1.6.4 Testability . . . . .	15
1.6.5 Conclusion . . . . .	15
1.7 Application areas . . . . .	16
1.7.1 Medical domain . . . . .	16
1.7.2 Banking industry . . . . .	17
1.7.3 Network monitoring . . . . .	18
1.7.4 Diverse use cases . . . . .	18
1.7.5 Qualities and suitability . . . . .	19
1.8 Conclusion . . . . .	19
<b>2 Existing languages</b>	<b>21</b>
2.1 Introduction . . . . .	21
2.2 Quantitative regular expressions . . . . .	21
2.3 Formal languages . . . . .	24
2.3.1 Hybrid automata . . . . .	24
2.3.2 Timed automata . . . . .	25
2.3.3 Use case . . . . .	25
2.4 Temporal logic . . . . .	26
2.5 Pattern recognition . . . . .	28

2.6	Database languages . . . . .	29
2.7	String processing . . . . .	31
2.8	ReactiveX . . . . .	32
2.9	Conclusion . . . . .	34
<b>3</b>	<b>Quantitative regular expressions</b>	<b>35</b>
3.1	Introduction . . . . .	35
3.2	Regular functions . . . . .	35
3.3	Ranked alphabet . . . . .	37
3.4	Cost grammar . . . . .	37
3.5	Cost model . . . . .	38
3.5.1	Cost type . . . . .	38
3.5.2	Cost register automata . . . . .	39
3.5.3	Streaming String-to-Tree Transducers . . . . .	41
3.6	Regular combinators . . . . .	43
3.6.1	Structure . . . . .	43
3.6.2	Term . . . . .	44
3.6.3	Domain . . . . .	46
3.6.4	Evaluation . . . . .	48
3.7	Regular combinator types . . . . .	48
3.7.1	Atomic . . . . .	49
3.7.2	Iteration . . . . .	49
3.7.3	Concatenation . . . . .	50
3.7.4	Conditional . . . . .	51
3.7.5	Combination . . . . .	52
3.7.6	Apply . . . . .	53
3.7.7	Window . . . . .	54
3.7.8	Streaming composition . . . . .	54
3.7.9	Empty sequence . . . . .	55
3.8	Conclusion . . . . .	56
<b>4</b>	<b>Existing incarnations</b>	<b>57</b>
4.1	Introduction . . . . .	57
4.2	DReX . . . . .	57
4.3	StreamQRE . . . . .	59
4.3.1	Language . . . . .	59
4.3.2	Compilation . . . . .	60
4.3.3	Implementation . . . . .	66
4.3.4	Example . . . . .	67
4.4	NetQRE . . . . .	68
4.4.1	Language . . . . .	71
4.4.2	Compilation . . . . .	73
4.4.3	Implementation . . . . .	74
4.4.4	Evaluation . . . . .	74
4.5	Road traffic monitoring use case: A StreamQRE solution . . . . .	75
4.5.1	Algorithm . . . . .	75
4.5.2	Performance . . . . .	79
4.5.3	Modifiability . . . . .	79
4.5.4	Testability . . . . .	80
4.5.5	Conclusion . . . . .	80
4.6	Conclusion . . . . .	80
<b>5</b>	<b>LazyQRE</b>	<b>81</b>
5.1	Introduction . . . . .	81
5.2	Scala . . . . .	81
5.2.1	Language flexibility . . . . .	81
5.2.2	Higher-order functions . . . . .	83
5.2.3	Monads . . . . .	84
5.2.4	Case classes . . . . .	85



5.2.5	Pattern matching . . . . .	86
5.3	Language . . . . .	87
5.4	Compilation . . . . .	87
5.5	Implementation . . . . .	89
5.5.1	Structure . . . . .	89
5.5.2	Parameter types . . . . .	90
5.5.3	Result . . . . .	93
5.5.4	Methods . . . . .	93
5.5.5	Advantages . . . . .	94
5.5.6	Disadvantages . . . . .	95
5.5.7	Example . . . . .	97
5.6	Road traffic monitoring use case: A LazyQRE solution . . . . .	97
5.7	Conclusion . . . . .	100
<b>Conclusion</b>		<b>103</b>
<b>Bibliography</b>		<b>105</b>
<b>Appendices</b>		<b>108</b>
<b>A</b>	<b>LazyQRE</b>	<b>109</b>
A.1	Combinators evaluation classes . . . . .	109
A.2	Combinators configuration classes . . . . .	117
A.3	Structure class . . . . .	122
<b>B</b>	<b>StreamQRE, NetQRE, DReX</b>	<b>123</b>
B.1	StreamQRE . . . . .	123
B.2	NetQRE . . . . .	123
B.3	DReX . . . . .	123
<b>C</b>	<b>Case studies</b>	<b>125</b>
C.1	Traffic density monitoring . . . . .	125
C.1.1	Equivalence factor matrix . . . . .	125
C.1.2	Naive algorithm . . . . .	127
C.1.3	StreamQRE algorithm . . . . .	129
C.1.4	LazyQRE algorithm . . . . .	132
C.2	Accident monitoring . . . . .	134
C.2.1	Encryption simulation . . . . .	135
C.2.2	StreamQRE algorithm . . . . .	135
C.2.3	LazyQRE algorithm . . . . .	136
<b>D</b>	<b>Vehicle Stream Simulation App</b>	<b>139</b>
D.1	Frontend . . . . .	139
D.2	Backend . . . . .	142



# List of Figures

1.1	Example of the repetitive infinite cycle of events from a traffic light . . . . .	3
1.2	Transforming a stream into a signal . . . . .	4
1.3	Example of the turnstile model with parking entries . . . . .	5
1.4	Example of the cash register model with kilometers counter on motorway . . . . .	6
1.5	Example of the time series model with cars entering on highway . . . . .	6
1.6	Example of tumbling windows . . . . .	7
1.7	Example of sliding windows . . . . .	7
1.8	Theoretical capacity computing example . . . . .	10
1.9	Peak hour factor computing example . . . . .	10
1.10	Matrix for equivalence factor in ramps . . . . .	11
1.11	Heavy vehicle factor computing example . . . . .	11
1.12	Testing architecture . . . . .	12
1.13	Performance of the naive solution . . . . .	14
1.14	Heartbeat comparison from heart.org, May 2021 . . . . .	16
1.15	SSL Handshake schema, encryption is done at step 3, and decryption at step 4 . . . . .	18
2.1	QREs, as a schema, to apply functions on regular subset of a stream . . . . .	22
2.2	Example of timed automata . . . . .	26
2.3	Example string processing query . . . . .	31
2.4	Results of RxJS operator showcase example. . . . .	33
3.1	Example of a finite automata. . . . .	36
3.2	Example of a weighted finite automata. . . . .	36
3.3	Term as a tree . . . . .	38
3.4	Maximum kilometers traveled by hour as a term . . . . .	38
3.5	Integer interpretation of maximum kilometers term . . . . .	39
3.6	Cost register automata schema . . . . .	40
3.7	Cost register automata example . . . . .	41
4.1	Theoric split of a stream . . . . .	61
4.2	Concrete splits of a stream . . . . .	62
4.3	One only concrete split possible . . . . .	62
4.4	Multiple split combinators . . . . .	62
4.5	Initial situation . . . . .	63
4.6	P1 - The item is part of $g$ domain . . . . .	63
4.7	P2 - The item is part of $f$ domain, $g$ accept the empty string . . . . .	63
4.8	P3 - The item is part of $f$ domain, $g$ doesn't accept the empty string . . . . .	63
4.9	State of the stream when an accident just occurs . . . . .	64
4.10	State of the stream when an accident have already occured, but another can happen anytime soon . . . . .	65
4.11	State of the stream when a second accident just occurs . . . . .	65
4.12	Initial situation . . . . .	65
4.13	The word to parse starting at the left. . . . .	68
4.14	First part of the StreamQRE demonstration . . . . .	69
4.15	Second part of the StreamQRE demonstration . . . . .	70
4.16	Example of PSA with parameters . . . . .	73

4.17	Example of PSA with instantiated parameters as SAs . . . . .	73
4.18	Example of PSAs as a decision trees . . . . .	74
4.19	Representative example of the types of tokens making up the data stream . . . . .	75
4.20	Performance of the StreamQRE solution . . . . .	79
5.1	State of the stream when an accident have already occured, but another can happen anytime soon . . . . .	88
5.2	Performance of the StreamQRE solution with encrypted vehicles . . . . .	89
5.3	Partial class diagram of a combinator, separating its logic between two classes . . . . .	90
5.4	Partial object diagram of a combinator, separating its logic between two classes . . . . .	90
5.5	LazyQRE iter operation as a tree . . . . .	91
5.6	Class diagram defining the typing correspondence constraint between parent and child combinators . . . . .	92
5.7	Class diagram of an iter combinator . . . . .	92
5.8	Class diagram of an iter combinator with its operations . . . . .	94
5.9	Performance of the LazyQRE solution with encrypted vehicles data, no accident detected	95
5.10	Performance of the LazyQRE solution with encrypted vehicles data, an accident detected	96
5.11	The word to parse starting at the left. . . . .	97
5.12	First part of the LazyQRE demonstration . . . . .	98
5.13	Second part of the LazyQRE demonstration . . . . .	99
5.14	Performance of the StreamQRE solution . . . . .	100
5.15	Application that was built to do every simulation testing . . . . .	101

# List of Listings

Listing 1: Naive solution state . . . . .	13
Listing 2: Clause imbrication . . . . .	15
Listing 3: Array mutation for sliding window . . . . .	15
Listing 4: QREs, as Java function, to apply functions on regular subset of a stream . . . . .	22
Listing 5: Observer pattern using RxJS . . . . .	32
Listing 6: RxJS operators showcase . . . . .	33
Listing 7: QREs, as Java function, to count each element of the stream . . . . .	59
Listing 8: QREs, as Java function, to apply functions on regular subset of a stream . . . . .	64
Listing 9: A QRE implemented with StreamQRE to count the number of vehicles behind two trucks . . . . .	67
Listing 10: A QRE implemented with StreamQRE to compute the theoretical capacity . . . . .	76
Listing 11: A QRE implemented with StreamQRE to compute the peak hour factor . . . . .	77
Listing 12: A QRE implemented with StreamQRE to compute the heavy vehicle factor . . . . .	78
Listing 13: A QRE implemented with StreamQRE to compute the maximum lane capacity . . . . .	79
Listing 14: An example of Java code . . . . .	82
Listing 15: An example of Scala code . . . . .	82
Listing 16: Higher-order function example in Scala . . . . .	83
Listing 17: Custom higher-order function example in Scala . . . . .	83
Listing 18: Function composition example in Scala . . . . .	83
Listing 19: Currying example in Scala . . . . .	84
Listing 20: Option type in practice . . . . .	85
Listing 21: Option type as a tool for chaining operations . . . . .	85
Listing 22: Monads flattening operation . . . . .	85
Listing 23: Case class in Scala . . . . .	86
Listing 24: Wrong re-assignment statement in Scala . . . . .	86
Listing 25: Pattern matching on case classes . . . . .	86
Listing 26: Pattern matching on monads . . . . .	87
Listing 27: Extract of code with LazyQRE . . . . .	87
Listing 28: Extract of code with StreamQRE . . . . .	87
Listing 29: Combinator applying a simulation of a vehicle speed encryption . . . . .	88
Listing 30: Example of a simple QRE for explaining the LazyQRE combinator structure . . . . .	93
Listing 31: Monads integration for combinator results . . . . .	93
Listing 32: Example of an iter combinator automatically aggregating the results after 1000 iterations . . . . .	96
Listing 33: Example of an iter combinator automatically aggregating the results after 1 iteration . . . . .	96



# Introduction

In recent years, efficient processing of data streams has become critical in multiple domains, and the explosion of IoT systems only increases the need. Data streams are present in the medical field to control the health of patients, but also in the financial field where they play an important role in transaction processing. Remote communication, big data analysis, monitoring of various sensory information such as temperature, position, light or pressure and many others areas constantly benefit and rely on data streaming processing.

Data flows need to be processed in real time. This leads to the application of hard constraints. The data value often depends directly on the ability to perform actions or calculate metrics by analyzing it within a very short delay, therefore, the response time must be minimal. At the same time, the throughput of the data processing system must be high, and handle large amounts of data. Depending on the host system, the algorithm can be very restricted in terms of memory and energy consumption.

The structure of a data flow requires to distinguish the way in which the data is going to be processed in comparison with other types of data structures. If we can visualize a stream as sequential data, it should be kept in mind that the elements arrive one after the other, and the time lapses between two arrivals are often subject to significant variances. In the same perspective, unlike predefined sets of sequential data, it can be really tough to predict the size and the workload of a stream.

Hopefully, some of these problems are well known and long-standing. Not just one or two, but a multitude of languages and approaches already exist. Thanks to them, the challenge now is to gather all this knowledge and build on this solid ground to define the qualities that we want to see standing out in a data flow processing system. If this thesis exists, this is partly because all existing strategies have their advantages and disadvantages, and that there is still room for improvement. Before tackling the development of a solution, the scope of the problem that will be considered is defined. There are indeed many streaming models and, although the proposed solution is intended to be generic, an efficient solution cannot cover all of them at the same time. The scope being defined, a main part will be to identify the crucial aspects which characterize each existing languages inside this scope. After reviewing nearly each of the existing approaches found, one will be kept and retained for the rest of the thesis.

This language, retained as being the one of the most suitable for processing data streams for a large number of problems, is called Quantitative Regular Expressions, and is abbreviated as QREs. Even though no one can affirm that this is the perfect solution, it's convincing by being a fairly recent language whose goal is to lead to more expressive and efficient solutions. Rooted in formal language theories, QREs have seriously evolved in the last years to result in multiple implementations all different from each other. The multiple stages of the evolution are used as a tool to explain them step by step.

It would be very tricky to explain a language, its details and implementations, as well as its differences with other languages, just by settling on theory. To facilitate comprehension throughout the reading, multiple examples of data streaming processing systems illustrate the concepts. The examples are mostly built around traffic monitoring problems. Not only is this an area where data stream processing plays an important role, but it has also received too little attention in the literature covering quantitative regular expressions at the present time. In consequence, the traffic monitoring domain will be introduced very early in the thesis and kept along.

A large part of the thesis will be devoted to the details and explanations of the language itself. To give the reader a panoramic view of its potential, the writings will gradually retrace its evolution from the beginning of the conception until the last few years. The beginning of the explanations will focus on the construction of a new type of automaton, which is at the origin of the language. A demonstration of its

capabilities will help to illustrate the syntactical aspects of the language and its theoretical guarantees about complexity and expressiveness. All of this is, of course, always embellished with examples.

By going forward a little further in time, and in the evolution of QREs, it will be clear to notice that what was initially only very conceptual evolves to a concise set of instructions. Although there are many ways to deal with data streams, there are not many languages with instructions set designed especially for streaming processing. It's easy to realize that a data flow can be very difficult to manipulate with languages that are not suitable for this purpose. Fortunately this new instruction set is designed specifically for that. In this point in time, finally, the reader can witness the creation of a grammar that is easily readable and understandable by a developer without having to wonder how it was implemented. Until then, the missing feature is still an true implementation that can execute the stream processing on a real use case.

Several existing implementations of the QREs will be presented, in particular one in Java and another one in C++. The advantages and disadvantages of each will be reviewed. The crucial stake is obviously the way their compilers behave and, therefore, their impact on performance. At this stage, we will have seen in the preceding chapters the theoretical performance achievable on paper by the language, but this does not mean that each compiler sticks to them. Some compilers take more freedom than others by deviating slightly from the specifications. Currently only three different implementations exists and they do not serve the same purpose at all. Their compilers are completely different. Their differences make the comparison between them a bit difficult, but it suggests that there is still room for other solutions based on the same specifications.

This opportunity will be seized by proposing a new implementation. The choice of an implementation based on a functional approach having never been considered before, the new compiler is written in Scala within the framework of this thesis. Scala is a language quite similar to the well-know Java language but with a syntax giving the developer more freedom and giving him access to the use of functional patterns. The features specific to functional languages will be explained, and these will allow the reader to notice their interest in the context of data streaming processing.

Finally, a simulation of a real use case will assess the performance and usefulness of the different implementations. Within the scope of an example, a monitoring of the program during runtime will allow to check if each implementation correctly meets the characteristics of the theory.

The whole thesis really aims to escort the reader from very theoretical concepts to the development of a practical use case. The goal is to highlight the importance of preliminary theoretical research, to awaken the interest around the problem of the data stream processing, and to propose a real solution. That solution has much utility to be deepened in the future, in particular as regards the impact of a functional language.



# Chapter 1

## Context

### 1.1 Introduction

This first chapter introduces the reader to the concept of data streaming processing. The basic concepts and the different models of streams are illustrated. Several fields of application, in which data streams play an important role, are explained. Among these, many already have writings which give technical or theoretical solutions, with regard to the processing of data streams. An other domain that has received little attention until now, but which has a great interest, is introduced in this chapter. It will serve as a common thread throughout this thesis, starting with the presentation of a naive solution.

The naive solution is a program in Java conceived to solve the processing of a flow without any particular attention to generic coding or to the performances of the program. After this naive solution has revealed inherent weaknesses, existing solutions, as well as a new one, are introduced.

### 1.2 Data streams and stream processing

A data stream is a countably infinite sequence of elements and is used to represent data elements that are made available over time [38]. The term *stream* comes from water flows. The parallel between the two can easily be drawn. When you are in front of a stream, whether it is made of data or filled with water, you can neither see the start nor the end. What you're facing has no discrete beginning or end.

On one side of the flow, data *transmitters* produce data, and on the other, data *consumers* receive and process them. For example, phones are very greedy data consumers. All day, notifications received by the users are streamed from data feeds to the phones. The idea of a possibly infinite sequence of elements can easily be caught. If the phone is switched off, notifications will still be produced without interruptions, but only consumed the next time the phone is switched on. On the other side, a traffic light is a heavy data streaming producer which non-stop switches from a light to another. A brief example is represented in Figure 1.1.

In data streaming, some systems plays the role of data producers and consumers at the same time. For example, a traffic light can consume data in addition to its producer role. An increasingly common use case is the category of traffic light which self-regulates according to traffic density detectors. These detectors produces an input stream that changes the behavior of the traffic lights, then an output stream is emit to fire the lights.

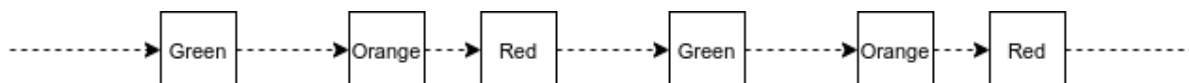


Figure 1.1: Example of the repetitive infinite cycle of events from a traffic light

In systems similar to traffic lights, the stream is infinite because the cycle indefinitely repeats itself, but obviously, sometimes, the transmission of a stream can be very short. A telephone call lasting a few

minutes can be treated as a data stream, even if it is only for the duration of the call. The one thing that must be kept in mind is that the final call time cannot be determined during the call. This makes its duration not countable, and possibly infinite at that time, which is a main characteristic of a stream.

The question now is why to care about streams, and what are their purpose. Fortunately, others before us have asked themselves the same question, and have found answers. First, a more abstract representation of data flows is needed to discern their usefulness.

### 1.2.1 Streaming processing purpose

Conceptually, a stream is a sequence of elements  $\sigma = \langle a_0, a_1, a_2, \dots, a_m \rangle$ . Each of these elements belongs to a *universe*  $[n] = \{0, \dots, n-1\}$ . The universe is the set of values that these elements can take. For example, considering that the stream contains only bits, its universe is the set  $\{0, 1\}$ . In scientific literature, the integer domain is usually considered, probably in the name of convenience, but any domain is eligible.

*Streaming processing* is all the processing steps that are carried out between the reception of the elements until the production of a result. There are a lot of different processing through which the elements can go (aggregations, filters, mapping, transformations,...). However, any processing are not suitable to any stream. Depending on the nature of the elements of the stream and their relations with each others, it is necessary to build the algorithms which are adapted to them. Generally, the result is a single value or a vector of values. Before categorizing the flows according to their elements type, let's see a concrete example of a processing.

**Example 1.2.1.** Let's consider the monitoring of cars entering a highway each hour. The goal is very simple. When each new hour begin, the data stream processing needs to compute the total number of car tokens received during the preceding hour. When starting the algorithm, the output is an empty vector, and a new value is added at each new hour.

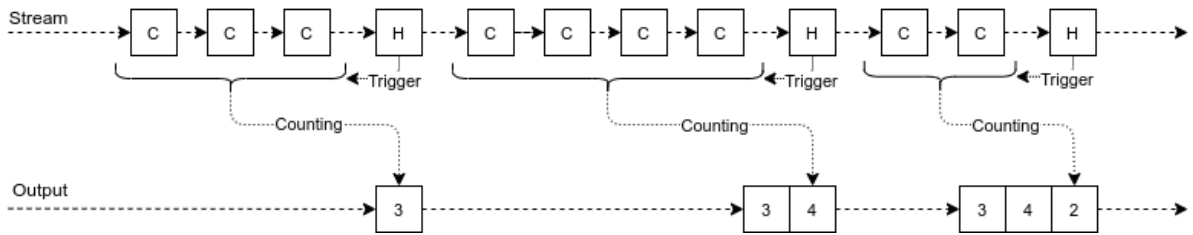


Figure 1.2: Transforming a stream into a signal

Figure 1.2 describes this situation during three successive hours. Concerning the graphical representation of the figure, the items of the stream are boxed. They contain the letter C for each car, and the letter H representing each new hour. The stream process the cars one by one. In this example, because the result is a vector containing their total number, it would be  $[3, 4, 2]$ .

This representation gives the general behavior of what data flow processing produces. However, as said before, the design of a streaming algorithm does not focus only on the transformation, but also on the purpose behind. Processing a giving stream without any purpose would not make sense. From the general model presented above, the streams can be categorized according to their processing interest. To understand these categories, it is necessary to dive a little deeper in the characteristics of the flows.

Two central concepts, involving the nature of the stream items, define the way in which they can be processed and their purpose:

- The streaming data model;
- The relation between the stream items and the time.

### 1.2.2 Streaming data model

The streams are categorized into two very distinct sets due to their structure.

The first one is made of unstructured data gathered from many sources, namely events shower or events clouds. In this case, the stream contains arbitrary content, and events are sometimes partially ordered. These events are usually the result of observing an existing environment or system. Recent studies have worked on these categories of events [27], they will not retain our interest in this thesis.

The second one is the one which is particularly interesting in our case. In contrast, these are structured streams. Divided into several models, these streams are distinguished by the relationships that their elements have with each other.

**Example 1.2.2.** For example, let's consider a stream where each element is a fuel consumption information. Imagine these describe the evolution of the fuel consumption of a single car during a trip. Each received number is the total of the consumption at some point in time. The tendency will be to keep only the last value, that will be updated throughout the trip, making sure that the updates arrive in temporal order. On the contrary, if the indices represent the consumption of each car passing at a precise location, the need could perhaps be to keep a vector of these values to see their evolution over time, and the concern will then be to pay attention to the consumption in memory of this growing vector.

In these two examples, the same type of data is emitted but for different purposes. Therefore the points of attention in the processing diverge, temporal order for one and memory consumption for the other. They are distinguished by what is called their underlying *signal*. Each stream describes an underlying signal. This signal is an essential part of the interest found in reading the stream. The signal defines the relation which the elements have between them and makes it possible to categorize the flows. To retrieve the signal from a stream, a function must be used. In the first example, the function would give us the current consumption of the car, and in the second, would give us a vector containing the consumption of each car.

In reference documents [29, 30, 38], they distinguish three different structured models, each with its own type of signal. To understand each of them, the different functions describing the transformation of flows into output signals are illustrated below. It should be noted that the signal is only used to categorize the flows. In the preceding example, by receiving the evolution of the fuel consumption of cars, a series of other operations can be done according to this category.

### Turnstile model

The name of this model is a bit inspired by the busy subway stations of New York. In this context, the turnstile keeps track of who has entered and exited from the station [41]. This continuous flow of people passing through the subway is observed and processed. The metaphor implies that each element is an incremental, or decremental, operation on the underlying signal.

**Example 1.2.3.** For example, this model can be applied to the capacity management of parking slots. The beginning of a stream processing for this situation is shown in Figure 1.3. When a car arrives, a signal element associated with its license plate number is incremented to 1, and decremented to 0 when it leaves the parking lot.

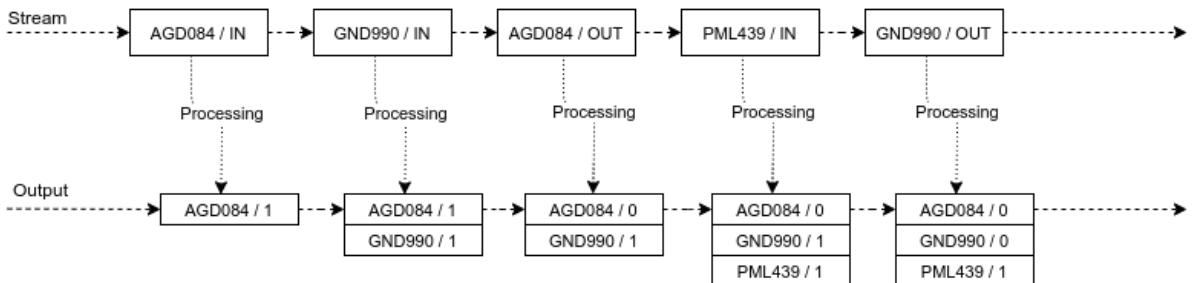


Figure 1.3: Example of the turnstile model with parking entries

The range of values are not limited between 0 and 1, and can extend over all the domain of integers. However, the algorithms in which the signal never contains negative numbers, like the one presented above, are a specialization called strict turnstile model.

In fact, like all streaming models, this one is not even restricted to integers only. More conceptually, this streaming model groups together all the streams in which each new item read is an update of the underlying signal. The model is very general and can adapt to many systems. One main characteristic is that the signal is a vector and its maximum size is the size of the universe of the stream, or in other words, the domain of the streaming items.

Another way to see this model is to visualize it as updates in a database table. This is actually the algorithm used to operate in a streaming fashion on traditional database, where inserts, deletes and updates are the analogy of incrementation and decrementation in the integers domain.

### Cash register model

The algorithms grouped together in this category are similar to those of the turnstile model, except that they only accept incremental operations. These algorithms are designed specifically for aggregations.

**Example 1.2.4.** A brief example of their usefulness is the counting of kilometers traveled on toll motorways. In this way, at the motorway exit, motorists can pay a suitable flat rate according to their distance. A situation within a processing aggregate the number of kilometers of a single car is represented in Figure 1.4. For instance, the second output item contains the total of the two first kilometer values from the input stream, and the third output contains the totals of the three.

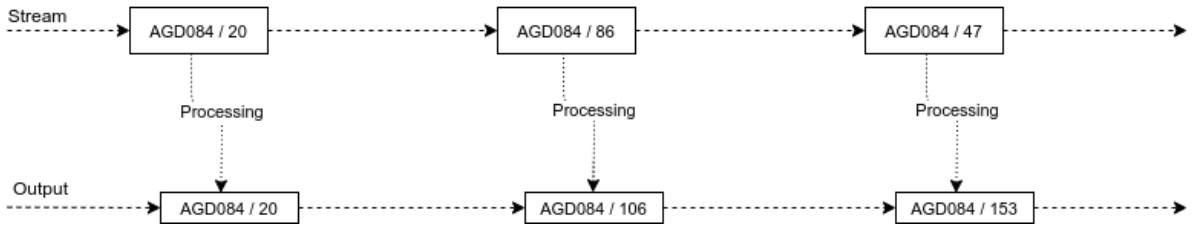


Figure 1.4: Example of the cash register model with kilometers counter on motorway

This model is well separated from the previous one because optimized algorithms are designed specifically for it. Another example of the model, outside the domain of integers, are databases that keep a growing history of their relations.

### Time series model

This last model treats each element of the stream as an independent temporal information. The time-series data is a statement of the observations made at different points in time. This model is the most basic among the others. For this reason, it is also called the vanilla model.

**Example 1.2.5.** The input stream shown in Figure 1.2 is a very good example. It describes the entry of cars on a highway in hourly time intervals. In Figure 1.2, a counting operation was performed directly on the stream. Instead, if the goal is simply to represent the underlying signal, the processing shown in Figure 1.5 can be used.

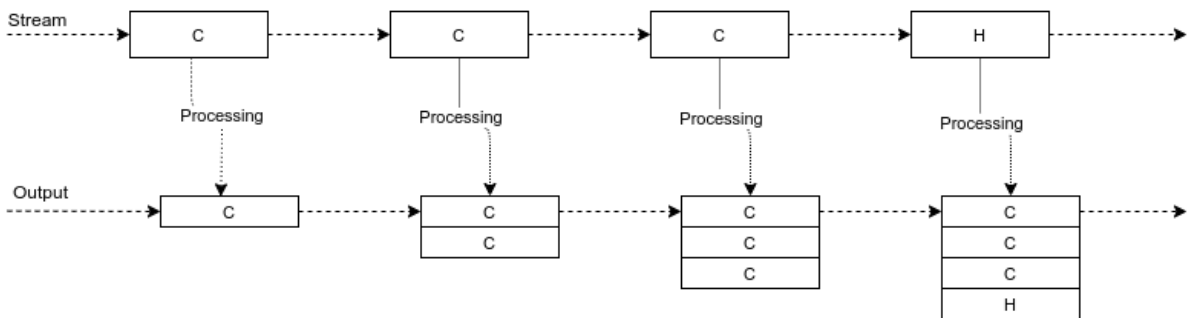


Figure 1.5: Example of the time series model with cars entering on highway

The resulting model is a vector whose size keeps increasing. Usually, the vector is unbounded which justifies the interest, as in Figure 1.2, to start processing directly and reducing its size to avoid facing memory problems. The algorithms designed throughout this thesis will be designed for this model in particular, which makes understandable the interest within the thesis for efficient memory management.

The three different models having been seen, it is time to dwell a little more on time management.

### 1.2.3 Time management

Time management is a crucial issue in the processing of data flows. When dealing with a stream, a major value is added if its elements are already ordered according to time. Until now, the elements of all the streams that were presented in the previous pages arrived in chronological order. However, this is not always the case. To understand this, two notions of time with different semantics must be considered [38]:

- **Event time:** The instant at which the element is produced;
- **Processing time:** The instant at which the stream processing system starts processing the element.

Ideally, these two time measures are equal, or very close, for all streaming elements. Unfortunately, in absolute terms, some noticeable delays between the two often happens. This could be because of latency on the network, or of the production of elements which is made from different places or from different transmitters. If they cause a disorder, these delays generate the need to reorder the elements within the processing system, or at least to provide in advance a mechanism to avoid this problem. To solve this, in many streams, each element is attached to the timestamp describing its event time.

The second aspect on which time has influence are the *windows*. In streaming processing, the windows are bounded portions of elements over an unbounded stream. Some computations are simply impossible to execute over all the elements of the stream, therefore their execution spans over a predefined limited section. Windows are usually delimited by time or elements count.

The most standard types of windows are represented in Figure 1.6 and Figure 1.7 with a simple stream of integers. The tumbling windows accumulates multiple elements and repeats at a non-overlapping interval. The sliding window continuously advance with the arrival of new items.

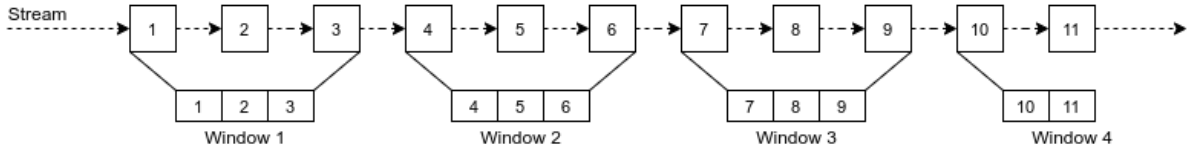


Figure 1.6: Example of tumbling windows

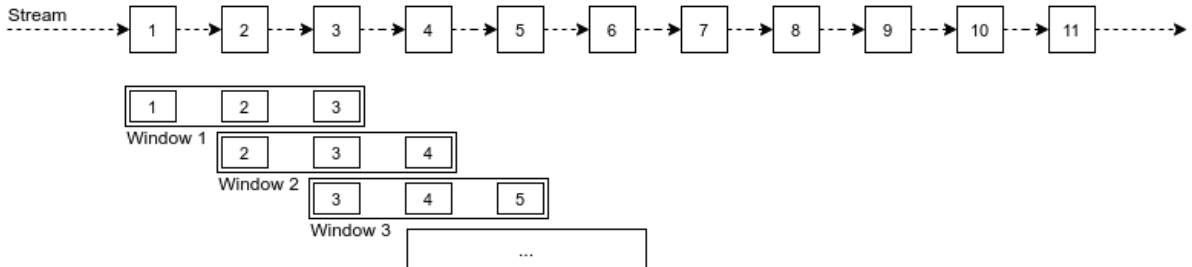


Figure 1.7: Example of sliding windows

Throughout this thesis, the concepts of window and time will be very important. The different ways for reordering elements will not be explained. Instead, the focus will be set on the processing of time series data in a window in which the elements are already in a specific order. Any type of window is compatible with the algorithms that will be presented, but these will generally be neither time-based nor

count-based as usual. Instead, the windows will be delimited by *regular patterns*, which will be explained later in the thesis. Because these regular patterns are victims of certain limitations, they can easily be enhanced by count-base windows to bypass their restrictions.

Now that the global scope of streaming processing is defined, a concrete example is proposed to evidence the purpose of creating a new language. Before that, it is necessary to introduce a field of application which will serve as a common thread for the examples throughout this thesis.

### 1.3 Road traffic monitoring

The streaming techniques that are presented in the next sections are intended to be adaptive to all kinds of situations. Multiple application areas in which they are ideally suited for problem solving will be introduced. In the majority of the articles that have been written on the problems appropriate to this context, the authors have used examples to illustrate the functioning of the concepts introduced. Among these are the banking transaction stream [10], network monitoring [50], arrhythmia detection [4] and some others. However, to make the thesis easier to read, the vast majority of examples will focus on one single field, the road traffic monitoring.

There is a good reason to choose this domain over the others. Even if there has been a lot of study in data flow processing about tracking vehicles and regulating traffic flow, only very few have focused on genericity and modifiability while maintaining good performance. However, depending on the types of road, periods, bad weather or other climatic conditions, the algorithms are required to change and evolve very often as well as to communicate with other programs. In addition, the monitoring tasks are known to demand a high level of performance due to the heavy number of vehicles during rush periods [14]. The choice of this field of expertise is therefore quite appropriate.

Among all the programs that already exist to monitor road traffic, many are proprietary software. They are built specially for real-time traffic monitoring, events capture and traffic control. They extract traffic data from diverse sensors such as camera streams. Many use LIDAR (Laser Imaging, Detection, and Ranging) technology. After processing the stream they alert traffic managers, and try to provide enough information in real-time to help them reducing the impact of incidents and congestion [31, 46].

In some systems, managers can create monitoring criteria to configure the applications on a case-by-case basis. These then filter events on the fly, and send them to controllers immediately which makes manual work easier afterwards. The problem is that the criteria are sometimes very difficult to modify without losing efficiency.

The scope of road problems can be very wide and some are quite complex. For example, the optimization of road traffic density to avoid traffic jams is a major problem since many years. It is among others necessary for applications such as automatic tolling systems. The goal can be to optimize the number of kiosks open during the day, and their type, according to the density of traffic. This could also lead to decision like road works planning. Others problems are quite simple, as it was shown previously a simple traffic light can be monitored using streaming algorithms.

The algorithms that will be presented show how to build programs in a modular way so that it is easily adaptable to different use cases, and can evolve over the years. The next section introduce a typical problem in the traffic monitoring domain.

### 1.4 Representative use case

So far, all of the streams that have been presented in the precedent examples are fairly straightforward. It only takes a few operations to extract a result. Only few constraints are present, both in terms of required performance and difficulty of design. Unfortunately, this is not often the case. In traffic monitoring like many other fields of application, specific tools have been developed due to the complexity of processing flows.

Before exploring these different areas and diving into the development of a solution, here is an example to give a general idea of a data streaming processing problem in practice. The problem presented is relatively simple in term of performance needed, but remains representative of the complexity that may be encountered in standard applications. It is introduced in an agnostic way to any form of implementation.

The example takes place in cross section design for bi-directional road tunnel. The document used for the given procedure is a report produced by PIARC, the World Road Association [47].

The goal is to compute the maximum car capacity for a tunnel section by observing the traffic speed and densities. The objective in analyzing such metrics is to assess the condition and safety of tunnels. The results then make it possible to improve the design of new tunnels and the maintenance of old ones.

Although the focus here is on the specific case of tunnels, the vast majority of the procedure followed does not depend on the type of road and could also apply for open roadways. These procedures can then also assess the impact of an unwanted blocking obstacle such as an accident or, on the contrary, planned such as automatic tolling systems, where the goal can be to optimize the number of kiosks open during the day, and their type, according to the density of traffic.

Without going into too much details, the following explanations transcribe verbatim and summarize the steps to follow to do the calculation.

First, several prerequisites are necessary to obtain consistent results. The procedure requires performing the calculation by direction of travel because the traffic flow can be very variable depending on the side. In addition, if the characteristics of the different sections on the same side of the tunnel defer, for example by the presence of a climbing lane in a part of the tunnel, the different sections should be studied separately. If several lanes are present on the same road in the same direction, only the busiest is considered when the tunnel is under conditions of high traffic intensity. Furthermore, to be precise, all tests should be performed under conditions close to the situation of full capacity, or saturation, because that's the easiest way to evaluate the maximum tunnel capacity.

To find the practical capacity of a road in number of vehicles per hour, the calculations are based on four factors:

- The theoretical capacity
- The peak hour factor
- The type of driver factor
- The heavy vehicle factor

Each factor will be explained. They should be computed separately, then the final formula bring them together while computing the final metric.

#### 1.4.1 Theoretical capacity

The theoretical capacity is computed using the mean speed of light vehicles. This is the basic factor for estimating the maximum traffic flow. It is subsequently adjusted by the other three factors to account for different types of vehicles and drivers.

The mean speed ( $FFS$ ) is quantified directly by observation, by averaging the speed of vehicles in circulation. The speed information data received are calculated by a section radar to take into account the vehicle speed over the entire tunnel. Once the data has been gathered, a standard calculation gives the theoretical capacity per traffic lane ( $TC_{pl}$ ):

$$TC_{pl} = FFS * 10 + 1200 \text{ (in passenger cars per hour per lane)}$$

It should be noted that there is one more prerequisite required for computing this factor. The observed capacity on the measured traffic lane should not be more than 2200 passenger cars per hour, otherwise the result cannot be considered reliable.

Figure 1.8 shows a simplified representation of a stream, receiving vehicles data containing for each vehicle their type and their speed according to the time series model. The type of vehicle is not necessary to calculate the theoretical capacity but will be for other factors. The result must be updated each time a new item arrives.

#### 1.4.2 Peak hour factor

The peak hour factor ( $PHF$ ) adjust the theoretical capacity by taking into account the maximum capacity attainable during peak hours. This factor is necessary because the average capacity calculated

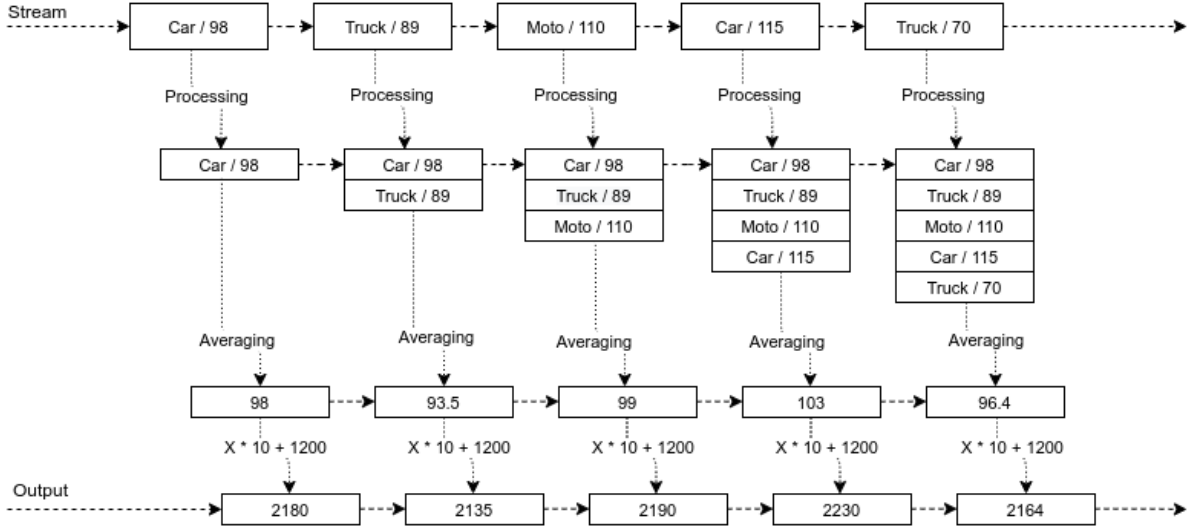


Figure 1.8: Theoretical capacity computing example

in theory can be misleading by an inconsistent distribution of traffic between peak hours and the rest of the time. The factor is determined by computing the variation in traffic density that may occurs over an hour.

The procedure to analyze while observing the traffic is to divide the hourly number of vehicles by four times the maximum number of vehicles over a period of 15 minutes. The result gives a coefficient that approaches 1 if the flow is stable and decreases the higher the variance in peak hours is.

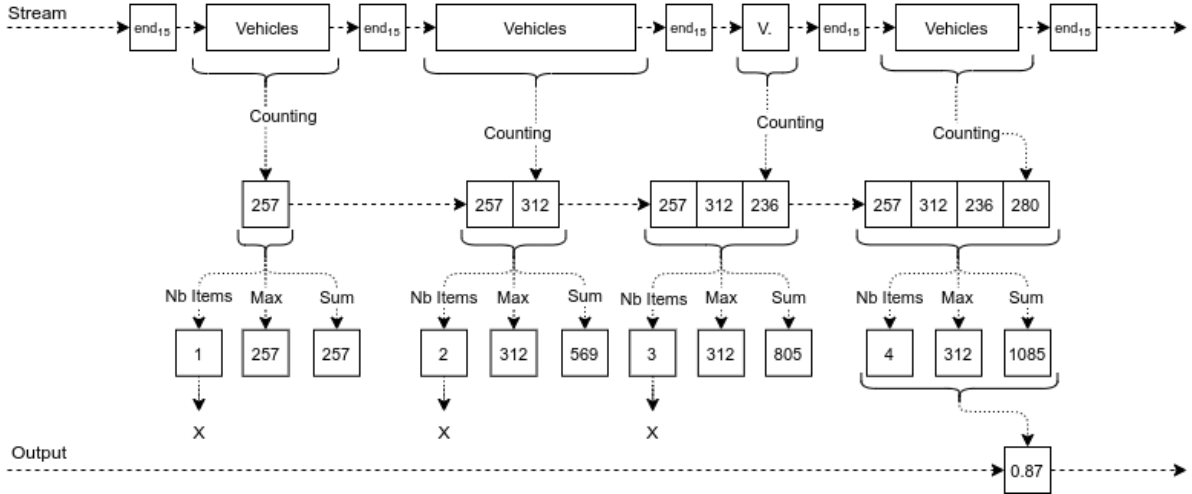


Figure 1.9: Peak hour factor computing example

In Figure 1.9, the representation of the input stream is drawn with  $end_{15}$  symbols delimiting the 15 minute periods observed, and Vehicles symbols representing series of vehicles arriving one by one.

The difficulty in obtaining a correct result is to perform synchronized calculations simultaneously. As can be seen from the example, each 15 minute period count should be kept until reaching the full hour, no result can be obtained before. At the same time, the total number of vehicles during the hour and the maximum number over 15 minutes should be calculated.

In the next steps of the example in Figure 1.9, which are not shown, the algorithm must not wait for a new entire hour to produce another result. Using the sliding window pattern, a new result will be produced every 15 minutes by removing the first period and adding a last. Note that, for a more precise



estimate, the 15-minute maximum count calculation could be repeated more than 4 times an hour. By choosing to perform the calculation every 5 minutes, it could then be repeated 12 times in the hour.

### 1.4.3 Type of driver factor

The coefficient linked to the type of driver ( $f_p$ ) is the only one that must not be calculated as a function on the stream. It's a constant between 1.00 and 0.85 which indicates the level of habit of the drivers on the analyzed road. The more the route is considered to be used by regulars, the higher the number.

### 1.4.4 Heavy vehicle factor

The heavy vehicle factor ( $f_{hv}$ ) makes it possible to take into account the influence of heavy goods vehicles on road traffic. Being given their load, they move forward more slowly, and they can cause slowdowns on the whole traffic if they are in large numbers.

This factor is determined in part by the ratio between the total number of vehicles and the number of trucks ( $P_c$ ). Moreover, if the track is on a slope, to this calculation is added another factor which corrects the previous one by implying the length and the inclination of the slope ( $E_q$ ).

Inclination (%)	Length of ramp (m)	E <sub>q</sub> Equivalence factor in ramps					
		Percentage of heavy vehicles ( % )					
		4	6	8	10	15	20
< 2	All	1.5	1.5	1.5	1.5	1.5	1.5
2	0-400	1.5	1.5	1.5	1.5	1.5	1.5
	400-800	1.5	1.5	1.5	1.5	1.5	1.5
	800-1200	1.5	1.5	1.5	1.5	1.5	1.5
	1200-1600	2	2	1.5	1.5	1.5	1.5
	1600-2400	3	3	2.5	2.5	2	2

Figure 1.10: Matrix for equivalence factor in ramps

The correction factor is predefined in a matrix, an extract of which is shown in Figure 1.10. This matrix define the relationship between the percentage of trucks, the characteristics of the slope and the correction factor.

The lower the percentage of the slope, its length or the percentage of trucks, the closer the heavy vehicle factor is to 1. The final formula for determining the factor is as follows:

$$f_{HV} = \frac{1}{1 + P_c(E_q - 1)}$$

Figure 1.11 represents a heavy vehicle factor process. The input flow is written with the symbols M representing motorcycles, C representing cars and T representing trucks. For this part of the example, the road was chosen arbitrarily as having a slope of 2% with a length between 1600 and 2400 meters.

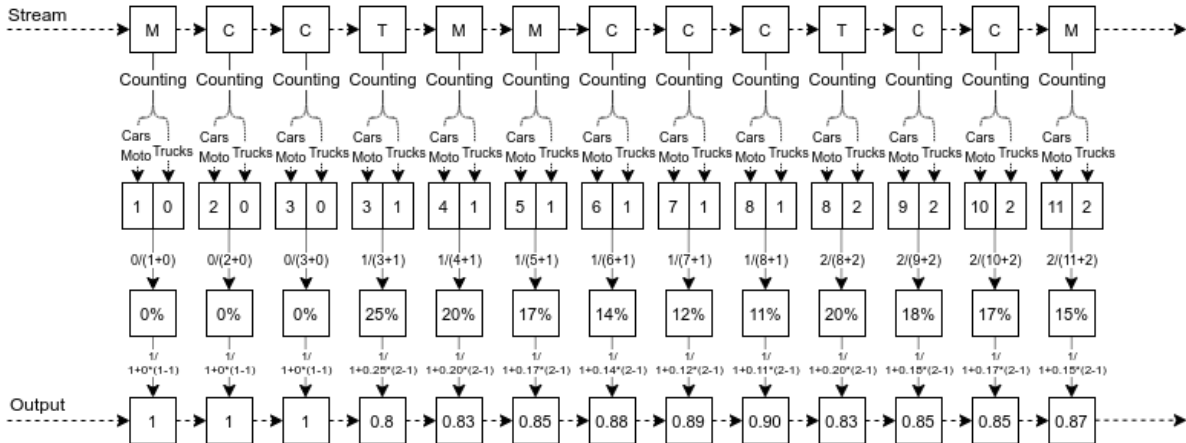


Figure 1.11: Heavy vehicle factor computing example

### 1.4.5 Practical capacity

The calculation of practical capacity ( $C_p$ ) is the final step in the process. However, it takes place at the same time as the others. Indeed to follow the evolution of the analysis in real time, the results of the previous four factors must also be combined in real time using a function to give the practical capacity. This marks once more the interest of streaming processing in this context.

The combination of the different factors is carried out by a simple multiplication as shown in the following formula:

$$C_p = TC_{pl} * PHF * f_{hv} * f_p$$

In the calculation of the different factors on the example data, the results tend to have strong variations. This is due in part to the use of a small sample of data only. In a real use case, it would then be useful to let the algorithm run until being capable of observing a stabilization of the results.

## 1.5 Testing strategy

To compare the ways of processing data flows, different streaming processing algorithm implementations will be analyzed and tested in the rest of the thesis.

All the implementations will remain in the theme of road traffic monitoring, and will deal more precisely with the example which has just been defined. Of course, other examples will also be presented but only from a theoretical point of view. In terms of software testability, keeping the same example for the different implementations provides an easy point of comparison.

The programming languages used for the developments are mainly Java and Scala. Therefore, all the tests will therefore be running on a JVM (Java Virtual Machine) under the same conditions.

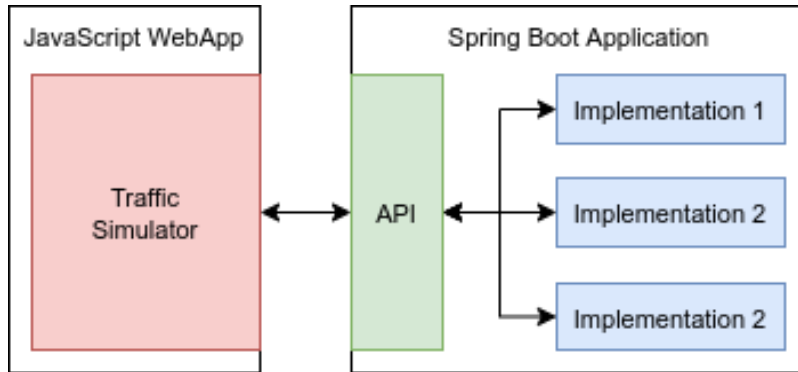


Figure 1.12: Testing architecture

Regarding the testing procedure, all tests were carried out by setting up a road traffic simulation via web services. The complete architecture is represented in Figure 1.12. The web services communicate with a Spring Boot framework based application which is responsible for executing one or another implementation.

While real-life use cases require waiting hours or days to observe conclusive results and verify system stability, this is not a mandatory requirement when using a simulator. Thus, the simulations were carried out in accelerated mode to be capable of observing changes over short periods of time. Concretely, a realistic simulation of road traffic observable over one hour has been condensed into one minute. The observation windows for the peak hour factor are therefore 15 seconds.

To put the system in a situation where the variation in performance of each algorithm has a great impact, the amount of memory allocated to the system (JVM) has been limited to 30 megabytes. In addition, it is necessary to be aware of all the components of the application, apart from the processing algorithms, which already consume around 15 megabytes. Information regarding the entire test system has been attached in Appendix D. The code is separated into two parts, one for the web application and another for the Spring Boot application.

## 1.6 Road traffic monitoring use case: A naive solution

After having defined the example and the testing strategy, the analysis of a first implementation is necessary to understand the difficulties which can be encountered for designing an algorithm for a streaming processing problem. This is why a simple implementation entirely in Java has been made. In this implementation, no external libraries are used. The whole algorithm is attached in Appendix C.1.2. Only the crucial parts will be explained here.

As a reminder, it must not be forgot that the goal is to create algorithms which are made for processing data flows in a flexible and expressive way while remaining efficient. By following this logic, the procedure used to analyze the implementation will be to discern its qualities and defaults in terms of modifiability, testability and performance. In the last parts of the thesis, other implementations will be presented with new languages and analyzed according to the same procedure.

### 1.6.1 Algorithm

First, some information on how the algorithm works in general. For each execution, a set of variables are instantiated. The algorithm reads each element sent to it, one by one, and updates the various variables. The arrays are used to aggregate the results by time periods. The arrays indexes are updated with each element defining a new hour or a new period of 15 minutes.

</>
Listing 1: Naive solution state
</>

```

1  Double length = 0D;
2  Double slopeGrade = 0D;
3  Double habitualUseFactor = 0D;
4
5  Double total = 0D;
6  Double totalTrucks = 0D;
7  Double totalBy1Hour = null;
8  Double totalBy15Min = null;
9  Double speedSum = 0D;
10
11 Double[] vehicleBy1Hour = {0D,0D,0D,0D,0D};
12 Integer indexVehicleBy1Hour = 0;
13 Double[] vehicleBy15Min = {0D,0D};
14 Integer indexVehicleBy15Min = 0;

```

The variables are initialized to the value 0D. This instruction assigns the value 0 to variables of type `Double`, which represents 64-bit numbers.

The variables `length`, `slopeGrade` and `habitualUseFactor` respectively contain the length of the tunnel, the percentage of its slope and the habit factor of the drivers who use it. They are initialized when the algorithm starts.

The `total` variable counts the number of vehicles observed. The variable `totalTrucks` counts the number of trucks observed. The variable `totalBy1Hour` accumulates the number of vehicles observed in the last hour. The variable `totalBy15Min` accumulates the number of vehicles observed in the last 15-minute period. The `speedSum` variable contains the sum of the speeds calculated from all the vehicles observed.

The `vehicleBy1Hour` array contains the number of vehicles observed over the last hour grouped into 15-minute periods. At each signal for a new 15-minute period, the `indexVehicleBy1Hour` variable changes the index at which the next vehicles must be counted. When the whole array is full, each value is moved to the next index to create the behavior of a sliding window. The `vehicleBy15Min` array and the `indexVehicleBy15Min` variable work on the same principle, but only keeping the vehicle count over the last 15-minute period.

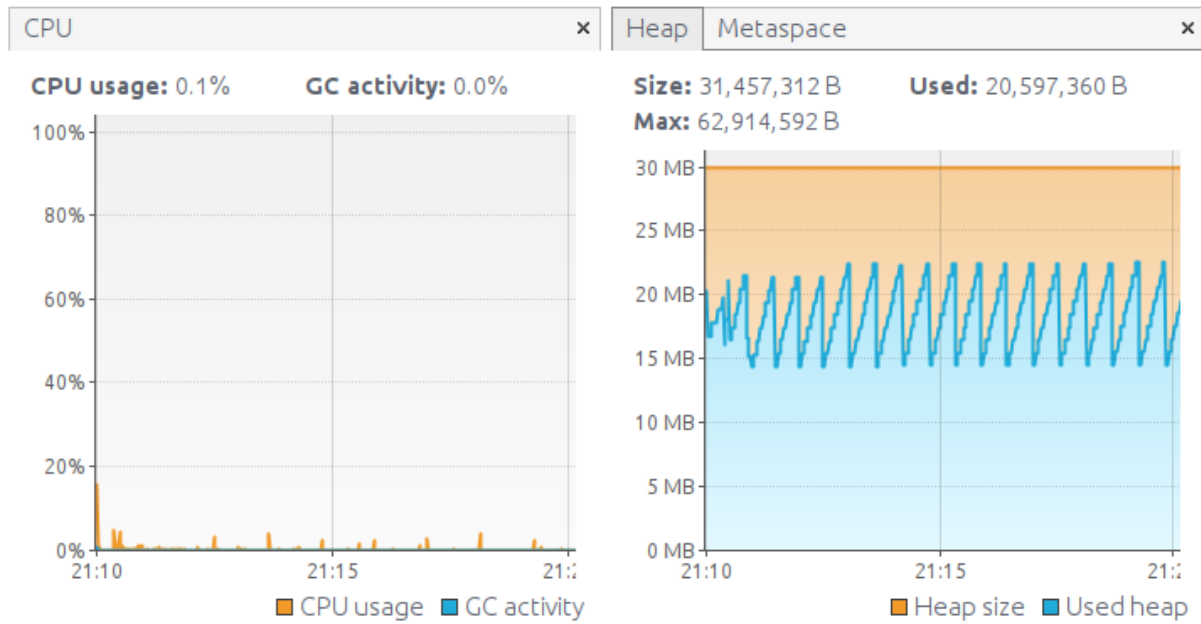


Figure 1.13: Performance of the naive solution

### 1.6.2 Performance

In Figure 1.13, the evolution of the performance of the application can be observed during a 10-minute simulation.

In terms of CPU usage, the operations to be carried out being inexpensive, the load remains very low. At the memory level, the JVM performs automatic memory management using garbage collection process. This automatic process explains why the memory used increases and decreases at regular intervals. As the iterations proceed, the amount of memory used remains stable and no increase remains between passes of the garbage collector.

In general, the implementation is efficient and facilitates good analysis of road traffic.

### 1.6.3 Modifiability

The design of the algorithm takes about a hundred lines of code. If the size of the program is small enough and the performance is still good, however the complexity of its conception is not easy.

It is necessary to maintain a growing number of variables the more the algorithm becomes more complex. Multiple nested conditions are required to optimize the moment to update each variable. Without accurate documentation, unit testing, and technical skills, the modifications are difficult to apply without risking regressions in performance or in correctness.

The following code is executed on receipt of each new `item` in the stream. The received object is verified as being a new `VEHICLE` by retrieving its type via the `getItemType` method. If this is the case, the variables declared previously are updated.

Otherwise, the `item` is announcing the start of a new 15-minute period. In this case, the values of `indexVehicleBy1Hour` and `indexVehicleBy15Min` are checked to find the new indexes for storing the count of the next observed vehicles. If the arrays are already filled, the inner values are shifted from one index to another to leave room for a new one and remove the one that is not useful anymore.

The last condition checks if any values are already present for the past hour and returns a result by combining them if so.

&lt;/&gt;

Listing 2: *Clause imbrication*

&lt;/&gt;

```

1  if(Objects.equals(item.getItemType(),VEHICLE)) {
2      ...
3  } else {
4      if(indexVehicleBy1Hour < 4) {
5          ...
6      } else {
7          ...
8      }
9      if(indexVehicleBy15Min < 1) {
10         ...
11     } else {
12         ...
13         if(totalBy15Min == null || totalBy15Min < vehicleBy15Min[0]) {
14             ...
15         }
16     }
17 }
18 ...
19 if(totalBy1Hour != null && totalBy15Min != null) {
20     ...
21 }

```

Among the states to be maintained, one of the most complicated depends on the sliding window pattern. The array and the logic used to aggregate the results must be updated each time to the number of period change.

&lt;/&gt;

Listing 3: *Array mutation for sliding window*

&lt;/&gt;

```

1  vehicleBy1Hour[0] = vehicleBy1Hour[1];
2  vehicleBy1Hour[1] = vehicleBy1Hour[2];
3  vehicleBy1Hour[2] = vehicleBy1Hour[3];
4  vehicleBy1Hour[3] = vehicleBy1Hour[4];
5  vehicleBy1Hour[4] = 0D;

```

#### 1.6.4 Testability

Modularity of the code is also an issue. To achieve such a concise program, the code is completely grouped into one single method. Since each variable must be updated according to other variables and a series of conditions, separating them in several methods without particular logic would have been complicated and would not have helped understanding.

This lack of modularity makes it impossible to test each part of the code in isolation. The tests on the code as a whole then increase the difficulty to find the exact source of the bugs during the development or maintenance phase.

#### 1.6.5 Conclusion

In this case, the performance is still quite good but the design starts to get very complicated, especially if the people who have to manipulate the language do not have sufficient technical skills.

In critical cases, the algorithms become more complex, chain more operators, and the number of properties computed at the same time on a single stream is expected to increase. Furthermore, the size of the stream is likely to grow indefinitely, it will be necessary to retain the minimum intermediate results necessary for each query.

Such problems highlight the need for an other language that would bring their resolution within the reach of the greatest number of people. Before analyzing the range of languages that already exist and to get a better idea of the scope of the problems involved, other areas of application will be reviewed.

## 1.7 Application areas

The aim of this section is to provide a general background for the data stream processing systems. Several areas are strongly impacted by the constraints related to IoT. By presenting some of these, the qualities necessary for an easy and efficient processing are highlighted.

### 1.7.1 Medical domain

When it comes to monitoring sensitive data streams, such as a patient's heart rate, calculations must be very precise and consume as few resources as possible, since these are very limited. Currently, the low-level languages used to make it necessary to constantly think about implementation details for that kind of tasks. In order to be able to easily design and adapt the algorithms for different treatments, depending on the nature of the streams, the need for a language with predictable performances (energy, runtime, memory) is more than required.

In streaming documentation [1], the explanations on the monitoring of heart rhythm disturbances make it possible to understand the challenges of such a language. These are as follows.

Arrhythmia Monitoring Algorithms (AMAs) are algorithms that detect an abnormal cardiac rhythm. However, they don't just detect that a heartbeat is missing. They go further by detecting patterns in the received signals.

Normally, the oxygenated blood supply throughout the body is ensured by the synchronized contraction of the lower and upper chambers of the heart, called respectively the ventricles and atria. These contractions are performed by regular electrical impulses (Normal Sinus Rhythm).

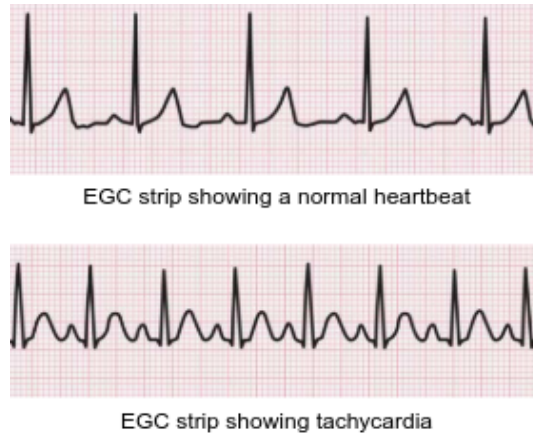


Figure 1.14: Heartbeat comparison from heart.org, May 2021

Disturbances in these contractions are called arrhythmias. They can generally fall into two categories:

- Ventricular Tachycardia: The ventricles depolarize at a very high rate. This does not leave enough time between the contraction and relaxation of the muscle to supply blood. It can get worse and lead to death within a minute.
- Supra-ventricular Tachycardia: The problem occurs above the ventricles. the disease consists of an excessively fast heartbeat which can be uncomfortable but is not fatal in the short term.

While the treatments must be radically different, these two types of arrhythmias remain difficult to distinguish on the basis of their symptoms. In addition, some arrhythmias, such as Atrial Fibrillation, which is a type of Supra-ventricular Tachycardia, should be monitored over continuous periods up to several months.

The long term of these periods requires a significant control of the energy cost of the algorithms. Implantable Cardioverter Defibrillators (ICDs) and Insertable Loop Recorders (ILRs) are the two devices implanted in the patient's body to execute them. The need for energy for the devices to keep working is low, only the shocks are very consuming. The rest of the energy is shared between pacing and monitoring, which therefore determines the lifespan of the device.

In addition, the operations to implant these devices are generally performed on senior patients, and the risks of disease or infection are not to be neglected. It's a priority not to increase the number of surgical operations.

The signals received to execute this monitoring are called electrocardiograms, or EGMs. They represent the electrical activity of the heart. The need to process these signals in real time and in a very short time frame imposes the use of a streaming model of calculation.

The challenge is to create algorithms from these signals. The priority is the ability to differentiate the arrhythmias efficiently, while constantly keeping updated health indicators calculated over the entire stream.

These constraints lead to the use of a specific language for streaming processing. This is a need for which a language with a relative simple syntax offer a particularly suitable solution. In particular by making the exploration of design choices much more accessible in the early stages of the algorithm design and bringing more productivity during the whole process.

In some other articles [4], beyond the performance, it's the need for more precision in the algorithms that is really put forward. Still for ICDs, it is necessary to know how to detect each heartbeat, without having a false positive or false negative. Currently 10% of ICDs errors are due to these inaccuracies.

The detection is made on the basis of different discriminators and, at runtime, their combination reveals the possible disease. Several discriminators require the use of different logics. For example, one might just detect each beat, while an other discriminator averages the beats that follow each other at high frequency.

When the logics used are different, the metrics in terms of energy cost, performance and precision are not always easily comparable and the formulas quickly become error-prone.

Being able to represent in a single formalism the different steps to detect high frequency peaks in EGMs helps to ensure both the overall accuracy of the algorithm and its energy consumption.

### 1.7.2 Banking industry

Economic fields of expertise are also in the need of efficient streaming processing solutions.

Concerning marketing, many companies benefit from immediate results through analyzing the data received in real time. One essential goal is to sell the products at the right price depending on customers, sales channel, or selling period. To perform precise actions on the price elasticity, streaming data are a good way to react immediately in a personalized fashion for each customer. These are techniques widely used in social media marketing. Companies receive so much data from social media that it raises the need to integrate it in their business intelligence process for better decision making. The problem is that, usually, existing data warehousing systems cannot support the volume of data, forcing the process to be executed in real time.

At the same time, the irregularity monitoring in transactions, using pattern detection applied to the transaction streams, avoids fraud attempts and allows the costs of fraudulent transactions to be canceled. Data are rarely used on their own. They can be combined with other historical data to make them more useful. This requires ease of integrating the algorithms designed into existing programs, which is often to the disadvantage of closed-systems.

In banking sector, they deal with problems such as the monitoring of banking transactions on ATMs. In this context, the requirement to monitor indicators on the expenses or withdrawals of a bank's customers is a huge challenge. For example, a bank may want to set up streaming processing algorithms to calculate the average or maximum expenses over a given time [10]. Such operations may seem simple but when they must be carried out in real time on all the customers, things can get complicated very quickly.

If data stream processing systems that handle transactions one by one exist, some studies [10] claim that a language to specify regular iterations over chunks of inputs as not been developed yet.

However, the need to be able to manipulate the stream while maintaining intermediate states in an efficient manner on different levels is a crucial issue. In the context of banking transactions, the different levels may represent time intervals. For example, one may want to calculate the cumulative sums deposited per day for a client. Then, when the end of the month arrives, only keep the maximum over

the entire month, and at the end of each year to have an average on the maximum peaks of money deposits.

### 1.7.3 Network monitoring

One of the main challenges of network monitoring is to detect anomalies to prevent attacks and to enforce a strict security policy.

As explained in streaming processing documents [50], security attacks, such as denial of service, requires to trace traffic variations through TCP and HTTP connections. The detection of patterns in a stream of packet transmissions helps to identify possible anomalies. For example, SSL renegotiation attack could be detected in this way. This attack depletes a server's resources by performing non-stop handshakes. For each handshake, the server will have to decrypt the message encrypted by the client with the certificate public key, using its private key. The encryption being way easier than the decryption, an SSL/TLS handshake requires at least 10 times more processing power on the server than on the client [43]. With this method, a single host can exhausts the server resources. One solution can be to setup a pattern detection for the presence of a large number of connections having each one an unusually low number of bytes transferred.

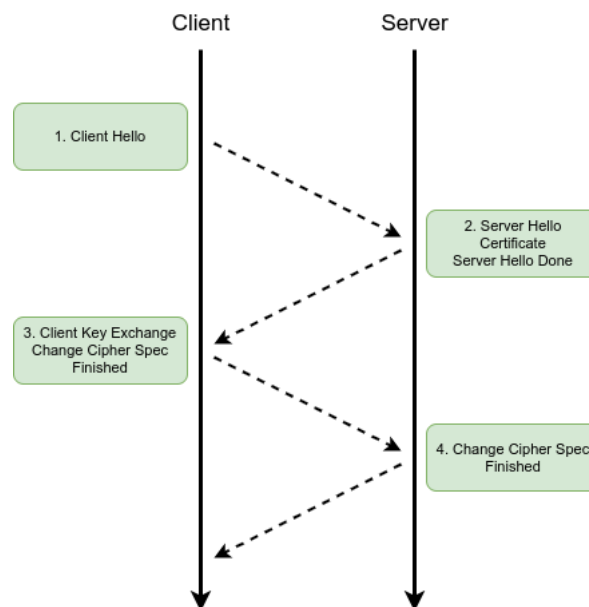


Figure 1.15: SSL Handshake schema, encryption is done at step 3, and decryption at step 4

Manually writing low-level code to detect those attacks with an imperative language can be really cumbersome because it needs to build complex state machines and to handle interactions across layers.

Apart from security, in network management, monitoring is also used to control application policies.

When it comes to bandwidth, the heavy-hitters are definitely video streaming and VoIP communications. During VoIP sessions, users are generally limited in the quota of data they can consume or in the duration of the connection. This involves monitoring sessions, by user, and updating their data indicators, even when a user's IP address is updated.

Whether this is application-level performance metrics or detection of security attacks, the monitoring should be capable of computing quantitative metrics and actual low-level languages are not suitable for aggregating information from a variety of network and application layers. An adapted language is a solid foundation for network configuration which needs dynamic updates in order to meet performance and security goals.

### 1.7.4 Diverse use cases

Many other use cases could be introduced. Even if the number of application areas are too high to present them all, four other isolated use cases are presented here to illustrate the required flexibility of



a potential solution. They mainly come from reliable streaming problem analysis [1].

Concerning public transport systems, one issue is to calculate the time needed to get from point A to point B. The event stream can be produced by every ticket scan at bus or metro stations. By analyzing the average time per trip, or per customer, the managers can quantify the need to add more transport on a particular sector or the necessity to analyze the time lost between bus or metro exchanges. However, the techniques used in the already existing solutions are essentially relying on batch processing, which are not designed to detect unexpected events, such as accidents or an increase in demand due to external factors. Previously, technical limitations required delaying processing operations. Strong assumptions were made in the machine learning techniques for example, such as the need stationary data distributions and finite training sets, but this is no longer the case now where research shows that these algorithms can be directly supplied in real time [40]. Specific streaming processing algorithms are therefore needed.

Concerning telecommunications operators, one obligation can be to keep the consumption status of a telephone subscription per month, in order to be able to decrease or increase the data cost for the following month.

Concerning global climate observing systems, the monitoring of the temperatures over the seasons and years is required to observe any increase or decrease from one year to the next.

Actually, some systems even aim to cross analyzed data from several different domains. This is the case with current concerns about climate change and its impact on health conditions. In recent books [16], research focuses on the continuous monitoring of the correlation between climate change and Dengue, a viral disease transmitted by mosquitoes, in the tropics and subtropics. The main task of their proposed system is to monitor the spread of disease and predict. In this way, they can observe and minimize damage by epidemic and pandemic situations, as well as provide knowledge on what factors contribute to such circumstances.

This type of system is not an isolated case. Others have been studied in the past for malaria and the Ebola virus in India. They represent an important big data challenge especially since more and more patients agree to share electronic health records.

### 1.7.5 Qualities and suitability

Within the various fields of application certain qualities are easily discernible. These qualities are the ones that a new language must work on to improve the quality of the solutions produced. A price point is that the language should not be intended for a specific field, and must therefore adapt to different needs without making too much compromises.

Among the features necessary to carry out the processing of data stream, the control of resource consumption is essential, as well as their quantification, in particular in the medical field. Good performance is always a necessity. There is an interest in the banking system for maintaining indices over long periods of time, therefore the performance must stay stable during potentially very long intervals. In addition to performance requirements, network monitoring needs a language which brings great accessibility and ease of design. Actually, low level languages make maintenance and improvement of algorithms very tough.

Overall, all these requirements demonstrate the call for a great flexibility regardless of the use case. It remains to be seen whether there exists, or not already, a language that might be suitable.

## 1.8 Conclusion

In conclusion, processing data streams in real time is not an easy task. In this contextualisation, the overview of the data streams principles procured the key elements to establish their characteristics. Different models exist and they cannot be handled in the same way. In this thesis, the scope of the language sought is restricted to processing time series streaming model.

Through some examples, particularly within the road traffic monitoring domain, visualization of the concepts, like filtering or aggregating data, showcased the difficulty of handling streaming data without suitable languages. This difficulty, of easily creating algorithms, has been put into perspective in light of the very high expectations in performance for such systems.

The algorithms need performance, with a language providing enough expressivity, while having a layer of abstraction to dissociate the languages features from the field of activity concerned. In this way, no need to reengineer a specific language for each use case. The flexibility of the language must be all the greater as it must focus on the most general streaming model, the time series model.

It remains to be seen whether such a language already exists as is or whether further research is necessary.

# Chapter 2

## Existing languages

### 2.1 Introduction

Many languages have already been designed for handling data streams. They are classified into different categories depending on the approach by which they were designed. In this second chapter, these different categories are studied.

First, a brief introduction presents the main language that is analyzed in this thesis, the Quantitative Regular Expressions, also known as QREs. Then, a series of other streaming languages are presented, one by one, with examples allowing to understand their syntax. The goal is to have an overview of the existing languages and to explain why it was interesting to have a new one, the QREs.

The purpose is not to go through the set of all the languages related to data processing in real time, nor to examine them in depth for each little detail. If they all have an interest, their goals can be noticeably different from that sought by the QREs, the scope would be way too big and not all languages can always be compared on the same characteristics.

Here, the analysis focuses on a subset of these languages. This subset only contains all those that are both linked to the context, which was defined in the previous chapter, and which come close enough to the objectives of QREs, but opting for a different angle of attack. For each of the explained languages, the differences in approach and objective are highlighted. This will make it possible to discern the biases, the differences and the commonalities with the QREs.

### 2.2 Quantitative regular expressions

Quantitative regular expressions (QREs) are a high-level abstraction language relying on theoretical bases to guarantee the performance and expressiveness of their implementations. The prefix quantitative comes from their initial goal, which was oriented for numerical data processing, although they are designed to adapt to other data types. The patterns used to recognize the domain of the stream, to which they apply, is derived from regular expressions. That justifies their suffix. The purpose of quantitative regular expressions takes place over several levels. The main goal is to facilitate the exploration of design possibilities without having to worry about low-level constraints thanks to a domain-specific language (DSL) which gives expressiveness, and a more natural reading of the language, while maintaining great performance.

Their first contribution is the definition of generic operations which can be performed on a part of the data stream determined by a regular expression. Next, the language uses a modular system allowing the integration of domain-specific and user-defined functions, which, integrated with a strict typing system, avoid losing efficiency in the processing. The operations are combined with each other using a hierarchical structure while staying easily readable thanks to their formal syntax and semantics. Then, the cost of a QRE can be computed in relation to the cost of user-defined computations that compose it. Finally, the precise complexity limits on the processing time by input element and on the total memory footprint can be statically computed as the expressions are restricted to apply on a regular language.

**Example 2.2.1.** In Figure 2.1, the example shows how quantitative expressions can split a stream into several pieces and apply different functions to each of the parts. In a heterogeneous flow, made up of car and hour ending tokens, the maximum number of observable cars per hour is computed by a composition of function aligned with the regular patterns. First, the input stream is divided into regular iterations on cars, the total of observed vehicles is emitted when receiving a token indicating the end of the current hour. At the same time, the iterations on the concatenation of cars followed by hour ending token work for keeping only the maximum count among the previous hours.

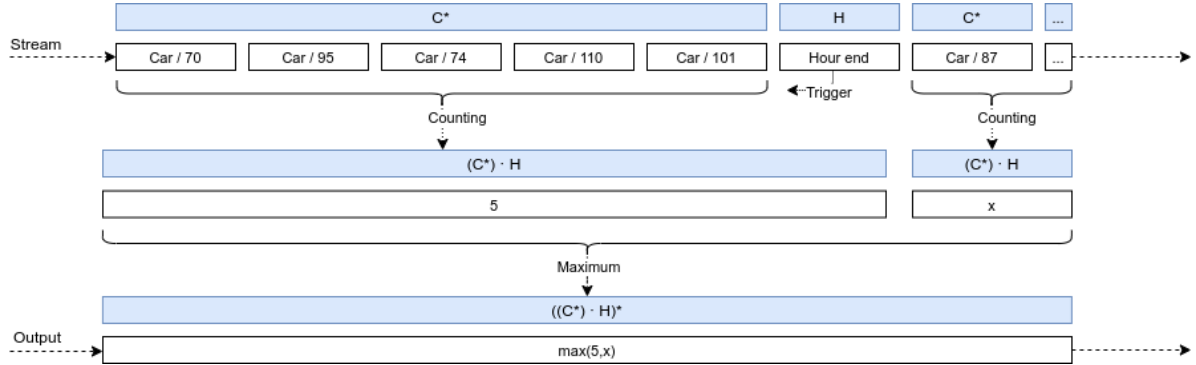


Figure 2.1: QREs, as a schema, to apply functions on regular subset of a stream

As a result, the developer does not need to worry about the implementation details for maintaining the state of the stream and updating it. He can focus on implementing the operations related to the business case, specifying at which part of the stream they are related, and the QREs system guarantee him the performance and the results.

The following code is the design of a function implementing the algorithm shown in Figure 2.1. The query is written in Java, using the implementation of QREs called StreamQRE. The code has the advantage of being fairly concise and easily configurable, which makes it pleasant to read and write while maintaining guaranteed performance, as will be demonstrated in the next chapters. Information to download the StreamQRE library and be able to integrate it into a Java project is provided in Appendix B.1.

</> Listing 4: QREs, as Java function, to apply functions on regular subset of a stream </>

```

1 private static QReIter<StreamingDto, Long, Long, Long> maximumOfVehiclesByHour() {
2     var isHourToken = new QReAtomic<StreamingDto, Long>(x ->
3         Objects.equals(x.getItemType(), TIMER), x -> 0L);
4     var isVehicleToken = new QReAtomic<StreamingDto, Long>(x ->
5         Objects.equals(x.getItemType(), VEHICLE), x -> 1L);
6     var isVehicleSum = new QReIter<>(isVehicleToken, 0L, Long::sum, x -> x);
7     var isVehicleSumByHour = new QReSplit<>(isVehicleSum, isHourToken, (x,y) -> x);
8     var isMaxVehicleSumInLastHours = new QReIter<>(isVehicleSumByHour, 0L,
9         Math::max, x -> x);
10    var isMaxVehicleSumByHour = new QReSplit<>(isMaxVehicleSumInLastHours,
11        isVehicleSum, Math::max);
12    return isMaximumVehicleSumByHour;
13 }

```

The query is divided into several subqueries, each one recognizing a subset of the stream.

The subquery `isHourToken` recognizes a single token whose type, retrieved by `getItemType` method, corresponds to the signal of the end of a 1-hour period and the start of a new one. It receives a `StreamingDto` input object and produces a value of type `Long`, a 64-bit integer. Here the value produced will always be 0 because of the function `x -> 0L`. This function indicates the value to return for each new item read and recognized by the subquery. The `StreamingDto` object represent the reification of a streaming item into a Java object.

The subquery `isVehicleToken` query is similar to `isHourToken` but recognizes the items corresponding

to a new observed vehicle signal. The return value is always 1.

The subquery `isVehicleSum` calculates a result based on the observation of a series of consecutive vehicle signals. It uses the result of `isVehicleToken` subquery and adds all the 1 values via the function `Long::sum`, starting with an initial value of 0. By the function `x -> x`, the resulting sum is returned as is.

The subquery `isVehicleSumByHour` recognizes an input stream splitted in two parts. The first part is delegated to the `isVehicleSum` query, the result of which is retrievable in the variable `x`. The second part is delegated to the `isHourToken` query, the result of which is retrievable in the variable `y`. The function `(x,y) -> x` returns the number of vehicles `x` observed before the end of an hour `y`.

The subquery `isMaxVehicleSumInLastHours` iterates over the `isVehicleSumByHour` query to recognize successive hour periods. The function `Math::max` only keeps the maximum number of vehicles observed over an hour, starting with the value 0. By the function `x -> x`, the maximum value is returned as is.

The final query `isMaxVehicleSumByHour` domain spans over the whole stream. It splits the input stream in two parts. The first part computes the maximum number of vehicles by hour during the last hours with the `isMaxVehicleSumInLastHours` subquery. The second part computes the consecutive sum of vehicles for the current hour while reusing the subquery `isVehicleSum`. The maximum number of vehicles observed in one hour is recalculated at the arrival of each new vehicle by combining these two parts with the function `Math::max`. This query closes the behaviour of the entire QRE.

StreamQRE is not the only implementation of the QRE specifications existing. Some others have already been developed and tested. A few years have passed between the beginnings of the language and its first implementations. The first stages of the language have been written at the beginning of the last decade. Since then, they have evolved over the years, while gradually introducing new abstractions, until they have been implemented in modern languages. This evolution can be divided into several successive stages.

In the early 2010s, the basics of the language are edified. Among those, the functions on which the quantitative regular expressions rely on, called *regular functions*, are created. They manipulate and transform the streaming elements and they are built on the basis of regular and predefined models. Their behavior can be simulated by a specific type of transducer, called a *cost register automata*. In the same way that a finite transducer can translate the words from a regular language, a cost register automata can produce the results of a regular function, progressively, as the elements of the stream are received. This type of automata extends deterministic finite state automata. The extension adds registers to keep some values in memory while reading the stream, and thus to support the calculation of quantitative properties as the result of processing data streams. Because these methods guarantee very good performances, the creation of the language could have stopped at the design of one of these new automata for each stream processing task. However, the design of automata is quite complicated for a person with little knowledge in the field. The desire was for the language to be easy to learn and handle. In the next years, abstractions in modern programming languages have been created so that its use is within the reach of the greatest number.

In a first implementation restricted to string-to-string transformations, called DReX, concrete stream manipulation techniques are implemented in a custom language for the first time. Through the use of multiple operators, the extraction of regular and finite subsets from a stream is made possible. Each operator, called *regular combinator*, using a regular expression, applies a regular function to its subset and produces the output of the stream processing. Thanks to recursion, using a bottom up way, the resulting stream of a combinator can feed, as input stream, another operator. To refer to the small example above, each line of code present defines a regular combinator, and the whole `maximumOfVehiclesByHour` function can be called a regular function.

Then, the algorithms were implemented using general-purpose programming languages. Between 2016 and 2019, StreamQRE, developed in Java, and NetQRE, developed in C++, were the main implementations created. The development of these systems led to their analysis in comparison with similar, and already existing systems, such as the ones presented in the next sections. These implementations are much easier to use than building an automaton from scratch. As shown in the example above, in a few lines of code a query is built and ready to be executed. Thanks to the internal functioning of StreamQRE and NetQRE libraries, which are limited to operations expressible by a cost register automata, performance remains guaranteed.

The multiple stages from the early conception until now were mainly done by Rajeev Alur, Konstantinos Mamouras and Houssam Abbas. They are the principal authors, relying on the reference documents, behind the foundation of quantitative regular expressions. Their articles were used to write this thesis. They cover the language evolution from 2010 until 2020.

Now is the time to browse other potential languages for dealing with data streams, and seeing in which way they differ.

## 2.3 Formal languages

To begin with, in the field of formal language theory, several approaches and languages have already been developed. This is the oldest way that has been used to conceive what can be compared to the actual streaming algorithms.

In the 90s, the processing of data streams is already seen as series of events distributed, continuously, over an indefinite period of time [9]. Among the many systems arising at this period, the need of a formalism for cyber-physical systems (CPS) drives the development of continuous monitoring of diverse mechanisms. It ranges from playing the record of a disk drive to flight controls inside airplane trips. The systems which processes events are called *reactive*. They automatically interact with their environment according to the arrival of certain types of events.

### 2.3.1 Hybrid automata

Formal models, well suited to build such systems, have already been studied at this time and are still in use at the present time. They simultaneously combine discrete and continuous dynamics. This justifies their name of hybrid systems. They enrich the model of finite state automata, capable of managing discrete events, with a finite number of dynamic variables whose values evolve dynamically over time. The evolution over time of these dynamic variables can be achieved by means of differential equations.

For example, in the context of medical devices [20], such streams can be electrical signals, or variations of the level of glucose in the blood. These are continuous, non-linear, data streams which describe the evolution of a property over time. A hybrid system makes it possible not only to monitor them continuously, but also in the same time react to stochastic events. Those can be random effects of the medical disposal including over-sensing, under-sensing, failure-to-capture, or simply noise on the sensor readings. This also helps to react to random behaviors in heart activity. Those stochastic behaviors arise, and can be simulated, through the use of probability distribution functions [32]. Depending on these events, the patient is detected as sick or not and the medical device can change the operating mode.

Concretely, a hybrid automaton  $H$  has the following components:

- **Variables.** A finite set  $X = \{x_1, x_2, \dots, x_n\}$  of real-valued variables. Their number is called the dimension of  $H$ . The set of the first derivatives  $\dot{X} = \{\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n\}$  used to represent continuous change of variables. The set of primed variables  $X' = \{x'_1, x'_2, \dots, x'_n\}$  representing values at the conclusion of discrete change.
- **Graph.** A control graph which is a finite directed multigraph composed of vertices, called control modes,  $V$ , and edges, called control switches,  $E$ .
- **Initial predicate.** Each vertex can be labeled by an initial predicate whose free variables are from  $X$ . The labelling function is written  $init(v)$ , with  $v \in V$ , and defines the initialisation of the continuous variables in the control modes. The set of initial states,  $Init$ , defines the initial state of  $H$ .
- **Invariant predicate.** Each vertex can be labeled by an invariant predicate whose free variables are from  $X$ . The labelling function is written  $inv(v)$ , with  $v \in V$ . The automaton can reside in the mode  $v$  while the predicate is *true*.
- **Flow predicate.** Each vertex can be labeled by a flow predicate whose free variables are from  $\dot{X} \cup X$ . The labelling function is written  $flow(v)$ , with  $v \in V$ . It describes by differential equation the continuous evolution of the variables in  $X$ , while the automaton is in mode  $v$ .

- **Jump predicate.** Each edge is labeled by a function whose variables are from  $X \cup X'$ . The function describes the condition to jump from a discrete mode to another using a control switch. During the jump, new values can be assigned to continuous variables.
- **Event.** Each edge can be labeled by events. The events shared by the automata allow them to interact with each other using parallel composition. In parallel composition, the transitions labeled with common events synchronize the transitions of the interacting automata.

Finally, the definition of the hybrid automata can be written as a tuple  $H = (V, E, init, inv, flow, jump)$  [33].

### 2.3.2 Timed automata

Originally, hybrid systems are a generalization of timed systems which were introduced by R. Alur in the 90s [44], the same author who introduced QREs in the previous years. In the theory of timed systems, the variables whose values change over time, express time variations themselves. Based on the model of a finite state automata, a timed automata is supplied with *clocks* that measure the delays between the actions of the systems. All clocks evolve continuously and synchronized all together with time.

These delays can serve as a *guard* to prevent, or automatically trigger, transitions between states. In fact, for each transition, clocks can be included in a condition authorizing their execution, and for each state, a condition can also restrict time in-state and therefore forces the execution of a transition. Clocks can also be reset during these transitions. They make it possible to program real-time systems and to establish several time properties which must be respected by these systems [34].

Those properties and conditions are set up by defining quantitative constraints on the delays required between events. These constraints can be specified using extensions of temporal logics, to include the use of clocks. Finally, the automata can use model-checking techniques [5] to verify the satisfiability of the properties by the system.

### 2.3.3 Use case

**Example 2.3.1.** In Figure 2.2, two automata are introduced to demonstrate the hybrid and timed system behaviours. In this example, the goal is to optimize the transitions between the different states of a traffic light to make the traffic more fluid.

On the left, a hybrid automaton follows the evolution of the number of cars waiting at a red light. On the right, a timed automaton triggers the change of state between the different modes of a traffic light.

First, let's analyze the specification of the timed automata. The clock,  $c$ , measure time elapsed in each discrete state of the system. The clock is reset after each transition between two vertices. The invariant for each mode restricts the possible time to stay in the discrete state. In the "Orange" mode, the constraint  $c \leq 1$  limit the maximum time to 1 time unit, while it's 6 time units in the others.

For example, in red mode, when the clock  $c$  reaches 6, a transition to the green state will be made. The guard of the transition,  $c \geq 6$ , is satisfied by the value of  $c$ .

Discrete events can also be used to trigger a state change, such as switching on or off the traffic light. The global state of the timed automata is the set of clock values at a given time, associated to the current vertex.

Next, let's analyze the behaviour of the hybrid automata. The automaton defines the states of a sensor which tracks the evolution of the number of cars waiting in front of a traffic light. The system is initialized in the "Low" state. Whenever the number of vehicles reaches 4, the automata jumps into the high state.

The variable  $x$  represent the number of vehicles waiting. In control mode "Low", the number of vehicles increase according to the flow condition  $\dot{x} = Kx$ ,  $K \in \mathbb{R}$ ,  $K > 0$ , where  $K$  is a constant. This differential equation was chosen arbitrarily, as is the one in the control mode "High". In a real situation the number of cars would not increase or decrease in a linear way. At the initial state, zero vehicles are waiting. According to the jump condition,  $x \geq 4$ , the system may change its control mode as soon as the number of vehicles reaches 4, and according to the invariant condition,  $x \leq 4$ , the control mode must switch when the number of vehicles reaches 5.

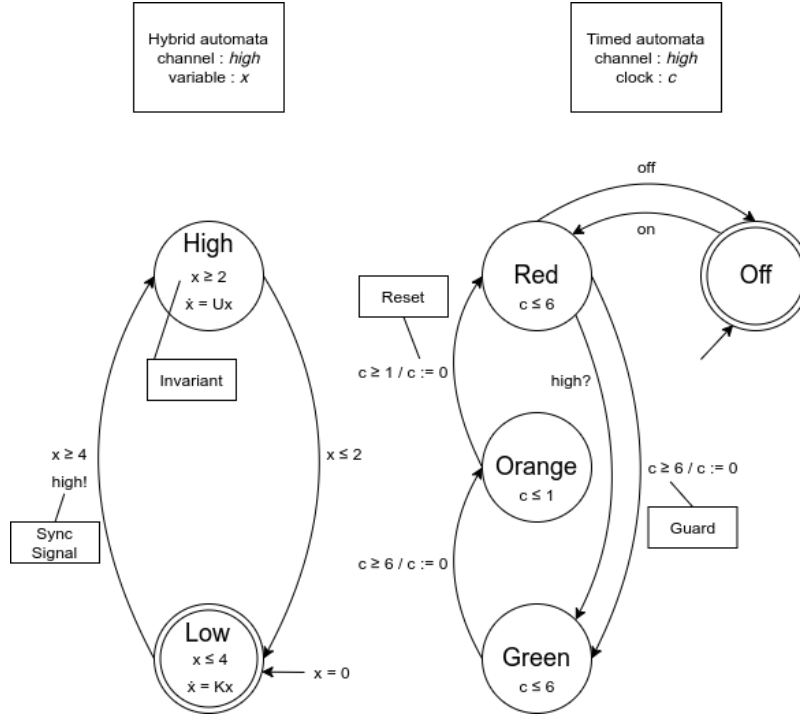


Figure 2.2: Example of timed automata

Finally, each time the hybrid automata detects too many vehicles, it emits the event *high* during its control switch. This event is received by the time automaton which automatically turns to green mode if possible. The emitted event is followed by an exclamation mark and the receiving transition event is associated with a question mark.

As in this example, continuous change of variables in hybrid automata is not always easy to describe with differential equations. The more complex the use cases become, the more difficult it becomes to have a good understanding of the general operational mode of the automaton.

Concerning timed automata, what made their notoriety is their relatively simple formalism allowing the expressiveness necessary for the modeling of timed systems. However today, in many cases, medical, traffic monitoring, or other ones, it's not sufficient to only detect a missing event after a certain delay, or to follow the temporal evolution of a variable.

For both, it is complicated to use them to keep the state of previous events in a real-time context. This prevents easily recognizing patterns on a whole stream. They therefore do not constitute a solid basis for designing a language allowing the user to avoid maintaining the algorithm state himself additionally to some expressiveness capacities.

## 2.4 Temporal logic

In parallel with the design of hybrid systems, studies have been carried out with the motivation to improve their validation methodology. That approach consists in relying on variants of temporal logic to validate the behavior of hybrid systems, or other continuous systems.

Basically, the temporal logic is widely use to do formal verification. The main technique is model-checking, in which, the requirements of hardware or software systems are verified while receiving a succession of events. For example, the verification can be that the access to a part of the system is prohibited following a specific event, or that an event must always trigger some kind of specific system behaviour. There are a multitude of variations of temporal logics, each one adapted to a very specific category of validation. Timed Propositional Temporal Logic (TPTL) includes variables to be able to measure the time between two events. Computation tree logic (CTL) is a branching-time logics using a tree-like structure to define the different paths the system could take in an indefinite future. In this



present case, the logic that could be the most suitable to streaming processing is the Signal Temporal Logic.

Signal Temporal Logic (STL) is a temporal logic particularly suitable to real-time validations. The logic is designed to do specification for validations on properties over real-valued signal, in real time. The aim is to immediately alert the user if a property is violated. The monitoring is therefore much more reliable than manual inspections on the system.

Their syntax is capable to express temporal properties of continuous real-valued signals, such as properties of trajectories<sup>1</sup> for hybrid automata. They are based on a bounded subset of the real-time logic Metric Interval Temporal Logic (MITL), which can express temporally evolving properties with time-critical constraints, or time-triggered constraints, on finite intervals.

The syntax of signal temporal logic (STL) is defined by [17]:

$$\varphi ::= \text{true} \mid p \mid f(\vec{x}) > 0 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathcal{U}_{[a,b]} \varphi$$

where

- $p$  belongs to a set of propositions  $P = \{p_1, \dots, p_n\}$ ,
- $\vec{x} : D \rightarrow \mathbb{R}^l$  is a  $l$ -dimensional signal,
- $f : \mathbb{R}^l \rightarrow \mathbb{R}$  is a real-valued function,
- and  $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$  must be read as, " $\varphi_1$  must remain *true* until  $\varphi_2$  becomes *true*". Note that the until operator always requires a time instant in this logic, which is not always the case in other logics.

The standard boolean and temporal operators *eventually* and *always* are defined as:

$$\Diamond_{[a,b]} \varphi = \top \mathcal{U}_{[a,b]} \varphi \text{ and } \Box_{[a,b]} \varphi = \neg \Diamond_{[a,b]} \neg \varphi$$

**Example 2.4.1.** Put in application, the syntax validate the satisfaction or the violation of properties by an automata. For example, several temporal properties could be specified on the hybrid automata drawn in Figure 2.2. These properties are not realistic, but they demonstrate the capacities of the validation:

- $\Box_{[0,300]}(x < 7)$ : during the first 300 time units, the number of vehicles waiting behind the traffic light will stay inferior to 7.
- $\Box_{[0,150]}(x > 5 \rightarrow \Box_{[5,10]} x < 3)$ : during the first 150 time units, whenever the number of vehicles increase to become greater than 5, the number of vehicles waiting will go down under 3 within 10 time units after 5 time units.
- $\Diamond_{[0,200]}(x = 0 \rightarrow \Diamond_{[0,6]} x > 3)$ : at some point during the first 200 time units, the number of vehicles waiting will go down to 0, and then will eventually increase to more than 3 during the next 6 time units after it happens.

Simulation and testing are the preferred way to execute the validations in the scope of automatic verification tools, as well as in other use cases. However, in previous years, adaptations of temporal logics brought the possibility to monitor software during runtime, while they are actually used [36].

This work was a step forward on more systematic and rigorous validations on streaming data for continuous and hybrid systems. Despite that, the problem lies in expressiveness of the temporal logic [3]. The class of problems that can be solved is still quite limited, even while taking into account the existence of the other temporal logics that follow similar patterns.

In addition, the system must keep in memory the state of stream efficiently, which is not that easy for that kind on tools. If validation of hybrid automata and timed automata detect irregularities in signals, they do not maintain a state on the previous elements of the stream. This feature can be achieved by involving pattern recognition, or a similar technique, to keep track of some, or all, events of the stream. In QREs, a key feature is that the pattern recognition is an integral part of the language.

---

<sup>1</sup>Evolution of the value of real-time variables

## 2.5 Pattern recognition

Quantitative regular expressions language isn't the first to use pattern recognition to process a data stream. A set of concepts and techniques to detect patterns in event streams, including both the occurrence and non-occurrence of events, already exists. These methods are called complex event processing systems and their queries can even specify intricate temporal constraints.

One of them is CEDR [18], the abbreviation of Complex Event Detection and Response. That declarative query language is based on a temporal stream model which provide support for consistency guarantees.

In this context, the data stream is considered as a time varying relation. Each tuple in the relation is assigned to its occurrence time, attributed by the event provider, but also with a validity time. To determine the status of an event, a time interval is associated with it. This interval defines its validity period. Queries make it possible to retrieve all the valid events at a given time. The properties of events, such as the validity interval, can be modified by issuing a new event with an identifier referring to a past event with the same identifier.

Operator	Description
<b>ATLEAST</b> ( $n, e_0, \dots, e_k, w$ )	Aggregate window to verify that the events $e_0$ to $e_k$ are received at least $n$ times during a $w$ interval.
<b>ATMOST</b> ( $n, e_0, \dots, e_k, w$ )	Aggregate window to verify that the events $e_0$ to $e_k$ are received at most $n$ times during a $w$ interval.
<b>ALL</b> ( $e_0, \dots, e_k, w$ )	Verify that the events $e_0$ to $e_k$ are all received during a $w$ interval.
<b>ANY</b> ( $e_0, \dots, e_k, w$ )	Verify that any of the events $e_0$ to $e_k$ is during a $w$ interval.
<b>SEQUENCE</b> ( $e_0, \dots, e_k, w$ )	Verify that the events $e_0$ to $e_k$ occur in the specified order during a $w$ interval.

Table 2.1: CEDR event sequencing operators.

Operator	Description
<b>UNLESS</b> ( $e_0, e_1, w$ )	Verify the event $e_0$ which occurs within a $w$ interval are followed by the non-occurrence of the event $e_1$ , or the inverse.
<b>UNLESS</b> ( $e_0, e_1, n, w$ )	Identical to the preceding operator but the interdiction to be followed by $e_1$ event only start after the event $e_0$ has occurred $n$ times.
<b>NOT</b> ( $e, SEQUENCE(e_0, \dots, e_k, w)$ )	Special negation operator to verify that the event $e$ doesn't occur during the scope of the sequence.
<b>CANCEL-WHEN</b> ( $e_0, e_1$ )	Cancel the event detection for $e_0$ when $e_1$ occur.

Table 2.2: CEDR conditional and negation operators.

Conceptually, each event is defined as a tuple  $(ID, V_s, V_e, O_s, O_e, Payload)$  where  $V_s$  and  $V_e$  denote valid start and end times, and  $O_s$  and  $O_e$  denote occurrence start and end times.

The syntax of a query is divided into four parts, which will be showcased in Example 2.5.1. The first clause, **EVENT**, is used to define the name of the query. The second clause, **WHEN**, defines the temporal constraints on the observed events. The third clause, **WHERE**, specify data filtering on the events gathered by the temporal predicates. Eventually, a last clause, **OUTPUT**, can map results through a transformation process. If the last clause is not present, all results will be output directly.

While the **WHERE** clause syntax is essentially comparison using first order logic, the **WHEN** clause use predefined operators to specify temporal patterns. Table 2.1 displays event sequencing operators and the Table 2.2 displays conditional and negation operators.

**Example 2.5.1.** Next, an example of CEDR query demonstrate the monitoring of parking slot, which should be paid by hour. `PARKING_ACCESS_IN` and `PARKING_ACCESS_OUT` are respectively the events emitted while a vehicle enter or exit from a slot. `PAY_ADDITIONAL_TIME` is emitted each time the driver pays for one more hour. The query should detect if a vehicle stay inside the parking for more time than what the driver has paid for. The temporal predicate gather events where the vehicles has entered but not exited within the hour, unless an additional payment has been done.

The variables  $x, y, z$  are bound to sub-expressions via AS construct. In this way, they can be referenced in the WHERE clause.

```

EVENT parking_tracking
WHEN UNLESS (
    SEQUENCE (
        PARKING_ACCESS_IN AS x,
        NOT(PARKING_ACCESS_OUT AS y),
        1 hour
    ),
    PAY_ADDITIONAL_TIME AS z, 1 hour)
WHERE {x.Vehicle.Number = y.Vehicle.Number} AND
    {x.Vehicle.Number = z.Vehicle.Number}

```

In terms of expressivity, the CEDR queries provide high level operators with intuitive and well-defined semantics. Compared to formal languages, it is much easier to understand, or modify, the validations.

Regarding general capabilities, patterns are used to identify significant events in a stream. However, in the majority of the operators, the stream is seen as a set of events which does not take into account the intrinsic temporal ordering that the events have in relation to each other. Only the `SEQUENCE` operator validate a strict order between events, but they must be specified one by one. Complex pattern relying on event ordering aren't possible .

For example, to check the place of an element in the stream with respect to an iteration of another event, it would be necessary to iterate on each of the elements that separate the event from the iteration and retain a temporary state. This processing is not supported by CEDR queries, and would require separate developments which could not be optimized by the compiler. In addition, in terms of performance, the structure of the queries causes the difficulty to guarantee minimum or maximum complexity bounds.

In a QRE, the advantage is that the regular structure of the expression determine the event position in the stream. The concatenation of multiple sub-expressions, each one matching with events depending on their positions in the stream, is wrapped up in a global expression. The global expression combine the results of the sub-expressions in a bottom-up way.

However, unlike CEDR, the selection of events in the stream is made filtering them using predicates, and no events validation based on a time interval is built in the language.

## 2.6 Database languages

Rather than building a new formal language, another approach is to extend well-known data management systems and adapt them to real-time processing.

In database management, traditional languages, such as SQL, have been intended to query sets of static data. Subsequently, extensions of these languages have been developed to query frequently updated datasets [10], or the continuous addition of new data. Grouped under the name continuous queries, they communicate with database streaming systems and use query processing in the presence of multiple, continuous, rapid, time-varying data streams. For example, CQL [15] is one of them. It can interact with systems such as Aurora, Borealis, STREAM, and StreamInsight.

CQL inherits operators and syntax from SQL. When the language was designed, it was demonstrated that the sole use of the operators present in SQL is not sufficient to interpret the reading of a data stream and to interact with the relational techniques for reading and updating databases. The language then defines three classes of operators to go from a stream to a relation. A relation is a mapping in which a

finite number of tuples are associated with a precise moment in time, and can then be manipulated by standard database techniques.

To take advantage of this approach and inherit from the formal foundations and huge body of implementation techniques of relational languages, the first class of operators, *relation-to-relation*, is identical to that of traditional SQL, augmented with temporal predicates. In Example 2.6.1, the operators SELECT and FROM are part of this operator class, they were taken from the initial SQL language. To operate on a time-varying relation instead of an instantaneous relation, the operators are executed each instant captured by a relation.

**Example 2.6.1.** For example, the following query fetch speed of vehicles that have transmitted their position within the last 30 seconds [15]:

```
SELECT id
FROM VehicleSpeedPosition [Range 30 Seconds]
```

To capture events over the 30 seconds period, the language requires the second class of operators, *stream-to-relation*. Conceptually the set of temporal predicates which transform a flow into an instantaneous relation are based on the concept of sliding window, which was explained in Section 1.2.3. Three types of sliding windows are programmable in CQL.

**Time-based** windows define a time interval in units of time calculable by the system. The output relation captures the latest portion of the stream covered by this interval. The syntax used is [Range T], where T is the interval. For example, the predicate [Range 15 Minutes] gather all items in the last 15 minutes. Two particular windows, which are [Range Now] and [Range Unbounded], treat respectively only the last and the whole of the events of the stream.

**Tuple-based** windows define a maximum number of elements to capture by specifying a positive integer. The syntax is [Rows N], where N is a number of elements. As time goes by, the window slide forward while always keeping only the last elements. For example, the predicate [Rows 5] always keeps the last 5 elements. The particular case of a predicate accepting an infinite number of elements is noted [Rows Unbounded].

**Partitioned windows** are similar to the Group By operator structure in traditional SQL. They divide the stream into sub-streams based on the value of one or more attributes and combining them with a time-base or tuple-based window. The syntax is written [Partition By  $A_1, \dots, A_k$  Rows N], in which  $A_1$  to  $A_k$  are attributes of the elements of the stream, and N is a positive integer. For example, the query composed with the predicate [Partition By id Rows 5] will apply to the last 5 tuples of each sub-stream containing the elements having the same identifiers, and then will combine the results.

For each of these three operators, a possible slide parameter can be added to specify the interval at which the sliding window must shift. The syntax [Range T] then becomes [Range T Slide U], where both T and U are time intervals. Thanks to this mechanism, a sliding window can be converted into a tumbling window. For example, the predicate [Range 1 Hour Slide 1 Hour] keeps the stream elements hour by hour. After 119 minutes, it still keeps the elements of the first 60 minutes, and changes at the 120th minute to the elements of the next interval.

The last operator class, *relation-to-stream*, projects the results of a relationship to produce a new stream in real time. Three operators make up this class. Istream (R) produces a stream containing only the new values filtered by the relation R. Dstream (R) produces a stream containing only the removed values filtered by the relation R. Rstream (R) produces a stream containing all the changes applied and filtered by the relation R.

**Example 2.6.2.** The following two example requests emit resulting streams. The first one detect when new trucks transmit their position. The second one detect vehicles which was emitting a speed greater than 120 km/h before the last hour, but which have not emitted a speed higher than that since then.

```
SELECT Istream(id)
FROM VehicleSpeedPosition
WHERE type = 'Truck'

SELECT Dstream(id)
FROM VehicleSpeedPosition [Range 1 Hour]
WHERE speed > 120
```

The advantages of the CQL language is its understanding which is quite easy due to the fact that it uses the syntax of the SQL language, which is already well known. However, the scope of use of this language is far away from the other required characteristics. CQL is a more general purpose language whose goal is specifically to be integrated with relational systems, and too little attention is given to performance.

The domain of the input language is not limited to regular patterns in data streams, or another specific grammar, and that handicaps the computability of performance and complexity. Also, their operations are not limited to the aggregation of quantitative properties, as for QREs, which makes it a very expressive language, but this language can be poorly performing sometimes. In summary, their goal is to target a wider area of functionality, with the counterpart of responses that are more approximate and less efficient with regard to pattern detection in streams.

However, some concepts, such as data filtering, inspired the development of operators of other languages, like the QRE language.

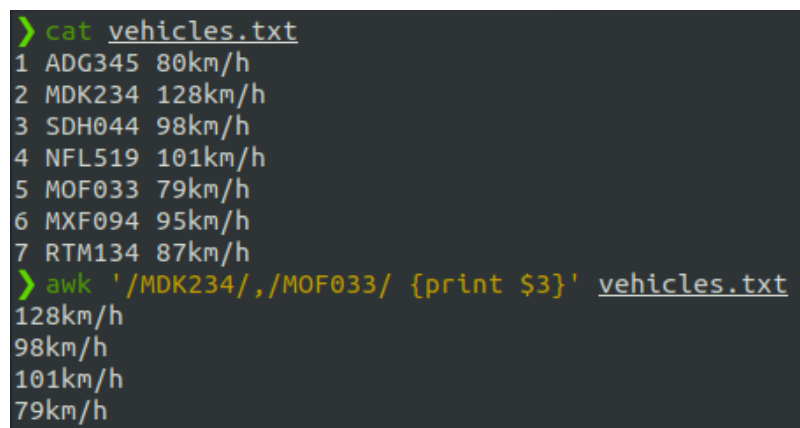
## 2.7 String processing

In the field of word processing, specific tools and languages already exists. Those are usually tools that are very light and can perform various tasks such as reformatting, translating, spell checking. These string processing tools have characteristics that make them quite close to stream processing systems. If the needs of a new language are not limited to a single type of data such as strings, these could constitute a clue for the research of a solution.

Among those, we find:

- AWK, a scripting language used on UNIX systems;
- Perl, an interpreted general-purpose programming language;
- sed (Stream EDitor), a command which can be used to transform the content of a characters input stream

**Example 2.7.1.** In Figure 2.3, an example of pattern detection inside a file containing vehicles and their speeds demonstrate the capabilities of AWK tool set. Here the example query search for all the speeds information, which are in the position 3 one each line, between vehicles with plate number MDK234 and MOF033.



```
> cat vehicles.txt
1 ADG345 80km/h
2 MDK234 128km/h
3 SDH044 98km/h
4 NFL519 101km/h
5 MOF033 79km/h
6 MXF094 95km/h
7 RTM134 87km/h
> awk '/MDK234/,/MOF033/ {print $3}' vehicles.txt
128km/h
98km/h
101km/h
79km/h
```

Figure 2.3: Example string processing query

As explained in some attempts of building a new streaming language relying in them [8], the major problem is that those tools are Turing complete. Therefore, the algorithmic analysis of the scripts, or other programs built, isn't possible.

On the other hand, string sanitizers, coders and decoders have been specifically developed so that they can be easily analyzed. They are generally tied to a specific finite state transducer. Some articles [25] present one of the possible approaches. They have introduced a new effective model for analysis of string

coders, based on a conservative generalization of Symbolic Finite Transducers (SFTs), called Extended Symbolic Finite Transducers (ESFTs).

One of the most famous system in this category is probably UTF-8 encoder and decoder which can transform a word like "Vehicle" to "\x56\x65\x68\x69\x63\x6C\x65\x73".

The downside is that these languages are quite limited in their expressiveness. In addition, the semantics used to design programs is very coupled with the corresponding transducers. The developer is then obliged to know the functioning of the state machine.

The balance between the expressiveness and the complexity of the language is a crucial issue concerning the problems the intended streaming processing system want to solve. As these languages are very tied to string processing, and optimized for it, it makes them too difficult to be detached from this domain. Like the languages presented before, which specifically concern string processing, other streaming processing languages specific to particular domains exist. ActiveSheet [48] analyzes data in real-time using a spreadsheet. XPath and XQuery benefits from extensions integrating data stream processing for XML queries [21].

Even though the precursor of QREs was specific to string processing [8], it quickly broke away from the standard models because their goal was to be parameterizable with several data type, and to be domain agnostic.

## 2.8 ReactiveX

ReactiveX [35, 39] is a combination of ideas from the observer and the iterator patterns, and from functional programming.

Recently, new asynchronous event processing models have been developed with the aim of facilitating the composition of operation on the data streams. One of the best known is ReactiveX. The language uses the Observer pattern, the Iterator pattern, and functional programming to abstracts all details related to threading, concurrency, and synchronization. It is implemented in many existing programming languages (RxJava in Java, RxJS in Javascript, RxSwift in Swift, ...). The interpretation of the syntax may change slightly depending on the implementations.

Focusing on the JavaScript implementation, the main features of the language can be described as follows. The central concept of the language is the Observable object. This object receives and reads each element of the stream. It takes three functions in parameters. A `next` function which allows to configure the operations to be performed on receipt of an item. An `error` function that allows to configure the operations to be performed on receipt of an error. A `complete` function which is executed if the flow ever ends.

**Example 2.8.1.** In the following example a basic structure shows the use of the pattern. The `of` function creates an Observable object simulating a predefined stream which will emit elements 1, 2 and 3 one after the other<sup>2</sup>.

</>
Listing 5: Observer pattern using RxJS
</>

```

1  const { of } = require('rxjs');
2
3  of(1,2,3)
4    .subscribe({
5      next: x => console.log(x * 2),
6      error: x => console.log(-1),
7      complete: x => console.log(-1)
8    })

```

Operation sequentialization is done through multiple built in operators. All operators of the language together are more than a hundred. They give the power for creating data streams, but also to filter the elements, transform them, and handle the eventual errors. Some more complex operators even allow to combine flows between them, or to delay the processing of elements.

<sup>2</sup>The `console.log` function is a standard function in JavaScript allowing to display text in a development console

**Example 2.8.2.** In the following example, the result of which is shown in Figure 2.4, a small part of the operators is highlighted. The flow used in the example represents the speed of vehicles. Three queries are presented. The declaration of each is executed using the `subscribe` operator to observe the results, or is used to compose another query as is the case with the second and third.

```

</> Listing 6: RxJS operators showcase </>
1  const { of, interval } = require('rxjs');
2  const { filter, map, zip, first } = require('rxjs/operators');
3
4  const vehicleStream =
5    of(80, 98, 103, 79, 110, 86).pipe(
6      zip(interval(20), (a, b) => a)
7    );
8  vehicleStream.subscribe(...);
9
10 const rapidVehicleStream = vehicleStream.pipe(
11   filter(x => x > 80)
12 );
13 rapidVehicleStream.subscribe(...);
14
15 const firstRapidVehicleStream = rapidVehicleStream.pipe(
16   first(),
17   map(x => x - 80)
18 );
19 firstRapidVehicleStream.subscribe(...);

```

The first query, `vehicleStream`, shows the generation of the stream from a list of values. Each element of the stream is delayed 20 milliseconds from the previous one. The `zip` operator emits the elements received from the input stream using the interval operator which delays them. The variables `a` and `b` represent respectively the value of each element and their indexes within the stream. Only the value, `a`, is kept. The second query, `rapidVehicleStream`, filters vehicles whose speed exceeds 80km / h. The last query, `firstRapidVehicleStream`, accepts only the first element of the input stream, and output the speed gap compared to 80km/h, using the `map` operator.

The queries results are displayed line by line in Figure 2.4, the lines are ordered the same way the queries are in the example.

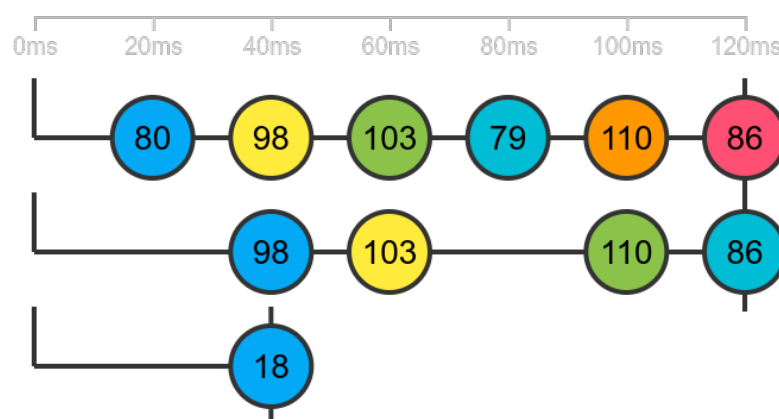


Figure 2.4: Results of RxJS operator showcase example.

The advantage of the language is that it is easily usable in many contexts given the number of implementations that already exist. The number of operators is very large, giving it many possibilities for transforming the flow, but requiring a fairly significant learning curve for being familiar with it. The main flaw is the absence of operators for handling pattern detection, and the impossibility of easily maintaining a state on multiple elements of the stream.

From these problems emerges the lack of tools to control and optimize performance. As calculation of performances is a crucial point for this thesis, ReactiveX cannot be retained as a valid solution.

## 2.9 Conclusion

At present, to try to meet all the required constraints, two categories of algorithms are clearly apparent.

First, some languages are mainly focused on high performance, and processing a large amount of data in real time, but they are also tightly coupled to specific models or domains of application. These systems are most commonly used in sensors and other embedded systems. These are low-level languages which lack of expressiveness, especially when it comes to be adaptable to multiple use cases. Among them, some do not allow sufficiently developed timing management. They are designed to capture the order of arrival of elements as a function over time, but cannot capture the order between them.

On the other hand, some sets of tools for general-purpose languages, which are used for their expressiveness, are very popular, especially in web applications where they must be able to adapt to any type of use case. ReactiveX, for example, is one of them. The counterpart for their expressiveness is the difficulty to predict performance in advance while exploring different algorithm possibilities at design time. They are therefore not suitable as one need is to maintain stable and predictable performance without having to go through multiple testing phases.

The interest in the QRE language, combining performance, expressiveness and flexibility, is no longer to be questioned. It clearly finds its place as a compromise between the two categories that have been listed. The following sections will describe the formalism of quantitative regular expressions by following the chronological order of the QREs creation steps.



## Chapter 3

# Quantitative regular expressions

### 3.1 Introduction

In this chapter, the process which was used for designing quantitative regular expressions language is described in its entirety. The basics of the language were established in the early 2010s and many concepts were added in the years that followed. The chapter will gradually describe these concepts and how they interact with each other in order to trace the evolution of language. The goal of the language is to compute quantitative properties on a stream in an efficient and expressive way. First, the explanations will focus on the formal language theory which establishes a solid ground for creating queries capable of transforming a stream to an expression. Next, a formal modeling approach will explain how the evaluation of these expressions can produce quantitative properties. These foundations give to the queries the ability to guarantee efficient and stable performance but also easily calculable complexity bounds.

Then, the emphasis will shift towards the quantitative regular expression language itself. The language gives an abstract high-level syntax for specifying the queries for computing the quantitative properties. This syntax will be very expressive, easy to manipulate and will lead to the different implementations already set up in the next chapter.

### 3.2 Regular functions

This section is mainly based on the article ‘*Regular Functions, Cost Register Automata, and Generalized Min-Cost Problems*’ [7] written in 2011. The purpose of this article was to define, as a whole, the class of regular functions as a new language. Those functions are the first block for creating queries to transform a stream into an expression, leading to the evaluation of a quantitative property.

To understand the purpose in the creation of a new language, a good place to start is the comprehension of formal language theory, the computational theory of languages and grammars. Following this theory, a language is defined as a finite or infinite set of strings over some finite alphabet. Usually, the set of strings for a language is not finite. Because of that, they are accompanied by a grammar that characterizes them with a set of rules. The rules make it possible to determine whether or not a string is part of the language. More simply, in a language composed of only ‘a’ and ‘b’ symbols, a rule can determine whether the letter ‘a’ alone can be followed by ‘b’. If so, the string ‘ab’ belongs to the language.

In order for a computer to recognize a designed language, a theoretical machine, called finite state automata, is built up. In intuitive terms, it is a very simple model of a computer. This machine reads an input tape containing a string, splitted into symbols, while interpreting the rules of grammar. It changes state as the symbols are read. At the end, the automata determines if the word is accepted by the language.

**Example 3.2.1.** In Figure 3.1, a simple deterministic finite automata recognize all the words of the language alternating between ‘a’ and ‘b’ symbols, while beginning and ending with ‘a’.

Overall, a regular function is set up on this theory, but in addition to recognizing a word, it also attributes

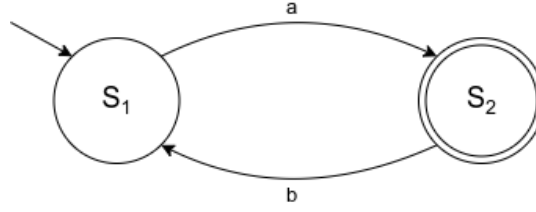


Figure 3.1: Example of a finite automata.

a cost to it. The regular function class is used to create quantitative languages, which assign to each word a value from a particular domain instead of a boolean value [19]. This value is called *cost*, or *quantitative property*. These regular functions are important because their behaviour conform exactly to what is necessary for reading and parsing a data stream. By considering the data stream as being the input word of the regular function, the desired processing result is given by the output quantitative property. In concrete terms, a quantitative regular expression will expresses a regular function by giving abstractions to implement it as a stream processing language.

As a yardstick between the expressiveness and the efficiency of these functions class, a notion of regularity is added. This notion makes it possible to define whether or not a function belongs to this class. Language designers highlight various characteristics, useful for the regular function class, according to which they have adjusted this yardstick. These determine the utility they want to give it:

- Machine independent
- Deterministic
- With closure properties
- Decidable in a given time
- With parameterizable operations

To be deterministic, the transitions from each state in the corresponding automata will be uniquely determined by the next input symbol. The closure properties requirement means that the set of operation handled by a regular function is closed. For a function to be recognized as belonging to the regular function class, it must in particular be written in the form of a composition of terms, each one representing the application of an operation on the stream, and conforming to a specific predefined grammar, which will be explained later on.

The use of weighted automaton, often associated with quantitative properties of finite-state systems, has been ruled out. A weighted finite automaton adds a weight to each transition. The product of the weights of the transitions along the path is the resulting weight of an accepting path through the automata. The computations that these automata are able to produce, are restricted to those defined by a semiring. A semiring consists of a set with two operations, addition and multiplication, satisfying certain natural axioms like associativity, commutativity, and distributivity, just like the natural numbers with their laws for sums and products [28]. In addition, they are inherently nondeterministic.

**Example 3.2.2.** An example of the transformation of the finite automata in Figure 3.1 as a weighted automata is displayed in Figure 3.2.

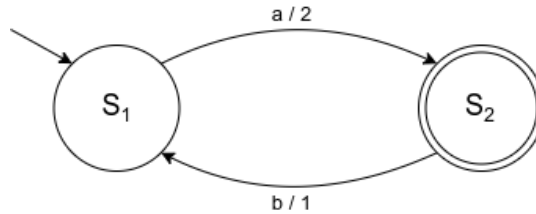


Figure 3.2: Example of a weighted finite automata.

One of the main goals of regular functions being the need for strong expressiveness, this model is not suitable. A new model, along with a new type of automaton, has to be defined to design and recognize

the regular function class. The basics of this model start with the definition of a new alphabet.

### 3.3 Ranked alphabet

An alphabet is a finite, and non-empty, set of symbols. A grammar, build to represent regular functions, is based on an ranked alphabet.

A *ranked alphabet* is composed by an ordinary alphabet  $F$ , and an arity  $Arity$ , which is associated with each of the symbols of the alphabet. It can be represented by a couple  $(F, Arity)$ , and for any symbol  $f \in F$ , the arity of  $f$  is defined by  $Arity(f)$  [23].

**Example 3.3.1.** For example, if the well-known addition operation  $+$  is considered as an element of a ranked alphabet. The arity of this operation will be  $Arity(f) = 2$  because this is a binary operation which has two operands.

Considering that all symbols with the arity  $p$  in the alphabet  $F$  are noted  $F_p$ , this alphabet can be separated into two subsets:

- The *constants*:  $F_0$ , a constant symbol will be noted  $c \in F_0$
- The *functions* of  $p$ -arity, noted  $F_p$

This separation is important to keep in mind. The interpretation of those two groups of symbols will be different all along this document.

The basic expressions that can be constructed with such an alphabet are called *terms*, or expressions, in reference to the mathematical structures. The terms of the alphabet  $F$  are noted  $T_F$ , such that:

- if  $p = 0, \forall c \in F_0, c \in T_F$
- if  $p > 0, \forall f \in F_p$ , and  $t_1, t_2, \dots, t_p \in T_F$ , then  $f(t_1, t_2, \dots, t_p) \in T_F$

Each regular function will interpret a language based on a predefined alphabet, and parse its words as the terms of a ranked alphabet. The compilation will be handled by an automaton, which should be capable of writing its corresponding term while reading the input word. For this to be possible, a set of predefined rules should specified how to transform each possible word of the input language. These are defined by a grammar.

### 3.4 Cost grammar

Without a predefined grammar complementary to a ranked alphabet, multiple terms could be created for each word of an input language. However, to efficiently compile an input language, only one single term should be deterministically associated with each input word. To narrow the scope of each language, a set of rules should be added to it. A *cost grammar*  $G$  is specified by a tuple  $(F, T)$ , where  $F$  is a ranked alphabet, and  $T$  is a regular subset of its composing terms  $T \subseteq T_F$ . The rules of the grammar determine the construction of the terms for a language, as well as which existing term belongs to the language, or not.

The structure of these terms, called *cost terms* [10], is generally recursive. This is due to the use of a ranked alphabet. They can be represented by a tree in which the minimum term, present in each leaf of the tree, corresponds to a constant.

**Example 3.4.1.** Let  $F = \{max, +, c\}$  be a ranked alphabet. Here  $max$  and  $+$  are binary symbols, and  $c$  is a constant symbol.

A cost grammar  $G(max, +, c)$ , using this alphabet, contains the terms defined by the rules  $t := max(t, t) \mid +(t, c) \mid c$ . As an example, one of them can be written  $max(c, +(c, c))$ . It can be represented by the tree in Figure 3.3. The terms accepted by the language, and belonging to  $T$ , are only those who follow these rules.

In a data streaming context, this kind of pattern is applied to evaluate the resulting quantitative properties of the stream. The property resulting from the evaluation of a stream will actually be the interpretation of a term built while following the rules of a cost grammar.

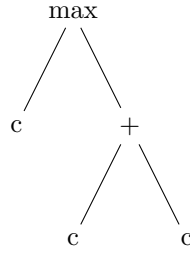


Figure 3.3: Term as a tree

For example, a traffic monitoring algorithm receiving a stream containing the kilometers traveled each hour all along a day by a vehicle can be aggregated into a single term. Using only the  $\max$  operation of the grammar defined previously, the maximum number of kilometers a vehicle has traveled within an hour during the day can be obtained as follows, in Figure 3.4.

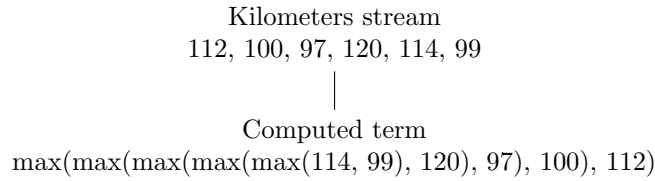


Figure 3.4: Maximum kilometers traveled by hour as a term

### 3.5 Cost model

A *cost model*,  $(D, G, [[\cdot]])$ , defines the semantic of a cost grammar  $G$ . The function symbols are interpreted by  $[[\cdot]]$ , depending on the nature of the function, over a domain  $D$ :

- A constant  $c$  is interpreted as a unique value:  $[[c]] \in D$
- A  $p$ -arity function  $f$  is interpreted as  $[[f]] : D^p \rightarrow D$ , and each of its term can be inductively interpreted.

The *cost domain*  $D$  is a finite or infinite set.

**Example 3.5.1.** Let's build a cost model based on the example grammar  $G(\min, \max)$ . The model  $C(Z, \min, \max)$  denotes the successive usages of the  $\min$  and  $\max$  functions on strings interpreted by the integer domain. For example, a term  $\max(3, \min(5, 6))$  characterized by this cost model, interpreted on the integer domain, would give the result 5.

Optionally, there is the possibility to add more constraints on the different operators. As an example, let's enable the first function of the model,  $\min$ , to always be applicable to the whole domain, but restrict the following function,  $\max$ , to a subset of this domain. That kind of restriction can be specified by preceding the concerned function by the desired restriction in the model definition.

The model  $C(Z, \min, [0, 20], \max)$  restricts the  $\max$  function domain to the input symbols between 0 and 20.

By taking the term that had been constructed in the Example 3.4.1 of the previous section, and by associating its grammar with the integer type, the cost model obtained is  $C(Z, \max, +c)$ . The result of the interpretation of the example cost term is the one displayed in Figure 3.5.

#### 3.5.1 Cost type

The symbols of the input strings that are mapped to a cost model always belongs to a specific alphabet, as well as the values produced by this model. As explained in reference documentation [10], each such alphabet is called a *cost type*. It's set up from a domain, a subset of a domain, or a combination of multiple domains.

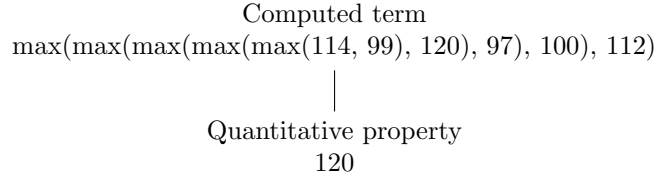


Figure 3.5: Integer interpretation of maximum kilometers term

The set of the cost types is represented  $\mathcal{T} = \{T_1, T_2, \dots\}$ . Each domain or subset  $T_i$  must be non-empty.

**Example 3.5.2.** For example, the set  $T = \{t, c, m\} \cup \{end_H\}$ , can represent a road traffic symbology, where  $t, c$  and  $m$  are respectively truck, car and motorbike symbols, and  $end_H$  marks the end of an hour.

### 3.5.2 Cost register automata

All the required tools to build the languages of the regular function class are now present. A regular function is built to transform an input language into a cost term. This cost term is written by following the rules of a cost grammar, which stands on the properties given by a ranked alphabet. Each cost term can then be interpreted by a cost model according to the domains of its evaluation. These domains are defined by cost types.

The creation of an automaton capable of interpreting the cost terms is needed. A *cost register automata* (CRA) is a deterministic finite state machine, that simulates the interpretation of the regular functions defined by cost model. It maps strings, that belong to the input language, to cost terms, and interprets them to a cost value depending on the types associated with the model. This *cost value* will be the quantitative property sought in the context of streaming processing. Given the inheritance of the cost register automata from the finite state machine properties, each input language must respect several constraints, and in particular the capabilities to be described by a *regular expression*.

Unlike finite automata, a cost register automata configuration is not only determined by its input head and current state, but also by a set of registers and their value. These registers will retain the intermediate values for the production of a quantitative property during the parsing. Each register is initialized by a constant.

The cost grammar  $G$ , while augmented with a set of registers  $X$ , make up a set of expressions  $E(G, X)$ . This set is used in order for the automata to execute register assignment expressions at each transition. Registers should be used in *write-only* mode. They are not tested, and they must only be used once in the right side of each expression. In this way, they ensure the copyless restriction, which will guarantee good performances as it will be explained in Section 3.5.3.

**Example 3.5.3.** The grammar  $G(max, +c)$ , augmented with a set of registers  $X$ , defines the set of expressions  $e := max(e, e) \mid +(e, c) \mid c \mid x$  with  $x \in X$ .

Some examples of expressions are  $z := max(z, 10)$ ,  $u := max(y, z)$  or  $y := z + 10$  with  $u, y, z \in X$ .

#### Automata model

The automata is represented by a sextuple:  $\langle Q, \Sigma, q_0, X, \delta, \rho, \mu \rangle$ , containing

- $Q$ : A finite set of states
- $\Sigma$ : A finite alphabet of inputs
- $q_0 \in Q$ : A start state
- $X$ : A finite set of registers

The relations  $\delta, \rho, \mu$  define the way the cost automata reads an input tape. The next explanations are in relation with Figure 3.6. This figure represents a cost register automata, reading input stream symbols  $a_i$  from 1 to  $n$ .

1. The function  $\delta : Q \times \Sigma \rightarrow Q$  reads the next input tape symbol, determining the next transition, and thus the next state  $q \in Q$ .

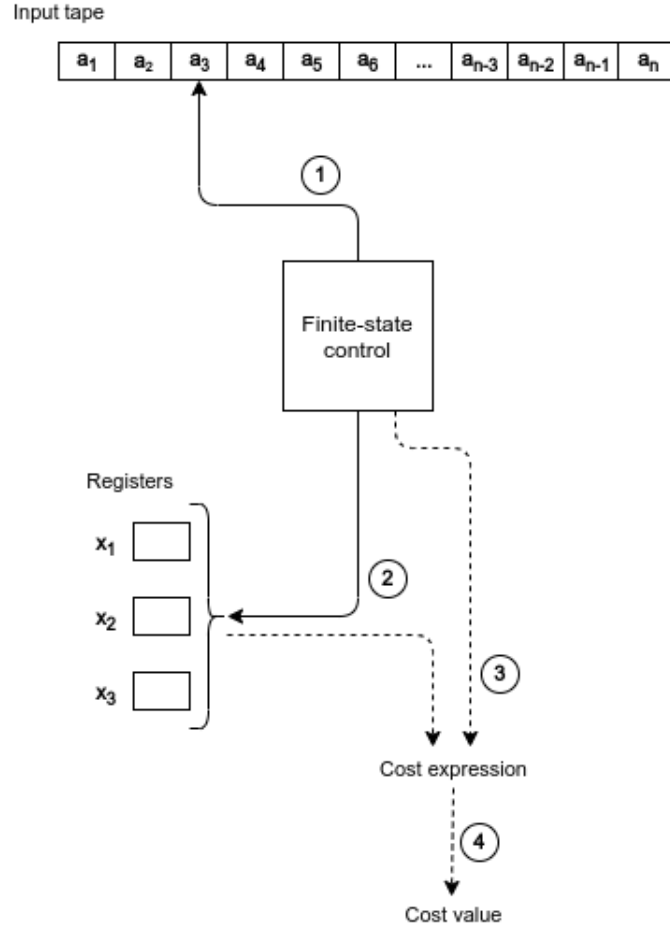


Figure 3.6: Cost register automata schema

2. In parallel, depending on the transition, the function  $\rho : Q \times \Sigma \times X \rightarrow E(G, X)$  updates the registers  $x \in X$ , using expressions defined by the cost grammar  $G$ .

The next functions are partial. They depends on the ability of the cost automata to recognize the string.

3. The function  $\mu : Q \rightarrow E(G, X)$  determines, at any time, the resulting expression according to the current state of the automaton, and depending on the input symbol to which it has stopped. For this function to be defined, the input head shouldn't be on a specific symbol, and there's no final state to reach, but the automata must have recognize the preceding symbols.
4. If so, the function,  $[[M, C]] : \Sigma^* \rightarrow D$ , is defined. This is the *cost function*, mapping the final expression to a cost value, according to the interpretation  $[[.]]$  defined by the cost model  $C$ . If not, the produced cost value stay undefined.

The input is considered accepted by the CRA if the output function is defined.

The design of the function  $\mu : Q \rightarrow E(G, X)$  corresponds well to the desired goal to be accomplished. This idea of not having to wait for a final state to be reached, for producing an output, fits to the processing of a stream, in which results are produced throughout the items arrival and not only at the end.

From the conception of quantitative properties languages to their evaluation, the overall behaviour of the class of regular functions is now complete.

**Example 3.5.4.** As an example for cost register automata, let's suppose that an automatic tolling system tracks the vehicles to avoid traffic jam all along the days. The system is fed with a stream on the input alphabet  $\Sigma = \{c, m, t, H\}$ , where  $c$  represents a car,  $m$  represents motorbike,  $t$  represents a truck, and  $H$  marks the start of a new hour.

The automata of Figure 3.7 measures the traffic density attributing a weight to each vehicle (motorbike: 1, car: 2, truck: 4 or 3). It computes the iterative sum of these weights during each hour, and keeps the maximum density compared to preceding hours. It uses two registers,  $x$  and  $y$ , that are initialized at the value 0.

As part of this simulation, the impact of the trucks, inside the road traffic, which are directly following another truck is estimated as being much lower than for an isolated truck. For that purpose, the calculation of the truck weight implies that all the directly following trucks weigh less, their cost is lowered to 3 instead of 4. It demonstrates that previous symbols can influence the behavior of the followings.

The whole example is computed over the cost grammar, previously defined,  $G(max, +c)$ . The final cost functions are  $\mu(q_0) = y$  in  $q_0$ , and  $\mu(q_1) = y$  in  $q_1$ . As long as the input stream respects the language alphabet, it is possible at any time to retrieve the cost value computed on the whole cost term.

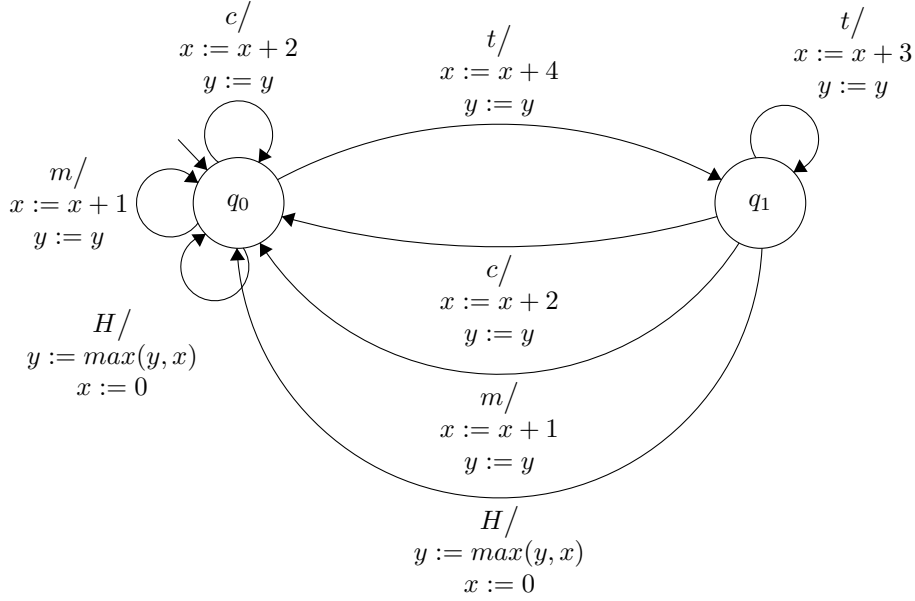


Figure 3.7: Cost register automata example

In the next sections, and especially from Section 3.6.1, the QREs will set up a layer of abstraction above the cost models. However, the models, and their interpretation by CRAs, will retain all of their properties. The partial evaluation of an input tape will be transposed in the structure of the QREs. The selection of different regular expressions will be possible thanks to their hierarchical construction and their associated regular expressions.

### 3.5.3 Streaming String-to-Tree Transducers

To determine the expressiveness and the decidability of the problems computed by a cost register automata, the designers of regular functions [7] asserts the equivalence of CRA with another model, the Streaming String-to-Tree Transducers (SSTT).

A cost grammar creates terms that can be represented as a ranked tree, as shown above. Under the conditions set out in the next subsections about single pass processing, expressiveness, and decidability, the transformations on all of these ranked trees can be determined in a decidable and expressive way by a SSTT, and so are the problem resolved by a CRA.

This section is based on the article ‘*Streaming Tree Transducers*’ [6].

#### Single pass processing

A streaming tree transducers (STT) is a deterministic machine model built for the transformation of ranked tree. Cost grammar terms can be encoded in the form of nested words. Nested words are structures representing ordered trees linearly. To build them, calling and return symbols are combined

with any symbol of the cost grammar alphabet. The symbols  $<$  and  $>$  in the XML language are, respectively, a good example of those special symbols.

The nested words can contain *holes*. It's a mechanism that delegates the completion of the current term to the following symbols transduction, leaving an empty space in it, the hole. This hole is explicitly added in an expression by the symbol  $?$ , which is therefore added to the expression grammar.

Complementarily with the holes, the substitution, or tree insertion, functionality is also added to the expression grammar. Represented by the symbol  $e[e']$  for an expression  $e$ , it leads to the insertion of a new expression  $e'$  to replace the symbol  $?$ .

As the transducer reads the inputs from left to right, the holes and the substitution method is the only way to handle non-commutative operations. The transformation uses a pushdown stack and a finite number of variables. It can be implemented with a *single-pass linear-time* algorithm, and computes *linearly-bounded outputs*, but it implies two restrictions:

- Each expression, represented by a nested word, can contain at most one hole at a time.
- Copyless restriction requires that a variable can only be used once in the same right-hand side assignation. For example  $x := x + x$  is forbidden. On the other hand, the successive assignations  $x := x$  and  $y := x$  are allowed provided that  $x$  and  $y$  are not combined in future assignments. This combination would cause a so-called conflict relation. These two restrictions are grouped under the name *Single Use Restriction*.

**Example 3.5.5.** Let's re-use our cost grammar  $G(max, +, c)$  with its ranked alphabet  $F = max, +, c$ . In the term  $max(+ (2, 1), + (4, 3))$ , calling and return symbols are respectively  $'('$  and  $')$ .

The set of expressions based on this grammar is modified with hole and substitution symbols, and written  $E^?(X, G)$ . The construction of the terms can then be achieved by  $e := max(e, e) \mid +(e, c) \mid c \mid x \mid ? \mid e[e]$ .

In fact, this is not totally correct, because with this grammar, expressions are allowed to contain multiple holes. For example, the nested word  $x := max(?, ?)$  should be forbidden. To avoid that, the grammar can be split in two parts,

- $e' := max(e, e') \mid max(e', e) \mid +(e, c) \mid c \mid x \mid ? \mid e[e]$ , and
- $e := max(e, e) \mid +(e, c) \mid c$ .

As an example, using the above expression grammar, the transducer can create the expression:  $max(max(+((0, 4), 2), +((0, 1), 4)))$ .

### Expressiveness

The class of problems solvable by a streaming tree transducer is equivalent to the problems definable by monadic second-order logic (MSO).

That particular logic expands first order logic. It describes properties on structures using predicates. These predicates use quantifiers such as  $\exists$  or  $\forall$  on two types of variables [24]:

- Object variables ( $x, y, z$ ) which belongs to a set.
- Set variables ( $X, Y, Z$ ) which represent sets or subsets.

It is said to be monadic (or unary) because these predicates can only use one argument. The expressions defined by MSO are commonly used to define from graph to graph transforms.

Nested words can be encoded as binary trees. We can define transductions from a nested word to a binary tree and vice versa. By transitivity, the class of transductions made on nested words is therefore equivalent to those on binary trees.

The transductions from binary tree to binary tree being MSO definable, it implies that those on nested word is also.

### Decidability

One way to check if a transduction is definable by a STT, is to check that it can be processed by an STT with *regular look ahead*.



A STT with regular look ahead is a modified version of the transducer. It can make its decisions depending on the belonging of the remaining suffix of the input word to a regular language.

To verify that condition, the suffix of the word that is being read, must be able to be recognized by a DFA, and more specifically by a NWA<sup>1</sup> in the case of nested words, that would read it in reverse. The STT model, like the CRA model [7], is closed under that property. A transduction executed by a STT is always STT-definable.

### Analyzability

Due to all the previous restrictions and properties, the transductions can be computed using a fixed number of variables  $k$ . Therefore, the outputs of a transduction on a word  $w$  are computed in time  $O(k * |w|)$ . The time complexity is constant in the length of the words. The *functional equivalence* between two STTs is established if they define the same transductions and recognize the same words. It can be verified in NExpTime<sup>2</sup>.

The trade-off between analyzability and expressiveness is common in formal language theory and subsequently allowed to predict the performance of the algorithms using them [1].

To be able to have the same performance and expressiveness properties, the restrictions on the STT transductions have been transposed into the CRA model, explained in Section 3.5.2. Therefore, a CRA can be seen as a variant of a STT.

## 3.6 Regular combinators

Regular combinators play a structuring role for the creation of regular functions. As has already been demonstrated, regular functions are very expressive. However they require a certain rigor to be able to be used extensively. It is not only required to know how to create a language with a syntax and a grammar for expressing the desired query, but it is also necessary to ensure that these comply with the properties required to be processed by a cost register automata.

Due to that difficulty of implementing complex regular functions, the combinators place themselves as an intermediary between their formal definition and an implementation in a high level language. They allow not to lose expressiveness and offer an abstraction which retains their properties. The formalism of regular combinators and regular functions, taken together, constitutes the quantitative regular expressions (QREs). If the general functioning of these expressions has already been explained previously, in this section, they will be deeply analyzed starting from the theoretical bases which have just been defined before.

The additional abstraction of regular combinators first appeared in the article ‘*Regular combinators for string transformations*’ [11], and was further defined later in ‘*Regular programming for quantitative properties of data streams*’ [10].

### 3.6.1 Structure

QREs, pronounced queries, are a flexible formal language for specifying queries over data stream. A QRE executes a regular function over a specific language. It uses a recursive structure to decompose a complex regular function into several simple ones. These simple regular functions are combined by the use of their regular combinators structure. The language on which a QRE is processed is defined by a regular expression, to comply with the transduction decidability constraints of a STT and the parsing capabilities of a cost register automata.

Other existing languages already use regular expressions. CQL Continuous Query Language and CEDR, mentioned in the creation documents about the QREs [37], and explained in the previous chapter, are some examples. However, the matching elements do not preserve temporal ordering unlike QREs. Since each streaming item recognized by a QRE is part of the domain of a regular expression, the item has a well-defined place in relation to the others inside the streaming processing query.

---

<sup>1</sup>Nested word automata

<sup>2</sup>Non deterministic ExpTime

If a QRE is an abstraction of a regular function, the query retains its advantages over performance and expressiveness, and the differences lie in the way of specifying and structuring the regular function. During this section and the followings, the correlation between the regular functions and QREs is explained. As a first step, it is useful to take a step back on the relationship between QREs and regular languages and focus only on the specification of the general structure of a QRE,

$$\underbrace{[[f]]}_{\text{QRE}} : \underbrace{D^*}_{\text{Data domain}} \rightarrow \underbrace{(T_1 \times T_2 \times \dots \times T_k \rightarrow \underbrace{C}_{\text{Cost value}})}_{\text{Cost term}} \cup \underbrace{\{\perp\}}_{\text{Undefined}}$$

Like the regular functions, the model of a QRE is defined over a set of cost types  $\mathcal{T}$ , previously explained in Section 3.5.1.

As *input*, it takes a stream  $w \in D^*$ . Each of the elements from this stream comes one by one, and belongs to a data domain  $D \in \mathcal{T}$ .  $D^*$  is the set of all possible data stream from this domain. The input language is the counterpart of this domain in the context of a regular function.

The *output* of a QRE can be two things. If the regular expression *matches* the element, it produces a cost value that belong to a cost domain  $C$ . This output is generated by a function that takes a set of parameters  $X = (x_1, x_2, \dots, x_k)$ . If the expression *doesn't match*, it returns the undefined value  $\perp$  [3].

The data domain, the cost domain and each parameter belong to a cost type defined in  $\mathcal{T}$ .

**Example 3.6.1.** This decomposition of QRE can be easily applied on an example. This example will intentionally be very simple and does not represent all the capabilities of QREs, but only the basics. More advanced possibilities will be introduced next.

In this case, the purpose of the query will be to calculate the number of vehicles entering a highway. The input stream is an integer stream producing an iteration of a single value, the value 1. Each time a new vehicle enters on the highway, a new 1 symbol is emitted. The cost term is composed by multiple integer type parameters, combined with the addition operation. The evaluation of the cost term produces the total of the number of vehicles read as a result.

The concrete structure gives a QRE construction such that,

$$[[f]] : \text{Integer}^* \rightarrow (\text{Integer} \times \text{Integer} \times \dots \times \text{Integer} \rightarrow \text{Integer}).$$

And in application, a flow detecting four vehicles would give the result,

$$[[f]] = 1, 1, 1, 1 \rightarrow (1 + 1 + 1 + 1 \rightarrow 4).$$

Then if a fifth vehicle comes in, the result would be modified in,

$$[[f]] = 1, 1, 1, 1, 1 \rightarrow (1 + 1 + 1 + 1 + 1 \rightarrow 5).$$

And so on.

In the case where a single value in the input stream would not be 1, the query evaluation would produce an undefined result,

$$[[f]] = 1, a, 1, 1 \rightarrow \perp.$$

### 3.6.2 Term

In the previous section, the overall behaviour of QREs is described as identical with the one of regular functions. Until now, however, the queries seem very limited. This is actually the approach for building the cost terms that sets them apart.

The cost terms, or cost expressions, are the operations produced by the QREs. To refer to the linguistics used in the construction of a cost register automata, a cost term is equivalent to a word written using a cost grammar, recognized and computed by the automaton. As described in regular programming writings [10], just like the standard regular functions, the cost model assign at each cost term an interpretation.

Even if a cost term is a shared feature between regular functions and QREs, this common point seems to end when we focus on the structure of their operations.

### Operations

Initially, a regular function maps input strings to various algebraic structures. Instead, the QREs are initially, and intentionally, limited to map input strings to monoids [11]. A monoid is an algebraic structure with an associative internal composition law and a neutral element. Which means that each QRE can only apply one single binary function  $\times$  on the input stream. The *cost operation* structure is as follows,

$$op : \underbrace{T_1}_{\text{Parameter type}} \times T_2 \times \dots \times \underbrace{T_k}_{\text{Binary operation}}$$

In all known specifications, the description of the operations remains general, expressed as above as a  $k$ -ary operation, not excluding the possibility of adding more complex ones in the future. However, it's clearly affirmed [11] that, if they are commutative, the monadic operations are sufficient to express any possible regular function when used in conjunction with a QRE.

**Example 3.6.2.** As it stands, an operation such as counting the number of cars is completely accepted by the axioms which govern the algebraic structure of a monoid. However, if the goal is to calculate the maximum density of road traffic per hour, as could have been done in Example 3.5.4 with an cost register automata, this is currently not possible.

A currently valid operation is:  $1 + 1 + 1 + 1 + 1 + 1$ , with the grammar  $G(+c)$ .

A currently invalid operation is:  $max(2 + 1 + 3, 2 + 3 + 1)$ , with the grammar  $G(max, +c)$ .

### Parameter

The limitation applied to the operations with regard to the use of a single binary function may seem to reduce the expressiveness, compared to regular functions. This loss of expressiveness is actually nonexistent, because the cost operations, as they have been introduced above, is only one part of the whole grammar of regular combinators. The latter is enriched when taking into account the possible construction of the *operations parameters*.

Conceptually, as explained in the global combinator structure documentation [2], to each type of parameter  $T_i$  can be assigned an effective parameter  $x_i$  such that the type of  $x_i$  is  $T_i$ , and the set of effective parameters is noted  $X = \{x_1, x_2, \dots, x_k\}$ . In general, the assignment of a parameter to a type is written  $x : T$ .

A cost term  $\tau$  is a construct combining cost operations with effective parameters. The cost term grammar is a little more complex than what was apparent in the general QRE structure, such that:

$$\tau := x \mid t \in T \mid op(\tau_1, \tau_2, \dots, \tau_k)$$

The third part of this grammar,  $op(\tau_1, \tau_2, \dots, \tau_k)$ , introduces the fact that cost operations can be nested one inside the other by means of their recursive structure. As shown more explicitly in the StreamQRE language [37], this is implemented by the explicit use of combinators as a parameter for other combinators.

Therefore, the evaluation of the parameters of an operation is divided into two cases due to their recursive structure. The basic case of the recursion corresponds to the interpretation of an element of the data stream itself. In this case, the operation is executed on the element directly. In the other cases, the use of a nested combinators as parameter requires to first evaluate the cost value produced by this combinator, then to execute the operation on the result of the evaluation.

The declination of cost operations according to a hierarchical structure exactly reproduces the behavior of the constants and functions of a ranked alphabet.

**Example 3.6.3.** Operations previously invalid due to the use of multiple operators can now be specified. Each cost term respects the limitations assigned to it by using a single associative binary operation, but combined with each others they allow to define a more complex structure.

The following two cost terms allow to specify the operations necessary to express the grammar  $G(max, +c)$ ,

$$\begin{aligned}\tau_1 &= \tau_2 + \tau_3 \\ \tau_4 &= \max(\tau_5, x_6).\end{aligned}$$

The operation  $\max(2 + 1 + 3, 2 + 3 + 1)$  now becomes valid. It remains to be determined the domain of the stream on which each operator should be used.

Another full demonstration of the expressiveness of the regular combinators, using a recursive structure, has been showcased in the context of medical studies [3], where QREs are reliably used to process signals that reflects the activity of the heart to detect critical irregularities.

### 3.6.3 Domain

As the grammar of cost terms used by the QREs is recursive, it imparts to the input data stream a logical hierarchical structure. Each QRE in the hierarchy must know how to deal with each symbol of the stream [37]. Therefore, for their operations to be defined and executed, it requires to divide the input stream, at each level of the hierarchy, into several chunks. In the case of a cost register automata, this separation of the operations was induced automatically by the construction of the automaton and its corresponding regular expression. For QREs, the domain definition is comparable but slightly different.

The domain of a QRE is the set of sequences  $w$  which belong to a data domain  $D^*$  such as:  $[[f]](w) \neq \perp$ . Where a deterministic cost register automata had only one way to process each of the elements of the stream, as now, a QRE has a large number of ways to split the stream into multiple parts, and eventually to use each of them as a parameter. The syntax is currently too expressive to represent a class of functions identical to those expressed by a CRA, and would regress to being equivalent to the use of a non-deterministic automaton. Therefore, some limitations must be added.

In derivative of quantitative regular expressions studies [13], the different possibilities of splitting the stream and applying operations on it are represented by a *multiset semantic*. Consider a stream  $w$  which can be split as,

$$w = u, v_1, v_2, \dots, v_n$$

where  $u$  is the prefix of the stream, and if  $n \geq 0$  then  $\forall i > 0 \wedge i < n : v_i \neq \epsilon$ . It's demonstrated that there is an exponential number of possible separations depending on  $n$ , the length of the stream. This eventually leads to different cost values per splitting possibility.

To avoid this the QREs provide a way to identify and restrict their data domain by a data languages  $L$ . Indeed, each QRE is associated with a *symbolic regular expression*. In the same way that a language defined by a regular expression is recognized by a finite state automata, their symbolic counterparts are recognized by a symbolic automata [26], which allows transitions to carry predicates over a boolean algebra.

**Example 3.6.4.** By leaning back to the calculation of the maximum vehicle density per hour, if the operations are now specifiable, without symbolic regular expressions, it is impossible to know which operation should be applied to which subset of the stream. It would be necessary to compute all the possible set of properties to finally find the only one which is useful.

On the stream containing the elements 2, 3, 1, 2, the possible operations would be,

$$\begin{aligned}&\max(2, 3 + 1 + 2) \\ &\max(2 + 3, 1 + 2) \\ &\max(2 + 3 + 1, 2) \\ &\max(2 + 3 + 1 + 2) \\ &\max(2 + 3, \max(1, 2)), \text{ and more.}\end{aligned}$$

### Predicates

Before understanding the capabilities of symbolic regular expressions, the concept of predicates should be defined as they are used in these expressions. Predicates over  $D \in T$  are noted  $\phi_D$ . Each of them

always expresses a boolean function. For example  $\phi_D = \{d \in R_+, d \neq 20, d < 3\}$  are valid predicates. The predicates must be decidable and satisfiable within a finite time, solvable by a SMT solver.

**Example 3.6.5.** In the context of traffic monitoring, several useful predicates are the ones used to discern the types of vehicles, such as,  $type = m$ ,  $type = c$  and  $type = t$ .

### Symbolic unambiguous regular expressions

A symbolic regular expression, written  $[[r]] \subseteq D$ , determines the language associated with a QRE. As the input is read by the function, the regular expression recognizes the prefixes that trigger the production of the output.

Depending on the reference documents, symbolic regular expressions can also be called the *rate* of a QRE, like in [13], where  $rate(f) = [[r]]$ , with reference to the frequency at which the function matches with the input stream.

To guarantee uniqueness of parsing at compile time, only regular expressions which are *unambiguously parseable* are considered. It means that the input stream must only be decomposable in, at most, one way to match the language described by the symbolic regular expression. In this way, the problem of maintaining all splitting possibilities is avoided and the class of problems solvable by a QRE will be therefore equivalent to the one of a deterministic CRA.

The way in which parsing constraint alternates the formalism of symbolic regular expressions is described such as [2]:

- Empty string  $\epsilon$  is always accepted as a symbolic regular expression.
- Predicates, as specified in the previous section, are symbolic regular expressions.
- Union of symbolic regular expressions is unambiguous, and accepted, if both are disjoint. Therefore,  $[[r_1 + r_2]] = [[r_1]] \cup [[r_2]]$ .
- Concatenation is accepted and defined as  $[[r_1 r_2]] = [[r_1]][[r_2]]$  if both symbolic regular expressions are unambiguously concatenable.
- Iteration is accepted as a symbolic regular expression, and defined as  $[[r^*]] = [[r]]^*$ , if the expression  $[[r]]$  is unambiguously iterable.

Those concepts of predicate and ambiguity are also described extensively in the initial quantitative regular expression specifications [2, 37].

As can be seen, symbolic regular expressions retain all the properties of regular expressions and the language of the input stream for feeding a regular function must follow these properties. Therefore, all the techniques used to recognize a regular language can be used to know if a stream can be processed, or not, using a regular function or a QRE.

One of the best known techniques for checking that a language is regular is the pumping lemma. For the patterns that need to be recognized in the input stream, it should be verified that all sufficiently long strings in the language, defined by this pattern, have a part that can be pumped. Pumping refers to generating many new input strings by pushing a symbol in a middle section of the current string again and again, with the guarantee that each new string is also part of the language.

**Example 3.6.6.** Let's take an example to realize the impact that the symbolic regular expressions have on the stream. In the case of calculating the maximum density of vehicles per hour, the structure of a stream, as it was defined in the previous examples, is only composed of symbols indicating the presence of a vehicle, or not. This structure does not enable to differentiate them based on the timing of their arrival from one hour to the next. In other languages, a timestamp could have associated with each item, allowing to aggregate them in different datasets.

In other words, in the following stream, it is impossible to make a separation between the items if they are emitted at different times,

$$\text{Stream: } \underbrace{2, 3, 1, 2}_? \quad \underbrace{3, 1, 1}_?$$

While in the following updated version of the stream, in which the symbol  $end_H$  represents the end of an hour, it's easily achievable,

$$\text{Stream: } \underbrace{2, 3, 1, 2}_{2+3+1+2} \text{ } end_H \underbrace{3, 1, 1}_{3+1+1} \text{ } end_H$$

$$\underbrace{\hspace{10em}}_{max(2+3+1+2, 3+1+1)}$$

The regular expression associated with the language is noted  $(x^* \cdot end_H)^*$ , where  $x \in \mathbb{N}^*$ .

This means that temporal inconsistencies, due to a network problem or a discrepancy between items coming from different sources, must be fixed upstream. This can be seen as a major flaw in this language, where other algorithms are able to easily reorder items. However, the other algorithms do that with the charge of an additional cost in performance.

By considering the constraints induced in the language by symbolic regular expressions, some problems are absolutely impossible to solve. To give an example, an algorithm that would be impossible to build with the sole use of quantitative regular expressions would be one to ensure that the number of cars entering a parking lot equals the number of trucks that also entered it. This would require recognizing a language defined by a regular expression close to  $c^n t^n$ , where  $t$  is a truck,  $c$  is a car and the exponent is an integer such that  $n > 0$ . However, this language is not regular.

### 3.6.4 Evaluation

Generally to designate a function as being a QRE, the written representation is  $f : QRE(D, C)$ , where  $D$  and  $C$  are the data and the cost domain. A second notation  $f : QRE(r, X, C)$ , presents in particular in one early article [2], describes more explicitly the query. The three components  $r$ ,  $X$  and  $C$  are respectively, the language, defined by a regular expression, the sequence of parameters and the cost domain of the QRE.

That notation makes it more easy to explain how a QRE is evaluated. A QRE is defined when its operation from a cost domain  $D$  to a cost domain  $C$  can be executed. To achieve that, the associated regular expression  $r$  must match with the input stream. Consequently, each parameter,  $x \in X$ , being part of the operation cannot be undefined.

The evaluation of a QRE is also restricted by its parameters. If the operation of a QRE has as input the result of operations of other QREs, due to their hierarchical structure, the children QREs must emit synchronized results. Explaining it more conceptually, if the parameters assigned to an operation,  $op : T_1 \times T_2 \rightarrow C$ , are two QREs,  $f_1 : QRE(r_1, X_1, C_1)$  and  $f_2 : QRE(r_2, X_2, C_2)$ , they must have the same rate,  $[[r_1]] = [[r_2]]$ , for the operation to be executed. Their symbolic regular expressions must match with the stream, in the same time, on the same word.

## 3.7 Regular combinator types

A QRE performs the application of regular functions on a stream depending on symbolic regular expressions. The problem remains that given the general structure of an QRE, it is difficult to easily write queries where the regular expressions and the customizable operations work together. To give a structure to the implementations of those expressions, the QRE syntax will offer several abstractions, simple to implement, and allowing to define any regular function. These are called regular combinators types.

In other words, instead of implementing a symbolic regular expression from scratch for each QRE, the query will be coupled to a predefined expression pattern. From a general definition of QRE, a finite number of combinators can be actually implemented, each one being complementary to the operations of QRE.

For example, the Kleene iteration pattern from regular expressions isn't explicitly written in a QRE, like  $QRE(r^*, X, C)$ . Instead, a specific combinator, *iter*, allows to apply it. Such a combinator has the definition  $QRE : iter(r, X, C)$ . Its internal structure will take charge of the iteration pattern in its implementation. The two QRE notations, with the keyword *iter* or without, are exactly identical in their behaviour, and only mark the difference between the definition of a QRE and its implementation as a type of combinator.

A finite number of different combinators exists. These have been chosen to be analogous to the operators of regular expressions. In this way, they easily allow to switch from a regular expression pattern to the combinator implementation. They fully respect the hierarchical structure defined by the QREs thanks to a typing system.

The goal is to be the most expressive with the least number of combinators [11]. Despite the fact that this list has grown a bit over the years, it remained quite small.

The following combinator specifications are written on the basis of those present in two articles, ‘*An Introduction to the StreamQRE Language*’ [12] and ‘*StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data*’ [37]. These articles are about the StreamQRE language, which is an implementation of the QREs in Java. The specifications still remain independent of this particular implementation.

### 3.7.1 Atomic

The *atom* combinator evaluates each symbol of an input stream depending on a predicate. According to the following notation, only the symbols which satisfy the predicate  $\phi$  are evaluated by the operation *op*. Otherwise, the QRE is undefined. This capability refers to the addition of predicates in the QRE domains by the symbolic regular expressions.

QRE	Rate
$h : QRE\langle D, C \rangle = atom(\phi, op)$	$rate(h) = \phi$
$\phi : D \rightarrow Bool$	
$op : D \rightarrow C$	

This combinator is the basis of many others. It’s the only one that doesn’t have any nested QRE as parameter. For this reason, it often goes along with an always-true predicate,  $atom(true, x \rightarrow x)$ . If this is the case, it evaluates all the elements of the stream [12]. That QRE is in opposition with the everywhere-undefined function [11] rejecting every symbol of a stream,  $\perp : D \rightarrow \perp$ .

**Example 3.7.1.** The next queries simulate the processing of a vehicle stream, where each item contains both the type of vehicle and its speed. Depending on one of the three filters chosen, the stream will be filtered to keep only *motorcycle(m)*, *car(c)* or *truck(t)* items. The output values are the speed values of each vehicle.

$$\begin{aligned}
\phi_1 &: x.type = m \\
\phi_2 &: x.type = c \\
\phi_3 &: x.type = t \\
isSpeed &= atom(\phi, x \rightarrow x.speed)
\end{aligned}$$

If the query was built with the predicate  $\phi_2$ , it would only emit the speed of cars. The query can be applied to the following stream, where the syntax of each tuple represent *(type, speed)*,

$$w = (t, 87), (c, 100), (m, 105)$$

The respective results for each of the values of  $w$  would be:  $\perp, 100, \perp$ .

### 3.7.2 Iteration

The *iter* combinator is the analog of Kleene-iteration operation of regular expression. Integrated in the following QRE  $h$ , it iterates over a nested QRE  $f$  and aggregates the results with an operation *op*, starting at the value *init*. The first value received by the QRE is combined with *init*, then stored. Next, for each new received value, the stored one is replaced by its combination with the new value using the *op* function.

QRE	Rate
$h : QRE\langle D, C \rangle = iter(init, f, op)$	$rate(h) = rate(f)^*$
$init : C$	
$f : QRE\langle D, A \rangle$	
$op : C \times A \rightarrow C$	

The combinator iterates over consecutive non-overlapping subsequences [37]. It implies that the nested QRE domain must be unambiguously iterable. The rate of  $h$  depends directly on the rate of  $f$ .

Initially, in the first draft on the quantitative regular expressions, two nested QREs were used by the combinator, and  $init$  was the second one [12]. The initial function was marking the start of the iteration, and had to be unambiguously concatenable with the domain of the inner QRE,  $f$ . At this time, the rate of  $h$  was:  $rate(init) \cdot rate(f)^*$ . Since the last work [37], the specifications have changed and the value  $init$  is now a constant value.

**Example 3.7.2.** The following query count the number of vehicles over a stream. The *iter* implementation computes successive additions starting from an initial value of 0. Each element of the stream is accepted by the predicate of the *atom* query, which is always *true*, then the value 1 is emitted and received by the *iter* operation.

$$\begin{aligned} init &= 0 \\ isVehicle &= atom(true, x \rightarrow 1) \\ isVehicleCount &= iter(init, f, (x, y) \rightarrow x + y) \end{aligned}$$

For example, let's consider an input stream of four vehicles, such as,

$$w = (t, 87), (c, 100), (m, 105), (m, 106)$$

Then, the result would be 4. While backtracking on the use of a cost grammar, the corresponding term would be:  $t = 0 + (1 + (1 + (1 + (1))))$ .

Be careful that, in this case, the *rate* of *atom* and *iter* QREs is strictly identical. If ever the predicate of *atom* changes to count only motorcycles, the query would change in,

$$\begin{aligned} init &= 0 \\ isMotorcycle &= atom(x \rightarrow x.type = m, x \rightarrow 1) \\ isMotorcycleCount &= iter(init, f, (x, y) \rightarrow x + y) \end{aligned}$$

In this case, on the same  $w$  stream as before, the query would always return an undefined value. While the *iter* combinator would wait for an input for each streaming item received, it would only receive them for motorcycle items. The cost term would be:  $t = 0 + (\perp + (\perp + (1 + (1))))$ , then the result could only be  $\perp$ .

With non-commutative operations, the construction of the term according to the order of arrival of the elements will influence the results. If the developer did not have the possibility to choose the order of operation parameters, this would be problematic even for a simple division. Therefore, in each operation *op*, the switch between the parameters, here  $x$  and  $y$ , allows to work around the problem. The operation  $(x, y) \rightarrow x/y$ , and the reversed one  $(x, y) \rightarrow y/x$ , are both valid operations.

In certain usage of the combinators [11], the possibility of reversing the order of operations is mentioned under the name of reverse combinator. This functionality coincides with the addition of holes in the cost grammar, as shown in Section 3.5.3. This additional feature has allowed the *iter* combinator to have the same capabilities than what have a cost register automata for iteration.

### 3.7.3 Concatenation

The *split* combinator, initially called sum operator [11], is the analog of concatenation operation of regular expression. The input stream is split in two consecutive non-overlapping parts. As shown in the specifications below, the function  $f$  is applied to the first part and  $g$  to the second. The operation *op* combines these two intermediate results.

QRE	Rate
$h : QRE\langle D, C \rangle = split(f, g, op)$	$rate(h) = rate(f) \cdot rate(g)$
$f : QRE\langle D, A \rangle$	
$g : QRE\langle D, B \rangle$	
$op : A \times B \rightarrow C$	



The input stream must be uniquely decomposable in two parts, which means that the rates of  $f$  and  $g$  must be unambiguously concatenable. In this way, the QRE  $f$  matches with the prefix of the stream and  $g$  with the suffix.

**Example 3.7.3.** In this example, with the addition of the split operator, the computation of the maximum density of vehicles by hour is now possible to process. This was not the case with only the *iter* and *atom* combinators. In the previous sections about the general structure of QREs, such queries had already been shown, among others in the Examples 3.6.2 to 3.6.6. The semantics used here will be the same as in these examples. Each element of the stream have a type and, eventually, a speed.

The first two combinators, *isEndHour* and *isVehicle*, execute a filtering on the items to separate those which mark the end of an hour from the items containing vehicle information.

The first iteration combinator, *isVehicleCount*, count the number of consecutive vehicle items. Considering only the type of each item, this combinator matching regular expression is  $(m|c|t)^*$ .

The first split combinator, *isVehicleCountByHour*, tries to divide its input term in two parts to emit the total number of vehicles during an hour. The left part counts the number of vehicles and the right part recognizes the end hour symbol. It's corresponding regular expression is  $(m|c|t)^* \cdot endHour$ .

The combinator *isMaxVehicleCountInLastHours* aggregates the total number of vehicles for each hour, while keeping only the maximum. The whole regular expression, considering the nested combinators, is now  $((m|c|t)^* \cdot endHour)^*$ .

The last combinator, *isMaxVehicleCount*, returns the maximum number of vehicle during every hour, considering all past hours but also the current one. The final regular expression embracing the whole algorithm is then  $((m|c|t)^* \cdot endHour)^* \cdot (m|c|t)^*$ .

*init* = 0

*isEndHour* = *atom*( $x \rightarrow x.type = endHour, x \rightarrow x$ )

*isVehicle* = *atom*( $x \rightarrow x.type = m \parallel x.type = c \parallel x.type = t, x \rightarrow 1$ )

*isVehicleCount* = *iter*(*init*, *isVehicle*,  $(x, y) \rightarrow x + y$ )

*isVehicleCountByHour* = *split*(*isVehicleCount*, *isEndHour*,  $(x, y) \rightarrow x$ )

*isMaxVehicleCountInLastHours* = *iter*(*init*, *isVehicleCountByHour*,  $(x, y) \rightarrow \max(x, y)$ )

*isMaxVehicleCount* = *split*(*isMaxVehicleCountInLastHours*, *isVehicleCount*,  $(x, y) \rightarrow \max(x, y)$ )

Intuitively, in the first approach, iterating over the *isVehicleCountByHour* query might seem sufficient given that it emits the desired result at the end of each hour. This would avoid using the last split combinator which does not seem necessary. However, the query must cover the entire stream. For example, let's consider the following term<sup>3</sup>,

$$w = \underbrace{c, m, m, m, c, t, c, t, m, endHour}_{(m|c|t)^* \cdot endHour}, \underbrace{c, t, m}_{?}.$$

While the prefix of the stream is recognized, the suffix cannot remain undetermined until the next end hour token. The last split, done by *isMaxVehicleCount*, is consequently a necessity. If this valid query was used to process the string  $w$  above, the result would be 9.

### 3.7.4 Conditional

The *or* combinator, also called global choice combinator, applies either one function or another,  $f$  or  $g$ , to the input stream depending on which one is defined. It doesn't modify the results produced by the evaluated nested QRE. It's the analog of union operation of regular expression.

**QRE**

$h : QRE\langle D, C \rangle = or(f, g)$

$f : QRE\langle D, C \rangle$

$g : QRE\langle D, C \rangle$

**Rate**

$rate(h) = rate(f) \sqcup rate(g)$

<sup>3</sup>The stream is only represented with the type values for comprehension purpose

The domain of both nested QREs must be disjoint for this combinator to be used. The rate of the combinator is the union of their domain.

**Example 3.7.4.** Considering an input stream, inside a traffic monitoring system, emitting for only information the type of each vehicle indicating its position, the *or* combinator is a solution for the computation of the cumulative sum of all the vehicles of a certain type. The definition of the first three queries, *isMotorcycle*, *isCar* and *isTruck*, filter each type of vehicle. In this case, let's imagine that only cars should be counted in the cumulative sum, and therefore produce the value 1, while motorcycles and trucks items produce the value 0.

The two *or* combinators manage to choose the right combinator for each iteration depending on the item type, then the sum is computed by the *iter* combinator. The regular expression corresponding to the whole query is  $(m|c|t)^*$ .

$$\begin{aligned} isMotorcycle &= atom(x = c, x \rightarrow 0) \\ isCar &= atom(x = c, x \rightarrow 1) \\ isTruck &= atom(x = c, x \rightarrow 0) \\ isMotorcycleOrCar &= or(isMotorcycle, isCar) \\ isMotorcycleOrCarOrTruck &= or(isMotorcycleOrCar, isTruck) \\ isCarCount &= iter(0, isMotorcycleOrCarOrTruck, (x, y) \rightarrow x + y) \end{aligned}$$

A potential stream accepted by the algorithm would be the following,

$$w = t, c, m, m, c, t, c, t, t, m$$

On this word, the total sum of the cars would be 3.

### 3.7.5 Combination

The combinator *combine* continually merges the results of two, or more, nested QREs. According to its definition, the functions  $f$  and  $g$  are processed in parallel and their results are combined using the operation  $op$ .

QRE	Rate
$h : QRE\langle D, C \rangle = combine(f, g, op)$	$rate(h) = rate(f) = rate(g)$
$f : QRE\langle D, A \rangle$	
$op : A \times B \rightarrow C$	

Both nested functions must emit, at the same time, a result on the processing of the input sequence for the combination to occur. Generally, they are fed by the same input stream, but that is not mandatory, they can have different input sources. Either way, the rates of  $f$  and  $g$  must be equivalent. Accordingly, the rate of  $h$  is the same as each of them.

**Example 3.7.5.** In this example, the operations compute the average speed from a vehicle stream. Because the calculation of the average speed needs to maintain the count of the vehicles and their cumulative speed in parallel, the *combine* combinator is the perfect feature to choose.

In this case, the stream is considered to contain vehicle items only, which is why the predicate of the first and the third combinators, *atom*, are always true.

The *isVehicleCount* and *isVehicleSpeedSum* queries are computed in parallel. For each new vehicle received, *combine* query transfers the item to both inner queries at the same time, get the results, and combine them immediately with the division operation.

$$\begin{aligned} init &= 0 \\ isVehicle &= atom(x \rightarrow true, x \rightarrow 1) \\ isVehicleCount &= iter(init, isVehicle, (x, y) \rightarrow x + y) \\ isVehicleSpeed &= atom(x \rightarrow true, x \rightarrow x.speed) \\ isVehicleSpeedSum &= iter(init, isVehicleSpeed, (x, y) \rightarrow x + y) \\ isVehicleAverageSpeed &= combine(isVehicleSpeedSum, isVehicleSpeed, (x, y) \rightarrow x/y) \end{aligned}$$

As example, the following stream simulates a possible string for the algorithm to process,

$$w = (t, 80), (c, 100), (108, m), (m, 105), (c, 98).$$

The queries *isVehicleCount* and *isVehicleSpeedSum* are both process under the rate of the regular expression  $(m|c|t)^*$ . The final result emitted by the *isVehicleAverageSpeed* query is 98.2, computed by the term  $(80 + 100 + 108 + 105 + 98)/(1 + 1 + 1 + 1 + 1)$ .

The combinator *combine* can be generalized to take as parameters an indefinite number of inner queries, as long as these queries always keep the same rate.

### 3.7.6 Apply

A similar combinator to *combine* is the application. Also called substitution [10], it's a combination with only one argument. The combinator transforms each element of the stream one by one.

#### QRE

$$h : QRE\langle D, C \rangle = apply(f, op)$$

$$f : QRE\langle D, A \rangle$$

$$op : A \rightarrow C$$

#### Rate

$$rate(h) = rate(f)$$

**Example 3.7.6.** In the context of traffic monitoring, it can be used to calculate a difference between the current speed of each vehicle and a reference speed. In this example, a stream containing the vehicles speed is filtered to return the difference of speed between 80km/h and each current vehicle speed. Not all the difference are emitted as a result but only the first vehicle speed which exceed it.

$$init = 0$$

$$isVehicleSpeedLeq80 = atom(x \rightarrow x \leq 80, x \rightarrow x.speed)$$

$$isVehicleSpeedIterLeq80 = iter(init, isVehicleSpeedLeq80, (x, y) \rightarrow 0)$$

$$isVehicleSpeedGreater80 = atom(x \rightarrow x > 80, x \rightarrow x.speed)$$

$$isVehicleSpeedDiff80 = apply(isVehicleSpeedGreater80, x \rightarrow x - 80)$$

$$isFirstVehicleSpeedMore80 = split(isVehicleSpeedIterLeq80, isVehicleSpeedDiff80, (x, y) \rightarrow y)$$

With the query *isVehicleSpeedIterLeq80*, the algorithm iterates on each vehicle whose speed is not greater than 80km/h. Then, with *isFirstVehicleSpeedMore80*, the algorithm splits on the first value which is strictly greater than 80, and the *apply* combinator returns the difference. The only result which will be produced is the first matching value. The corresponding regular expression  $(x \leq 80)^* \cdot (x > 80)$ , will never match again after, and the next results will always be undefined.

In practice, considering the following stream, this state of the stream, as it is shown, is in the only state where the query will ever recognize it,

$$w = (c, 68), (c, 53), (c, 70), (m, 85)$$

After processing of the element  $(m, 85)$ , the result will be 5. If a new element arrives after that, the regular pattern  $(x \leq 80)^*$  will not recognize the prefix of the stream anymore, as it will contain the element  $(m, 85)$ , for which the speed is over 80. This element cannot be part of the suffix of the query either, because the regular expression  $(x > 80)$  only accept the last element of the stream.

Actually, the *apply* combinator is optional in this algorithm. The *atom* combinator syntax includes sort of a built-in *apply* combinator. To achieve same algorithm, giving the same result, without *apply*, the queries *isVehicleSpeedGreater80* and *isFirstVehicleSpeedMore80* have just to be changed to those,

$$isVehicleSpeedGreater80 = atom(x \rightarrow x > 80, x \rightarrow x.speed - 80)$$

$$isFirstVehicleSpeedMore80 = split(isVehicleSpeedIterLeq80, isVehicleSpeedDiff80, (x, y) \rightarrow y)$$

This whole example can also serve as a comparison with ReactiveX. The processing performed here is almost identical to the one presented in Section 2.8. The *apply* combinator is the analog of the *map* operator of RxJS.

### 3.7.7 Window

Sometimes, the *split* combinator syntax can be hard to manage. In the case where a number of predefined successive concatenations on the same type of element with the same operation must chain, the syntax can become very verbose. For example, let's imagine that the goal of an algorithm is to detect a slowdown in the current vehicles speed. The calculation of the average speed must then not be carried out on all the vehicles, but only on the last few to see the difference compared to the previous ones.

Taking into account only the last five vehicles, here is the query that should be used, on a stream containing only vehicles elements, if the only operator available for concatenation is *split*,

```

init = 0
isVehicleSpeed = atom(x → true, x → x.speed)
isVehicleSpeedSum = iter(init, isVehicleSpeed, (x, y) → x + y)
isVehicleSpeedSum1 = split(isVehicleSpeedSum, isVehicleSpeed, (x, y) → y)
isVehicleSpeedSum2 = split(isVehicleSpeedSum1, isVehicleSpeed, (x, y) → x + y)
isVehicleSpeedSum3 = split(isVehicleSpeedSum2, isVehicleSpeed, (x, y) → x + y)
isVehicleSpeedSum4 = split(isVehicleSpeedSum3, isVehicleSpeed, (x, y) → x + y)
isVehicleAverageSpeed = split(isVehicleSpeedSum4, isVehicleSpeed, (x, y) → (x + y)/5)

```

The number of combinators is already starting to be quite large when only 5 vehicles are observed. If the number increased to 20, 30 or 100, the code would become unreadable which is contrary to the intended purpose of the QREs. Moreover, the syntax of the *split* combinators is a strict repetition of the same operation. To make the query easier to write it would be necessary to factorize them.

For this reason, the *window* combinator was invented. It recognizes a predefined number of successive concatenations, defined by the *size* parameter, on a stream, and aggregates them together using an operation *op*. Its rate depends directly on the *size* parameter.

QRE	Rate
$size : B$	$rate(h) = rate(f) * size$
$h : QRE\langle D, C \rangle = window(size, f, op)$	
$f : QRE\langle D, A \rangle$	
$op : A \times A \rightarrow C$	

**Example 3.7.7.** Here is the averaging query on the last five vehicle speeds, written once again but this time with the use of the *window* combinator,

```

init = 0
isVehicleSpeed = atom(x → true, x → x.speed)
isVehicleSpeedSum = iter(init, isVehicleSpeed, (x, y) → x + y)
isVehicleSpeedSumForLast5 = window(5, isVehicleSpeed, (x, y) → x + y)
isVehicleAverageSpeed = split(isVehicleSpeedSum, isVehicleSpeedSumForLast5, (x, y) → y/5).

```

Both queries output the same results, but in term of expressivity, the last one is much more readable. Their corresponding regular expression is  $x^* \cdot x \cdot x \cdot x \cdot x \cdot x$ .

The following stream well represents a case of application of the query in which the average speed of the vehicles was quite high, and has just fallen sharply.

$$w = (c, 138), (c, 120), (t, 110), (m, 40), (c, 40), (c, 40), (t, 40), (m, 40)$$

The results after processing each element will be  $\perp, \perp, \perp, \perp, 89.6, 70, 54, 40$ . Results are undefined until there are at least 5 items in the stream to match the regular expression.

### 3.7.8 Streaming composition

The streaming composition combinator is noted  $\gg$ . This operation is the analog of the function composition. It feeds a query with the output of an other. In the function  $f \gg g$ , the output of a QRE

$f$  is supplied as input stream for the QRE  $g$ . A correlation between the rates of the two queries is not necessary. One restriction is that the streaming composition can only be used at top level of a QRE.

**QRE.**

$$f \gg g : QRE\langle D, C \rangle = g \circ f$$

$$f : QRE\langle D, A \rangle$$

$$g : QRE\langle A, C \rangle$$

**Example 3.7.8.** In the following stream, vehicles of all types are mixed,

$$w = (m, 120), (c, 117), (t, 110), (t, 90), (c, 100), (m, 100), (t, 96), (c, 88)$$

If the need is to calculate the average speed of the trucks, it can be easier to use two QREs than to try to isolate the trucks, and calculate their average speed in one single query. In the following example, the first query filters the trucks, and emits a result when the last item is one of them,

$$init = 0$$

$$isTruck = atom(x \rightarrow x.type = t, x \rightarrow x)$$

$$isVehicle = atom(x \rightarrow true = t, x \rightarrow 0)$$

$$isVehicles = iter(init, isVehicle, (x, y) \rightarrow 0)$$

$$isTruckAsLastVehicle = split(isVehicles, isTruck, (x, y) \rightarrow y).$$

The second query compute the average of the speed data from its input stream. The query no longer needs to worry about the type of vehicle,

$$init = 0$$

$$isVehicle = atom(x \rightarrow true, x \rightarrow 1)$$

$$isVehicleCount = iter(init, isVehicle, (x, y) \rightarrow x + y)$$

$$isVehicleSpeed = atom(x \rightarrow true, x \rightarrow x.speed)$$

$$isVehicleSpeedSum = iter(init, isVehicleSpeed, (x, y) \rightarrow x + y)$$

$$isVehicleAverageSpeed = combine(isVehicleSpeedSum, isVehicleSpeed, (x, y) \rightarrow x/y).$$

Using the combinator  $\gg$ , the output stream of the first query containing only trucks can be transferred to the second to calculate their average speed,

$$isTruckAsLastVehicle \gg isVehicleAverageSpeed$$

The regular expression corresponding to this whole query is simply  $t^*$ . Processing the example stream, the final result term will be  $(110 + 90 + 96)/3$ , which gives the value 98.66.

### 3.7.9 Empty sequence

The *eps* combinator produces a constant value if the input stream is empty.

**QRE**

$$h : QRE\langle D, C \rangle = eps(c)$$

**Rate**

$$rate(h) = \epsilon$$

This combinator was only mentioned in the first work about StreamQRE [12]. Associated with the previous version of the *iter* combinator, which is described in Section 3.7.2, it allowed to initialize a new iteration with a starting value.

In this previous version, the *iter* combinator was built as a split expression between the initial value and the effective iteration.

It has been removed from the actual version [37], and an initial value was added in the syntax of the *iter* combinator without the need of an inner QRE.

**Example 3.7.9.** Using the syntax of the old *iter* combinator, a query to count the number of vehicles in a stream, or any other items, could have been written as follows,

$$\begin{aligned} init &= 0 \\ isVehicle &= atom(true, x \rightarrow 1) \\ isVehicleCount &= iter(eps(init), f, (x, y). \rightarrow x + y) \end{aligned}$$

The only effect that the removal of the *eps* combinator have produced is syntactic, because the query *isVehicleCount* is now written,

$$isVehicleCount = iter(init, f, (x, y). \rightarrow x + y)$$

This concludes the series of regular combinator types available.

### 3.8 Conclusion

This chapter retraces the conception of the quantitative regular expressions. They are intended to create high-level languages capable of processing a stream of data, giving to the users enough expressivity so they don't need to dive in implementation details for combining items. Their strength lies in the fact that they simultaneously guarantee great and stable performances, as well as low memory footprint.

The QREs rely on formal language theory to be able to apply functions on a stream with limited computational power. Accordingly, the stream is seen as a string recognized by a formal language which can be parsed by an automaton. After realizing that no actual automaton, or existing language, guarantees both performance and expressiveness for the evaluation of a string as a quantitative property, a new class of language is designed. Each language in this new class defines grammar rules, called cost grammar, allowing to insert operations between the elements of the stream. Each well-formed word is called a cost term, the cost naming referring to the quantitative property produced by its evaluation. This class of language also constrains the domain of the stream to respect regular language properties. In this way, the stream can be parsed and evaluated by a deterministic finite state machine, which guarantees good performances. To compute the quantitative property throughout the compilation, the finite automata is upgraded and comes with a finite number of "write-only" registers, giving it its name of cost register automata. These registers store the intermediate values of the calculations. Their finite number and their restricted access rules ensure minimal memory consumption.

The set of properties for this new class of language being complete, it would suffice for each desired quantitative property of a stream to create a new language, with its grammar, and to implement it with a cost register automata. However, performing these tasks from scratch is quite difficult. Therefore, a new language with a predefined syntax forms a layer of abstraction to create the desired query languages more easily. This syntax, called quantitative regular expressions, starts from a general structure matching operations to the regular expression patterns observed in the stream. This structure is declined in predefined operators, named regular combinators, analogous to the operations of regular languages. These combinators can be nested together to create a combinators tree, constituting the complete algorithm to compute a quantitative property. In addition, they ensure the conservation of the properties of regular languages and their evaluation by a cost register automata.

The quantitative regular expression language having been defined, it can now be implemented and translated in other abstractions.

# Chapter 4

## Existing incarnations

### 4.1 Introduction

Now that all the specifications of QREs have been defined, it remains to implement them in general-purpose languages to make them available for the conception of streaming processing algorithms on real use cases. Based on the specifications, three implementations have already been written in the past and each one has its specificities. All three are incarnations of quantitative regular expression features translated into a programming language. The first one, DReX, is only a partial implementation based on a first version of the QREs which was limited to the processing of strings only. Even if this implementation is no longer intended to be used, it is still important to review its characteristics to see the evolution compared to other implementations. The second implementation, StreamQRE, is the most well-known. This implementation was written in Java, was built over several years, and has been described in various articles. It fully complies with all the specifications of the QREs in the way they were defined in the previous chapter. The last one, NetQRE, is probably the least known of all, only one article is entirely devoted to it. The implementation is written in C++. Specialized in the field of network monitoring, it also brings its set of specific characteristics different from the others.

In this chapter, an analysis of all of them makes it possible to realize the biases, differences, and common features that have been taken by one or the other implementation. At the same time, this chapter will show the way in which a new implementation of the QREs can be developed, and will finally expose a new implementation for the initial tunnel traffic monitoring problem.

### 4.2 DReX

DReX is the precursor of the QREs. Known as a Declarative Language for Efficiently Evaluating Regular String Transformations, it was introduced in a 2015 article [8]. Having been developed in the early stages of the QREs, it does not fully comply with the specifications but remains interesting because some of these features were taken up later in other implementations. Unfortunately, no trace of the source code of this implementation has been found. Despite this constraint, which has limited its analysis, the articles explaining the DReX principles were complete enough to have a general understanding.

The main difference with QREs lies in the input stream domain for which the language creates queries, and the cost domain for which the created algorithms produce values. In standard QREs, the nature of the data received as an input stream can be of any type, as long as the streaming data conforms to the constraints of a regular language. Normally, when declaring the QRE, the input domain and the cost type can be configured in each combinator. For DReX expressions, this is not the case. The input domain and the cost type are fixed to a single domain for all combinators and for all queries.

Originally, the input stream and the produced values were forced to be of type string. The language was made to build expressions that map input strings to monoids. The cost terms were always built and describing a combination of string data. The evaluations of these expressions were string-to-string transformations and the types of string operations that the language was able to perform were insertion, deletion, substring swap, and reversal.

Subsequently, with the development of studies on DReX language capabilities, the specifications evolved to allow the creation of cost terms and the production of resulting values on other domains. Despite this evolution, the obligation to keep the same input domain and cost domain for all the combinators of the same query has always remained. If the input stream consists of boolean data then the quantitative property produced by the created algorithms will also be of boolean type.

Another difference of the DReX expressions is the syntax to specify their operations. The expressions are limited to the use of specific operation where each predicate is specified directly inside the operation.

**Example 4.2.1.** For example, in DReX, an iteration over each element of an integer stream combined with an addition operation could only be written with a specific operation *add*, containing the predicate. In the standard QREs, the predicates are specified within the *atom* combinator. They are separated from the operation and can be applied to the other combinators by taking advantage of their recursive structure. Concretely, the queries in both languages have a very different syntax but produce the same results. They are written as follows,

- DReX:  $iter(add(x \rightarrow true))$
- QRE:  $iter(0, atom(x \rightarrow true, x \rightarrow x), (x, y) \rightarrow x + y)$

In addition, while the operations in QREs are fully configurable, in DReX, each combinator is limited to an unique operation, which creates the need to build specific operations for each use case.

**Example 4.2.2.** For example, let's consider the subset of a stream composed of two integers that an algorithm should combine,

$$w = 4, 5$$

In QREs, during the process of building a query to recognize the concatenation of these integers, an operation can be associated to first add them, and then multiply their value by 10. The operation can be customized and set as parameter for the split combinator. Since the syntax of DReX is slightly different, a simple addition operation could be written  $add(x \rightarrow true)$ , but an addition followed by a multiplication would need to implement another specific operation. The two following operations show the difference in implementation. The QRE operation is easily customizable with a lambda compared to the DReX operation. In the QRE operation, the value 10 is a customizable parameter, while in the DReX operation, a specific operator *addAndMultBy10* needs to be predefined in the language implementation itself to be used. This operator is not customizable by the user. Those two operation can be used as a parameter for a split combinator,

- DReX op.:  $addAndMultBy10(x \rightarrow true)$
- QRE op.:  $(x, y) \rightarrow (x + y) * 10$

Beyond these differences, the language is similar to all other aspects of the QREs. The same types of combinators as for the QREs are applicable. For example, the combinators corresponding to Kleene-star iteration and to the concatenation, respectively *iter* and *split*, are usable to implement DReX expressions. In term of expressivity, the DReX language, considering the chosen domain as the string domain, ensured that its queries can express all of the regular string-to-string transformations. A DReX query will recognize the streaming patterns exactly the same way as in the QREs. In term of performance, the two languages cover the same class of regular functions, guaranteeing efficient evaluation.

**Example 4.2.3.** Another example in the traffic monitoring field can show the limitations of DReX. Let's consider a traffic stream as a sequence of words  $w \in \{v\}$ , where  $v$  are vehicles. The query counting the number of vehicles is not computable since the input domain,  $\{v\}$  and the cost domain,  $\mathbb{Z}$ , will be different, but if the traffic stream is a sequence of words  $w \in \mathbb{Z}$ , where each integer is a vehicle, the query is possible to specify with DReX.

Here is the queries, in DReX and QRE languages, for computing the aggregated count on a stream, where the QRE expression is valid and possible to evaluate, while the DReX is not,

- DReX:  $countInvalid = iter(add(x \rightarrow x == v))$
- QRE:  $countValid = iter(0, atom(x \rightarrow x = v, x \rightarrow 1), (x, y) \rightarrow x + y)$

The details concerning the implementations of the operations in the DReX language will not be presented here because the syntax is already sufficient to realize the differences compared to QREs. After the



conception of DReX, the QREs have generalized the DReX combinators by adding more flexibility for their operators and removing the unique domain constraint.

That language was necessary to consider because of the common points it has with the QREs. A part of the studies made on the time and memory complexity, concerning the recognition of a regular language by the DReX combinators within a stream, can be applied to the QREs. These studies have been taken into account for writing this thesis [8].

## 4.3 StreamQRE

The most prominent and studied implementation of QREs is StreamQRE. This is an implementation in Java whose compilation algorithm uses stacks to maintain the state of the flow between the arrival of elements. Within the framework of this implementation, an important work was carried out on the management of parallel computations, which are necessary due to the split operator. As it will be explained, stack-based operations have proven to be suitable for this task. The information necessary to download the StreamQRE Java library is available in Appendix B.1.

Unlike other implementations, the goal of StreamQRE is to adapt to all kinds of situations. It was built in a generic way, while perfectly respecting the specifications of the QREs. Many performance and memory testing have been carried out in a medical context, in particular to monitor the flow of patients' heartbeats, but many other use cases have been tested too. The testing results have demonstrated good performances without having to reduce the expressiveness of the language.

Using Java syntax, the implementation of the hierarchical structure of the combinators take advantage of the object-oriented paradigm. Each combinator is objectified into an object instance, and takes other combinators as parameters. At runtime, the different combinators can thus easily communicate together.

### 4.3.1 Language

One goal of the language being its generic nature, the conception was very focused on its ease of use to adapt to all kinds of situation. Therefore, the syntax, written in Java, is very similar to the abstract syntax of QREs. This makes it easier to reason at high level on the logic of the algorithms.

The major difference with the specification syntax is the use of generic types. By parameterizing QRE objects, the input and the cost domain of the combinator is defined. In Java, the generic types are written between angle brackets.

**Example 4.3.1.** Here is an example of a query in Java syntax, compared to the specification syntax. This query simply count the elements of the stream. It could be used for aggregating the number of vehicles observed on a highway. In Java, the input and the cost domain are defined with the Double Java type to use 64-bit floating point numbers.

The query uses two combinators. The first combinator, `item`, recognizes each element of the stream because its boolean filtering predicate, `x -> true`, will always accept any item. For each observed vehicle, the emitted result is the value 1. The second combinator, `sum`, successively adds all the values received from the combinator `item`, with the function `Double::sum`, starting from an initial value of 0. The cumulative sum is returned as is by the function `x -> x`.

Java:

</>
**Listing 7:** QREs, as Java function, to count each element of the stream
</>

```

1  QRe<Double, Double> item = new QReAtomic<>(x -> true, x -> 1);
2  QRe<Double, Double> sum = new QReIter<>(item, 0.0, Double::sum, x -> x);

```

Agnostic specifications:

$$\begin{aligned}
 item &= atom(x \rightarrow true, x \rightarrow 1) \\
 sum &= iter(0, item, (x, y) \rightarrow x + y)
 \end{aligned}$$

The high similarities between the two syntaxes are easily noticeable. This confirms the ease of moving from the design of an abstract algorithm to a concrete implementation.

Beyond ease of design, performance plays an important role in algorithm creation. The capacity of easily doing a performance analysis on a QRE makes it possible to reason on the processing time, as well as on the impact of certain operators on the global algorithm, and even on the energy consumption over time.

As StreamQRE has been experimented a lot in the medical field, the capacity of realizing the impact in terms of energy consumption of the design choices for the algorithms has been decisive. In the creation process, the engineers are led to adapt the algorithm as quickly as possible if they realize that it consumes too much energy. Most patients who have to replace implants are elderly and an increased battery life is a huge benefit for them.

In some medical documentation about streaming processing [1], the authors explain the correlation between the level of abstraction of an algorithm and the difficulty of measuring its power. The higher the level of abstraction of the algorithm is, the easier it is to analyze the cost, but the less precise the estimated cost is. In the context of StreamQRE, despite the theoretical bases ensuring strict complexity bounds, the implementation is built as a high-level abstraction in a high level language. This intentional choice favors having an estimate early enough in the design process, but it must be admitted that it will not be as precise as it could be in lower level languages.

One might think that relying on the theoretical bases of the QRE language is enough in all kind of situation. However, in critical cases, such as medical monitoring, it will often be needed to do additional performance testing, especially because each query can be parameterized by customizable operations. It is difficult to know with precision their impact without having already executed them. Hopefully, some current methods are able to measure performance by running the processing algorithms on a typical load, and testing them with different hardware components. The use of QREs, while relying on their complexity bounds, is not intended to replace them, but to provide additional information, including an order of magnitude of performance earlier in the design process.

Regarding performance, StreamQRE relies entirely on QREs and regular function properties, as they were specified in previous sections. The way the compilation process benefit from these properties is defined in the next section.

### 4.3.2 Compilation

This section marks the link between StreamQRE implementation and the guarantee in terms of performances of the QREs. It is crucial to understand this section of the thesis to realize how the operating mode of a cost register automata has been transposed into a Java algorithm. This part of the work was conceived after analysing in details ‘*StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data*’ [37] and ‘*Regular programming for quantitative properties of data streams*’ [2].

Recall that the evaluator of a regular function receives one item at a time in a streaming fashion, and every time it evaluates the result of the function from left to right taking into account the entire stream. Then, if the regular expression expressed by the function is defined, it produces a valid result. If not, it produces an undefined result.

Usually, to assess the complexity of an algorithm on a finite string, the procedure consists to consider the entire string, making initial assumptions about its type, and then deducing complexity bounds according to its size. For example, to sort an array with a bubble sort, the worst time complexity will be  $O(n^2)$ , where  $n$  is the size of the array. One difference between a string whose length is fixed and a stream is that the total length of the stream cannot be predetermined. Therefore, to calculate the complexity of the compilation algorithm that evaluates the expressions, the stream length cannot be used as a representative measure. Instead, the algorithm consumption per data item in time and memory is a much more appropriate unit of measurement than the consumption over a whole stream.

Given that the purpose of a QRE is to calculate a quantitative property on the whole stream, it must still be taken into account in the complexity calculation. Therefore, the goal is to minimize the time and space necessary to evaluate the whole stream at the arrival of each new item. Since the evaluation of QREs, based on the SSTT model, should respect copyless and single use restriction, the complexity should be linear depending on the length of the stream. If the function  $f$  is a regular function, the evaluation  $[f](w)$  has a complexity of  $O(k * |w|)$  where  $k$  is a constant value and  $w$  is a stream of items. Thanks to this, the complexity will not grow up exponentially as the elements arrive. This topic was deeply analyzed in Section 3.5.3.

With that being said, it's assumed that the added complexity when processing a new element should not depend on its place in the stream but should be constant. The design of StreamQRE is no exception to this rule, each algorithm implemented in this language will respect the theoretical QRE definitions and will have fixed complexity limits both in terms of memory and performance.

If the  $k$  constant in the formula  $O(k*|w|)$  doesn't depend on the length of the stream, it turns out that it depends directly on the composition of the regular function  $f$ . For the conception of this implementation [10], the authors affirm that each data item is processed in polynomial time depending on the length of the function,  $|f|$ . By the length of the function, it's the combinators composing the function which are targeted. This means that the number of the combinators composing the regular function, and their nature, directly influence the time and memory complexity of the global algorithm.

In particular, the increase in complexity is due to *split* and *iter* combinators. In the context of this implementation [8], these are called stateful combinators in opposition to *else* or *combine* which are state-free combinators. These names are justified by the fact that stateful combinators must maintain a state in memory between the arrival of new items, while state-free don't. In other words, the *iter* combinator must keep the aggregated value in memory between its iterations, and *split* must keep the aggregated value resulting from the combination of its parameters. However, only keeping the last aggregated value is not always sufficient.

The next paragraphs explain the process that led to maintain a state in these combinators, and their correlation with the impact on the performance of regular functions.

In three articles which detail this problem [1, 8, 10], the explanations rely on a very theoretical approach. They employ the use of signals, sent to the combinators, to explain when the evaluation of a part of the stream should start or stop. In this thesis, the problem is approached from another point of view. The behaviour of the combinators is determined by analyzing the variations of the possible patterns expressed by the regular expressions.

The choice of a more practical approach is justified by the desire to carry out another implementation later in the document.

### Split

A first explanation focuses on the impact of *split* on the compilation. This explanation is introduced, starting from a naive solution to implement the combinator. A stream  $w$  is composed of the items  $a_0$  to  $a_j$  where  $j > 0$ . Considering that  $f$  and  $g$  are two nested combinators, this stream is split in two parts by a combinator  $split(f, g, op)$ . Let's suppose that, with the index  $i$  such that  $0 < i < j$ , the part  $a_0$  to  $a_i$  is recognized by the function  $f$  and the part  $a_i$  to  $a_j$  is recognized by the function  $g$ , as shown in Figure 4.1.

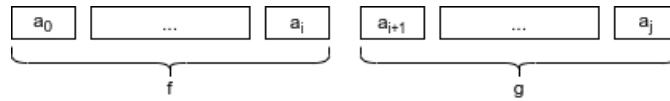


Figure 4.1: Theoretic split of a stream

Based on that definition, there could be  $j$  possible splittings of the stream as the variable  $i$  can take any possible value from 0 to  $j$ . This is illustrated in Figure 4.2.

In this case, the number of splitting possibilities would be  $O(|w|)$ , and would indefinitely increase with the arrival of new items in the stream.

This definition does not take into account the properties of the QREs language which were explained previously. In fact, for each split combinator, the domain recognized is defined by two symbolic regular expressions which are unambiguously concatenable. These two expressions define the domain of  $f$  and  $g$  functions. For each possible index  $i$ , if the two sub-stream domains from 0 to  $i$  and  $i$  to  $j$  are respectively recognized by these regular expressions, the separation at this index is the only one possible in the stream.

For example, if the split combinator recognizes the decomposition in Figure 4.3, then all the others decompositions aren't possible.

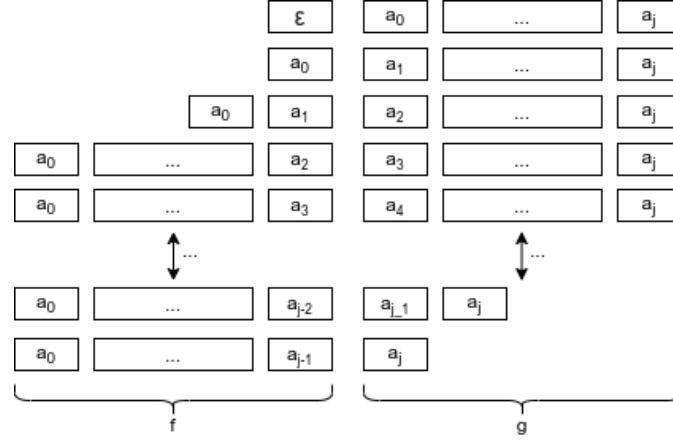


Figure 4.2: Concrete splits of a stream

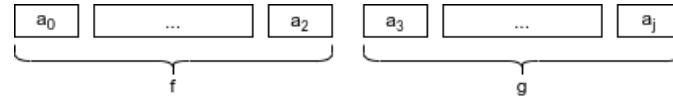


Figure 4.3: One only concrete split possible

However, the stream splitting performed above rely on the fact that only one split combinator is part of the QRE. The QRE domain would therefore be very limited. The regular expression, which defines that domain, could only be composed of two parts, for example  $x^* \cdot y^*$ , considering a domain defined on the alphabet  $\{x, y\}$ . Hopefully, as we have seen before, each combinator uses nested combinators which can themselves be split combinators. An example of a stream split in three part is shown on Figure 4.4.

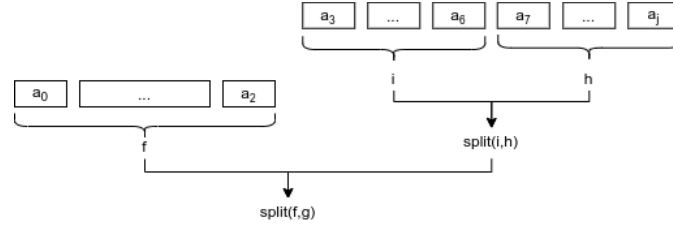


Figure 4.4: Multiple split combinators

As the concatenations present in the regular expression are multiplied, the number of split combinators increases and the number of splitting possibilities at the arrival of a new item too. The amount of possibilities for splitting the stream is therefore limited by the size of the regular expression representing the domain of the QRE. For example, in Figure 4.4, one only unambiguous splitting exists in the domain covered by the combinator  $split(f, g)$ , but one only other splitting also exists under the domain of the function  $g$  covered by the combinator  $split(i, h)$ . If more splits combinators was included, the number of splitting possibilities would have increased further.

From a first intuition, the process is simply to check the domain of which combinator each new item belongs. By testing them one after the other, the complexity seems to increase linearly according to the size of the corresponding regular expression. However, checking that each new element belongs to different parts of the expression independently of each other is not sufficient. The next paragraphs show why the belonging of a new item to a fixed domain, at his arrival, can't always be established.

First, let's imagine the case of a stream  $w$  whose  $j$  elements have already been read by the query and whose domain is recognized by the  $split(f, g)$  combinator on the basis of a separation at index  $i$ . Here is the several possibilities to test at the arrival of a new item,  $a_{j+1}$ . This initial situation is shown in Figure 4.5.

The first possibility, P1, is the recognition of the new element as being part of the domain of the function

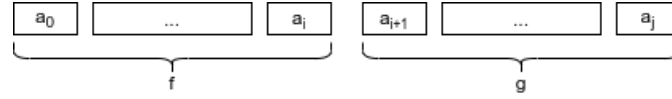
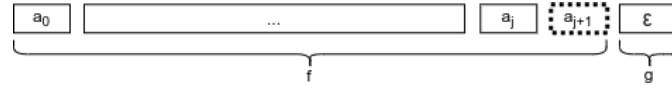


Figure 4.5: Initial situation

$g$ . In this case, the state of the stream splitting is updated as shown in Figure 4.6.

A second possibility, P2, would be that the function  $f$  recognizes the set of elements from 0 to  $j + 1$ , including the new element, and that the function  $g$  accepts the empty string. This possibility is shown in Figure 4.7.

A last possibility, P3, would be that the function  $f$  recognizes the set of elements from 0 to  $j + 1$ , including the new element, but considering that the function  $g$  doesn't accept the empty string. The result of the query would become undefined. This decomposition on the stream is shown in Figure 4.8.

Figure 4.6: P1 - The item is part of  $g$  domainFigure 4.7: P2 - The item is part of  $f$  domain,  $g$  accept the empty stringFigure 4.8: P3 - The item is part of  $f$  domain,  $g$  doesn't accept the empty string

The last two cases, P2 and P3, are the most impactful in terms of performance and complexity. Without having an immediate result of  $f$  and  $g$  concatenated, the QRE evaluator is forced to keep in memory the partial result of the function  $f$ . That result is kept in the expectation that, as the next elements ( $a_{j+2}$ ,  $a_{j+3}$ , etc.) arrive, at some point, the function  $g$  recognizes the other part of the stream (from  $a_{j+2}$  to  $a_{j+x}$ , where  $x > 2$ ).

If that happens, a pattern similar to the one explained in P1 would emerge.

Therefore, at each new item arrival, a partial term should possibly kept in memory by the QRE evaluator because of the split combinator.

The cases in which  $f$  does not accept the stream prefix are not mentioned, and they shouldn't be taken into account. In the case, where  $f$  doesn't recognize the set of elements from 0 to  $j + 1$ , it would be impossible for the split combinator to expect any result in the future.

**Example 4.3.2.** Let's build an example to demonstrate the necessity for the split combinator to keep a partial result in memory. In this use case, the goal will be to create an algorithm to learn more about the conditions in which each road accident occurs. The following QRE is built to calculate the speed of the last 10 vehicles observed before the last occurrence of an accident.

The implementation of the query is exposed in the following code snippet. The intended stream to process is composed of vehicle and accident tokens. Each accident token is recognized by the combinator `isAccidentToken`, and the speed value of each vehicle is recognized by the combinator `isVehicleSpeedToken`. This distinction is made by the comparison between the type of item received, retrieved by `getItemType` method, and a constant value, which is `ACCIDENT` or `VEHICLE`. The lambda function `x -> x.getSpeed()` returns the speed of each observed vehicle as a result. The `isVehicleSpeedOrAccidentToken` combinator switches for the right inner combinator, between `isAccidentToken` and `isVehicleSpeedToken`,

to match each received streaming item. The combinator `is10ItemsSpeedSum` uses the sliding window pattern to compute the aggregated sum of the speeds of 10 vehicles<sup>1</sup>. It calculates the cumulative sum of the speed values received from its inner combinator `isVehicleSpeedOrAccidentToken`. The function `Double::sum` performs this sum, starting with an initial value of 0. It returns the result as is via the function `x -> x`. When an accident occurs, the `is10ItemsAvgSpeedBeforeAccident` combinator emits the average speed of the 10 vehicles preceding the accident. The domain recognized by this combinator is the concatenation of `is10ItemsSpeedSum` with `isAccidentToken`. Since `is10ItemsSpeedSum` returns a cumulative speed sum, the function `(x,y) -> x/10D` just divides it by 10 to compute the average. The variable `y`, containing the result of the combinator `isAccidentToken`, has no interest to be used at all. The next combinator, `isVehicleSpeedSum`, computes the cumulative sum of the speeds for an indefinite number of vehicles. The sum is computed by the function `Double::sum`. The last combinator, which is named `is10ItemsAvgSpeedBeforeLastAccident`, only keeps the average speed of the 10 vehicles preceding the last accident. For this purpose, it separates the domain of `is10ItemsAvgSpeedBeforeAccident` from the domain of `isVehicleSpeedSum`. It keeps only the result of the variable `x` in the function `(x,y) -> x` and doesn't use the result from `isVehicleSpeedSum` combinator.

</> Listing 8: QREs, as Java function, to apply functions on regular subset of a stream </>

```

1 private static QReSplit<StreamingDto, Double, Double, Double>
  ↪ averageSpeedOfVehiclesBeforeLastAccident() {
2     var isAccidentToken = new QReAtomic<StreamingDto, Double>(x ->
  ↪     Objects.equals(x.getItemType(), ACCIDENT), x -> 0D);
3     var isVehicleSpeedToken = new QReAtomic<StreamingDto, Double>(x ->
  ↪     Objects.equals(x.getItemType(), VEHICLE), x -> x.getSpeed());
4     var isVehicleSpeedOrAccidentToken = new
  ↪     QReElse<>(isAccidentToken, isVehicleSpeedToken);
5     var is10ItemsSpeedSum = new QReWindow<>(isVehicleSpeedOrAccidentToken, 0D,
  ↪     Double::sum, x -> x, 10);
6     var is10ItemsAvgSpeedBeforeAccident = new QReSplit<>(is10ItemsSpeedSum,
  ↪     isAccidentToken, (x,y) -> x/10D);
7     var isVehicleSpeedSum = new QReIter<>(isVehicleSpeedToken, 0D, Double::sum, x ->
  ↪     x);
8     var is10ItemsAvgSpeedBeforeLastAccident = new
  ↪     QReSplit<>(is10ItemsAvgSpeedBeforeAccident, isVehicleSpeedSum, (x,y) -> x);
9     return is10ItemsAvgSpeedBeforeLastAccident;
10 }

```

Obviously, this is the split combinators `is10ItemsAvgSpeedBeforeAccident` and `is10ItemsAvgSpeedBeforeLastAccident` which are interesting to analyze in this case. Let's take a snapshot of a possible streaming processing situation. In Figure 4.9, the state of the stream, represented by the white boxes, shows that an accident(A) has just occurred. The left part is recognized by the combinator `is10ItemsAvgSpeedBeforeAccident` and the right part is recognized by `isVehicleSpeedSum`. The blue and red boxes shows the elements aggregated by the sliding window just before the accident, only 10 vehicles(V) are aggregated.



Figure 4.9: State of the stream when an accident just occurs

After this state, the vehicles will continue to arrive, and be processed by the algorithm.

Figure 4.10 shows an interesting step. While the last accident has passed for a little while, the split combinator is forced to keep in memory the previously aggregated values until another accident occurs,

<sup>1</sup>The algorithm will not return a correct result if there is not at least 10 vehicles passing before each accident, but it is considered as an edge case that never happens.

because the `is10ItemsAvgSpeedBeforeLastAccident` combinator is made this way. However, the sliding window, being part of the nested combinator, must continue to go on to aggregate the data of the last 10 vehicles, in case another accident occurs.

Two vehicle aggregations are then stored in memory. Those are represented by the blue boxes. The red box containing a question mark represents the assumption that the next incoming item may be an accident.

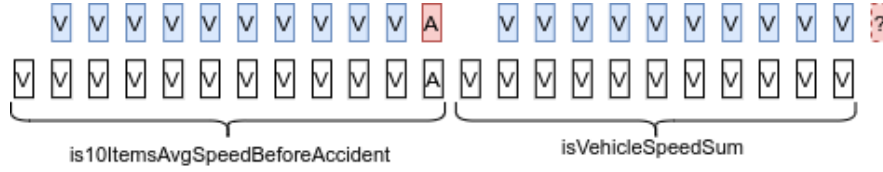


Figure 4.10: State of the stream when an accident have already occurred, but another can happen anytime soon

If ever a new accident occurs, as it is the case in Figure 4.11, then only the latest results aggregated by the sliding window are retained. This pattern already proves that the memory complexity can't be linear in reference to the split combinators.



Figure 4.11: State of the stream when a second accident just occurs

### Iter

The *iter* combinator must also maintain a state between iterations. Thanks to the unambiguous iteration guarantee imposed by the symbolic regular expressions, it is not necessary to keep in memory several splitting possibilities for the same iter combinator. This would be the case if this constraint was not present, as demonstrated in Section 3.6.3 using the multiset semantic. This combinator does not face the same problem as the split combinator, and is spared having to consider several splitting possibilities. However, the *iter* combinator is still required to always keep two separate states in memory.

Let's imagine a query using an *iter*( $f, op$ ) combinator to recognize the stream decomposition represented on Figure 4.12.

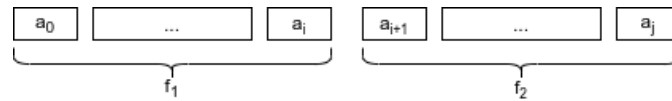


Figure 4.12: Initial situation

The part  $f_1$  is a completed iteration, and the part  $f_2$  is the current iteration. Each part will emit its own result. This decomposition must be kept in memory as an intermediate state as long as the domain of  $f_2$  is recognized. The result is evaluated while combining of these two parts.

This state will be updated at the arrival of an item  $a_{j+1}$ . Two cases are possible:

- The part  $a_{i+1}$  to  $a_{j+1}$  is recognized as the extension of the current iteration;
- The part  $a_{j+1}$  is recognized as the start of a new iteration.

Once a new iteration start, the two previous iterations can be aggregated, as they will not be modified anymore. Then, the cycle continue with a new decomposition between the aggregated previous iterations and a new one.

Unlike the *split* combinator, using *iter* has less of an impact on execution time and memory, but the combinator still required to potentially keep multiple state in memory

### Complexity

Because of the different states to be tested and maintained, the complexity is polynomial, both in terms of memory and time, depending on the size of the regular expression corresponding to the QRE. The way these states are maintained depends on the implementation. However that's not the only factor to take into account.

The overall complexity of a QRE  $f$  to evaluate a stream  $w$ , such that  $[f](w) : T_1 \times T_2 \times \dots \times T_k \rightarrow C$ , must also take into account the evaluation of the basic predicates,  $\phi$ , as used in the atom combinator. The maximum time needed to evaluate predicates appearing in  $f$  on data values, is noted  $time_{\phi-eval}(f)$  [10].

In addition, each time the stream is recognized by a combinator, the operations must map values to parameters from  $T_1$  to  $T_k$ , and evaluate the result, which adds an additional complexity. The maximum time (resp. memory) needed to perform this evaluation is noted  $time_{\tau}$  (resp.  $mem_{\tau}$ ) [10].

Finally, the time and memory to evaluate each item of the stream are:

- Time:  $poly(|f|) time_{\tau} time_{\phi-eval}(f)$
- Memory:  $poly(|f|) mem_{\tau}$

The time and memory necessary to evaluate the whole stream  $w$  are:

- Time:  $|w| poly(|f|) time_{\tau} time_{\phi-eval}(f)$
- Memory:  $|w| poly(|f|) mem_{\tau}$

### 4.3.3 Implementation

StreamQRE is written and compiled in Java. It is then executed on the Java virtual machine. The Java language is an object oriented programming language. Although some features inherited from functional programming, such as Stream API, have been incorporated into the language in recent years, it has been designed around the imperative programming paradigm.

Therefore, all combinators (*split*, *iter*, etc.) are reified in the form of object instances (resp. **QReSpit**, **QReIter**, etc.). The internal structure of objects defines how they will handle the stream.

Each combinator is implemented via several methods:

- **start**: Initializes the internal structure of the combinator. This method is called at the launch of the evaluation of the algorithm.
- **next**: Consume the data items. On receipt of each new item, the method evaluates the stream on the regular pattern represented by the combinator. It keeps the result if the concatenation of the previous items with the one newly received is recognized.
- **getOutputStack**: Allows to retrieve the intermediate results of each combinators.
- **getOutput**: Returns the result of the evaluation of the function.

As combinators have a hierarchical structure, a suitable implementation was needed. At their creation, the constructors of combinators, which are build for using nested combinators, will take them as parameters. In the same way, when initializing or processing an item, each **start** and **next** method of a combinator will also call the **start** and **next** methods of the nested combinators.

The parameters and the results are transferred between the combinators in a bottom up way. During start and next methods, combinators are mutating internal data structures, then return their new state to their parent combinator.

In imperative programming, sequences of instructions are executed to update the state of the program. In the case of QREs, as shown in Section 4.3.2, stateful combinators (*split* and *iter*) must keep the results of the current evaluation of the stream in memory. These results will be used when new items are received. Meanwhile, they are stored in stacks



The stacks of values have been chosen for their compact representation. Stacks are provided with standard operations:

- **push**: Add a new item on the top of the stack.
- **pop**: Remove the item on the top of the stack.
- **getTop**: Get the item on the top of the stack without removing it.

#### 4.3.4 Example

To illustrate the theory demonstrated above, a concrete use case of a quantitative property calculated by StreamQRE is explained. The example will also help for the construction of a new implementation later in the thesis.

As an example, let's suppose that, in traffic management, specific patterns exist to prevent increase in traffic density. One of them is the monitoring of sequences of two trucks following each others. When it happens, traffic load increases and can lead to traffic jam behind them.

To solve this problem, a query is created to monitor the arrival of each sequence of two trucks among vehicles. The stream processing is done with the alphabet  $\Sigma = \{c, t\}$ , where  $c$  is a car and  $t$  is a truck. The result of the query is the number of vehicles preceding the trucks. This number is reset to 0 after two trucks are observed. Note that this example is not realistic and just serves as a demonstration.

The following query, made of several combinators, will count vehicles behind two trucks. The `isVehicleToken` combinator recognizes each vehicle, whether it's a car or a truck. The `getItemType` method gives the capacity to know the nature of each incoming item. If the predicate is true, the `x -> 1D` function returns a value of 1 in Double number format. As for the `isTruckToken` combinator, it filters vehicles to recognize only those of the TRUCK type. The `countVehicles` combinator counts all incoming vehicles regardless of their type thanks to its inner combinator `isVehicleToken`. The function `x -> x + 1` successively adds the value 1, for each new vehicle, from a starting value of 0. The sum is each time returned as is via the method `x -> x`. The `twoTrucks` combinator recognizes a direct sequence of two trucks. It uses two inner `isTruckToken` combinators for this purpose. By the concatenation between `countVehicles` and `twoTrucks`, the `countVehiclesBeforeTrucks` combinator returns the number of vehicles observed by the left inner combinator when the right combinator matches the two trucks signal. The number of vehicles `x` is returned as a result. The last combinator, `iterCountingSequence`, repeats the pattern observed by the combinator `countVehiclesBeforeTrucks` and emits a result only when the latter is recognized again.

**Listing 9:** A QRE implemented with StreamQRE to count the number of vehicles behind two trucks

```

1 private static QReSplit<StreamingDto, Double, Double, Double>
   ↳ numberOfVehiclesBehindEachSequenceOfTwoTrucks() {
2     var isVehicleToken = new QReAtomic<StreamingDto, Double>(x ->
       ↳ Objects.equals(x.getItemType(), CAR) ||
       ↳ Objects.equals(x.getItemType(), TRUCK), x -> 1D);
3     var isTruckToken = new QReAtomic<StreamingDto, Double>(x ->
       ↳ Objects.equals(x.getItemType(), TRUCK), x -> 1D);
4     var countVehicles = new QReIter<>(isVehicleToken, 0D, (x,y) -> x + 1, x -> x);
5     var twoTrucks = new QReSplit<>(isTruck, isTruck, (x,y) -> 1D);
6     var countVehiclesBeforeTrucks = new QReSplit<>(countVehicles, twoTrucks,
       ↳ (x,y) -> x);
7     var iterCountingSequence = new QReIter<>(countVehiclesBeforeTrucks, 0D, (x,y)
       ↳ -> y, x -> x);
8     return iterCountingSequence;
9 }

```

As a regular expression, the pattern associated with the QRE would be written:  $(c^* \cdot t \cdot t)^*$ . The evaluation of this query with its combinators will be explained in the following demonstration.

The sequence of cars and trucks used for the demonstration will be the one showed on Figure 4.13.

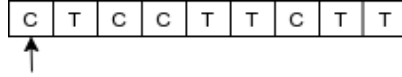


Figure 4.13: The word to parse starting at the left.

To demonstrate the evaluation of the queries at runtime, the combinators will be presented graphically as a tree. It's an easily understandable representation since the tree visually represents the nesting of the combinators.

Figures 4.14 and 4.15 illustrate each step of the demonstration. The combinators tree is placed at the very top of the images. Below, each section, delimited by dotted lines, describes the behavior of the algorithm when receiving a new element. The type of the received element is written in the box at the left of each section. Under each part of the tree, the stacks inside the sections describe the current state of the algorithm for each corresponding combinator. Note that only the atom combinators keep a stack in memory. In the top of each section is represented the state at the arrival of the element, and at the bottom of the section is represented the state after the processing of the element.

The left part of the tree is in charge of counting vehicles and the right one detects the presence of the two trucks. Each branch of the tree ends with an atom combinator.

The first section is a bit special, it's an initialization step that automatically occurs at the launch of the algorithm. Because the iter combinator accept an empty word, the stacks are initialized at 0 at this step.

In each section, under each atom combinator, if the current element matches with its predicate, the element is accepted by the pattern and the current stack is moved under the next atom combinator. In this way, the stream is recognized part by part as the elements arrive. The acceptance is visually represented by a green square, and the stack displacement is represented by the arrows. When moving the stack, its size is reduced by applying the eventual outer combinator operations. For example, between the two first columns the counting operation is applied by the iteration over each vehicles.

If the element doesn't match, the stack is dropped and the system frees the memory space. This event is represented by a red square and a cancel labelled arrow.

Let's detail one section. During the first step after the initialization, the *c* element is recognized as a vehicle. Considering that, the iter operation is applied and the new stack is moved under the next combinator. This stack becomes the base of the next combinator evaluation, waiting for a next element. As each iteration relies on the previous one, the new stack is also kept in the iter combinator as an intermediate result for the processing of the future elements.

With multiple stacks moving from a combinator to another, the parallel computations are easily maintainable. The next steps are similar until two *t* elements appear in the stream. In the third section on Figure 4.15, a result to the whole QRE is produced by recognizing one last *t*, and applying the operations of the two outer split combinators. Four vehicles was preceding the two trucks.

Then, a next iteration starts due to the `iterCountingSequence` combinator, and the same loop restart. This loop will repeat until the algorithm is switched off.

After this demonstration, the theory explaining the relationship between the number of combinator composing the QREs and their complexities is easily understandable.

From writing the queries to analyzing their implementation, the entire StreamQRE language has now been reviewed.

## 4.4 NetQRE

Another extension of QREs is NetQRE. Unlike StreamQRE, the language is geared towards a specific need, network monitoring. It has a smaller number of combinators available, for example combination and window partitioning are not part of it, but the focus is on performance when evaluating the stream. Some other tools have been added for this purpose and will be explained here. The language has moved away from the flexibility that is initially sought by QREs but remains interesting to analyze for the biases that it has chosen. Eventually, some of these features could be used for a new implementation.

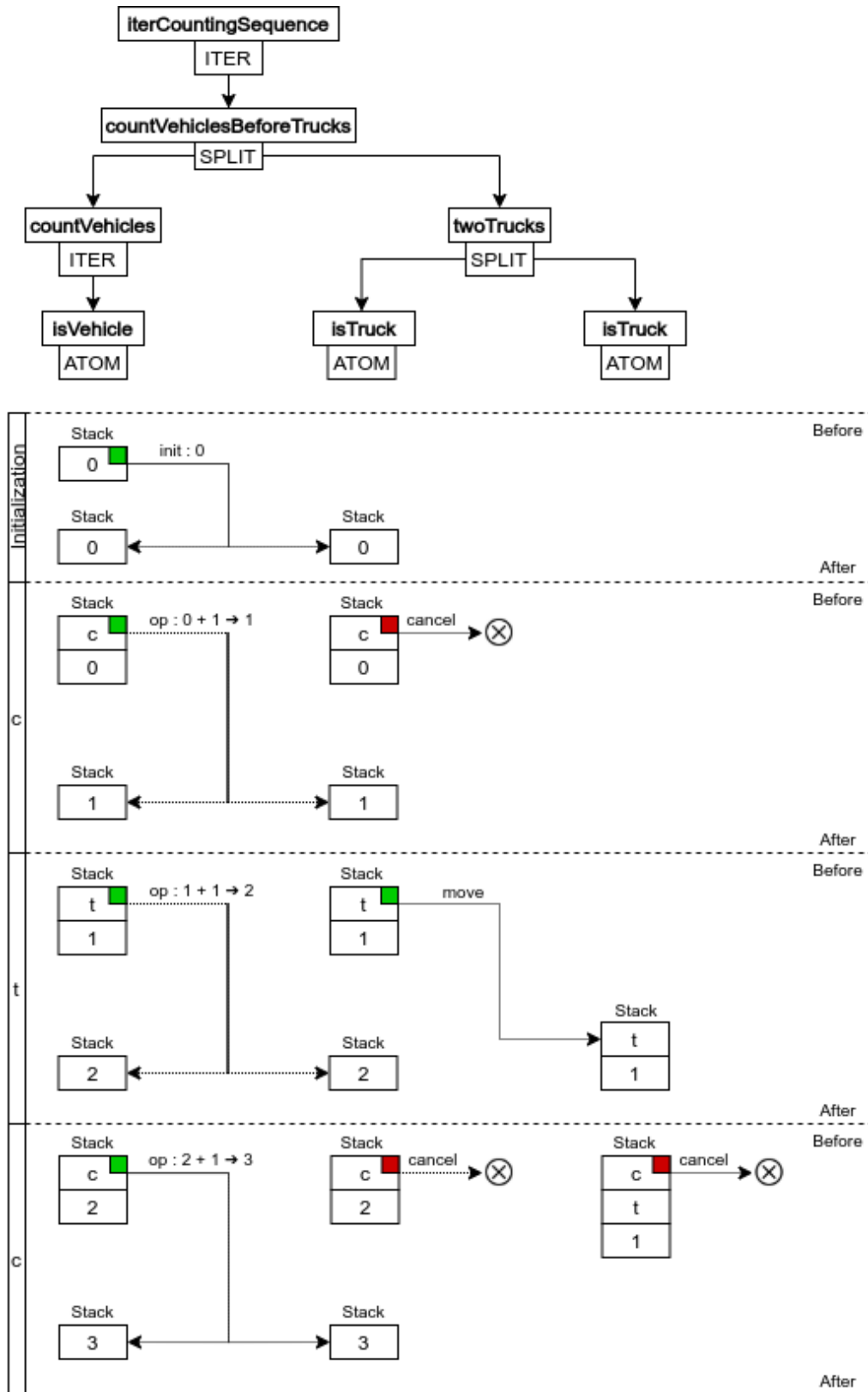


Figure 4.14: First part of the StreamQRE demonstration

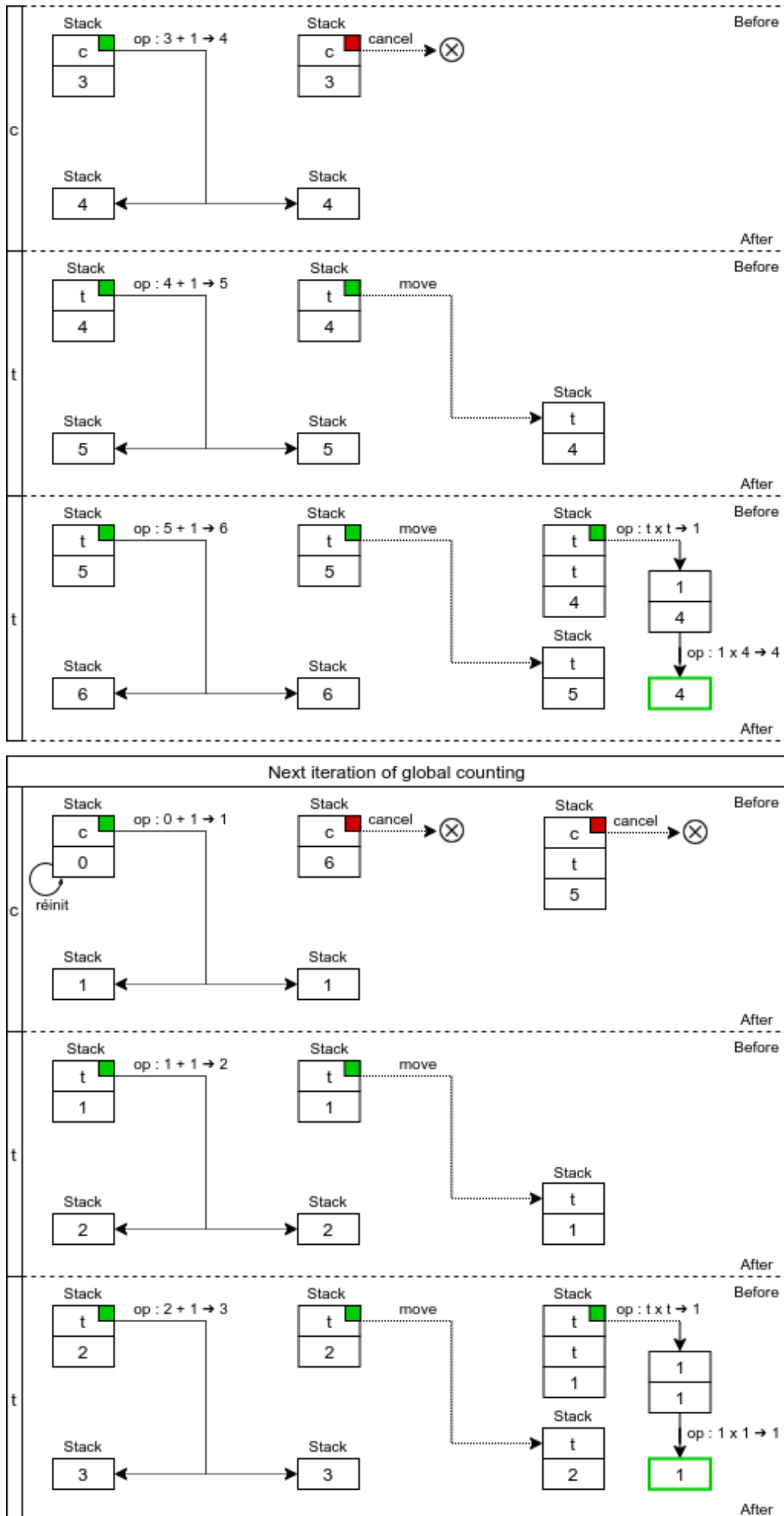


Figure 4.15: Second part of the StreamQRE demonstration

NetQRE is a high level declarative language that has a wide range of tools for quantitative network monitoring. Since it matches the specifications of QREs, quantitative policies are expressed using regular expressions.

By quantitative monitoring, the aim is to detect patterns in a stream of packets on a network during a connection, for example a VoIP session. The filtering by pattern matching is used to collect measurements that capture application-level or session-level semantics. A state is maintained on those measurements. The result of the monitoring produces analysis reports or performs actions dynamically on the network or on the filtered packets.

As explained in Section 1.7.3, the need for an easy-to-write expressive language that can produce dynamic updates in response to traffic engineering was necessary in network monitoring.

To fulfill this role, the language was not written in an imperative programming language (Java, C++,...). Unlike StreamQRE, which is aimed for a more generalist use and therefore can use a general programming language, NetQRE has no equivalent for a use intended for network monitoring. The use of a domain-specific language was necessary.

The language design has been divided into three parts:

- The specifications, a DSL based on QREs;
- The compilation, a symbolic automaton going along with a series of instructions that describes how to run the specifications;
- The implementation, a compiler to transform the specifications into an executable that can run with low memory footprint and a runtime to execute the output.

#### 4.4.1 Language

The language complies exactly with the specifications of the QREs. However, it extends them to suit the intended use case. The authors define them as PQREs, for Parametrized Quantitative Regular Expressions [50]. They differ from QREs by two criteria.

First, the regular expressions are applied on the packets, and no longer the separate symbols. A network packet is a sequence of bytes. The information contained by a packet can be used in predicates and operations.

As the language is specific to network monitoring, it contains several built-in features among which we find:

- Specific types, such as IP, Port or Connection;
- Specific functions, such as `srcport` or `srcip` to retrieve the source IP and port of packets;
- Specific aggregate operations, such as `sum`, `average`, `max`, `min` which can be used with `iter` combinator;
- Specific actions to generate alerts or send updates to switches.

A specific grammar and language to the DSL have been defined. They include the `split`, `iter` and `streaming` composition operators. Those will not be re-explained, as their behaviour is exactly the same compared to the original specifications. They also include an aggregation operator which will be analyzed more deeply as it's a key part of the implementation of NetQRE.

In fact, the second specific criteria of NetQRE is strongly related to this aggregation operator. The language has the possibility to use parameters in the predicates. The specificity of these parameters is that they will possibly only be known at runtime. For example, aggregating all the IPs from previous packets, and keeping them into an array to compare them to the one of the packet being processed is possible.

The parameters can also be used without the aggregation operator. To explain their use case, three examples will illustrate the differences.

Before presenting these examples, it is important to give the basics of the DSL. A function in NetQRE is written:

```
sfun type func_name (type var) = exp
```

where `type` is the return type of the function, followed by its name, then the function arguments with their types and then the body, `exp`.

**Example 4.4.1.** First, let's build a simple function with the `iter` combinator. The following function contains a regular expression but don't use any predicate. Its results computes the total of the number of packets in the stream. The operation `sum` is a predefined operation which updates the variable count after having read each packet.

The regular expression used in this example is `/./`, and next to it, a nullish coalescing operator is represented by `?`. It emits the value 1 when the expression matches. Because the character `'.'` matches a single packet and `'/'` delimits the start and the end of the regular expression, each packet of the stream matches the expression one by one.

```
sfun int count = iter (/./?1 , sum )
```

The regular expression can be modified by including a predicate to count only the packets with a predefined source IP. The predicate is written inside the brackets.

```
sfun int count = iter (/ [srcip='1.0.0.1']/?1 , sum )
```

The syntax remains the same with the `split` operator or any other, only the expression part changes. For example, the `split` operator expression have the syntax `split(regexp, regexp, op)`.

**Example 4.4.2.** In this second example, built-in functions, function composition and the use of parameters are explained. The aim is to define a function to match with the packets of a whole user session from the beginning until the end, and to aggregate over the number of sessions of a same connection.

Among the built-in functions, NetQRE contains a macro function which filters TCP packets for a specific connection, `is_tcp(c)`. The variable `c` is the connection.

The macros can be easily used in predicates. The function `filter_tcp` uses `is_tcp` macro to emit the last packet received if it matches with the predicate. The word `last` is a built-in keyword in the language to return the last packet. The connection parameter is only known at runtime. The parameters are initialized by the calling program or through another function.

```
sfun packet filter_tcp (Conn c) = /*[ is_tcp (c)] / ? last
```

The final function uses streaming composition with the operator `>>` for, successively, filtering TCP packets of a connection, then for filtering packets with `SYN` and `FIN` flags which are used in the `iter` predicate to determine the beginning and the end of the session.

```
sfun int count_flow ( Conn c) =
  filter_tcp (c) >>
  filter_flag ( flag=SYN || flag=FIN ) >>
  iter (/ [ fin =1]*[ syn =1]*[ syn =1][ fin =1] / ? 1 , sum)
```

The function `count_flow` totals the sum of the packets of each session. Note that the function `filter_flag` is not defined here, but is very similar to the `filter_tcp` function.

**Example 4.4.3.** This last example highlights the biggest difference between NetQRE, StreamQRE and the base specifications of QREs, which is the aggregation operator.

In the next function, the `exist` predicate checks whether a source IP `x` appeared in the stream. The function `count_distinct_ips` aggregates each IP that appeared in the stream and counts them.

```
sfun bool exist (IP x) = /*[ srcip=x ].*/
sfun int count_distinct_ips = sum { exist(x)?1:0 | IP x }
```

The aggregation expressions are split in two parts. The predicate is at the left of the vertical pipe and the type of the parameter used in the predicate is at the right.

For each packet, the predicate is checked, then the value corresponding to the result of the predicate evaluation, here 1 or 0, is aggregated. In this case, it's the `sum` operator which aggregates the values.

In addition, the algorithm should keep track of all the previous aggregated values, in this case all the previous IPs.

That's the breaking change compared to the QRE specifications. Even if the stream is read in a single pass, the performance and memory consumption is not bounded by the length of the regular expression anymore. For example, in this use case, the memory consumption is clearly bounded by the number of IPs which depends on the length of the stream.

#### 4.4.2 Compilation

The variant of QREs using dynamic parameters is called PSREs, meaning *Parametrized Symbolic Regular Expression*. It is not compiled using a simple cost register automaton anymore. The CRAs were initially extending symbolic automata to be able to evaluate predicates. Their equivalent for the evaluation of NetQRE expressions is called *Parametrized Symbolic Automata* (PSA) and is able to evaluate dynamic predicates.

The general operations of the combinators remain the same. Split and iter combinators maintain a minimum state to produce a cost for each received packet while combining them with all the previous ones. If the QRE specifications wasn't extended, the result of a NetQRE expression could have been interpreted by a symbolic automaton and would have retained all the properties of regular functions. Moreover if only static parameters were used in the predicates, the predicates could have been instantiated once at runtime, then evaluated multiple times but never modified.

However, with the addition of dynamic parameters in the predicates, the number of states for a regular function have been strongly decoupled. Intuitively, there are potentially different states for each possible value of the parameters, and therefore an exponential number of possibilities. To solve this, a naive solution would have been to instantiate a different symbolic automaton for each possible parameter value.

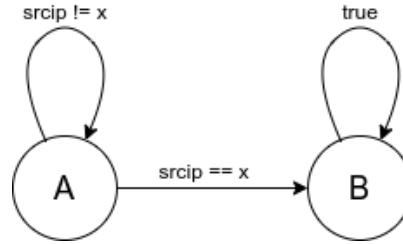


Figure 4.16: Example of PSA with parameters

In the preceding example (4.4.3), in which IP addresses are aggregated to only keep distinct ones, the number of automata would have been the same as the number of possible addresses,  $2^{32}$ . Figure 4.16 represents such a PSA without taking into account its dynamical parameter. Figure 4.17 represents its instantiation into a set of symbolic automata<sup>2</sup>.

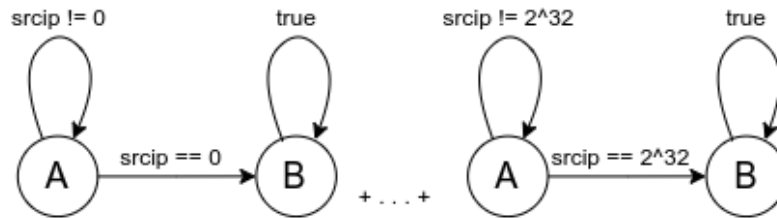


Figure 4.17: Example of PSA with instantiated parameters as SAs

Hopefully, a more clever approach has been found. The solution is the lazy instantiation of the states composing the PSA, using an algorithm to update it on demand. The reasoning comes from the fact that among the set of possible values for a parameter, many of them will never be considered.

<sup>2</sup>These figures were built while relying on an example in a conference about NetQRE [49]

Concretely, this lazy instantiation can be materialized by a decision tree. At the beginning, the decision tree contains one single node. Thinking back to the representation of a PSA as a set of SAs, this is logical as each automata will start at the same initial state, no matter the parameter values.

When a new packet arrives, all the transitions of the SAs are evaluated with the new parameter value. If the predicate is verified, then the decision tree is updated.

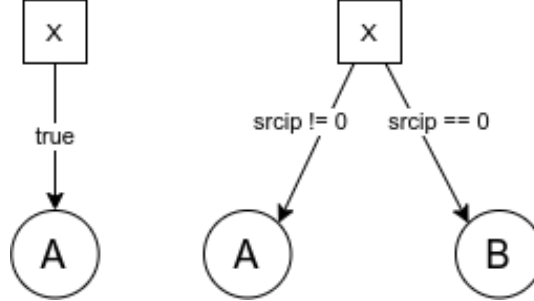


Figure 4.18: Example of PSAs as a decision trees

As an example, Figure 4.18 shows the instantiation of the `exist` function before, at the left, and after, at the right, reading the first packet of the stream with the source IP 0. As more IPs are read, the guards will be updated in the decision tree.

### 4.4.3 Implementation

An implementation of NetQRE split into two part, a compiler and a runtime, has been written in C++. Apart from the inclusion of libraries specific to network monitoring, one main difference between it and StreamQRE exists.

While StreamQRE only use a stack to store the intermediate states, NetQRE runtime additionally use a tree data structure to store dynamic parameters values.

This implementation will not be detailed here as it is strongly coupled with the domain of network monitoring. In this thesis, the goal is to keep using a generalist approach and not to be locked into a specific language implementation. However, Appendix B.2 gives the necessary information to download the C++ library.

### 4.4.4 Evaluation

Even if the implementation is domain specific, it shows that QREs are able to adapt to different implementations quite easily. This section details the results of tests that were performed under different conditions during the implementation of the language [50].

At application, session, or flow level, queries written with NetQRE show great expressiveness. Compared to other systems, the extent of the querying possibilities is much wider. The programs are very concise, rarely exceeding 20 lines. In comparison, handwritten implementations require double, triple, or even more lines, while being error-prone.

In terms of performance, the data processing throughput is very similar to that of an equivalent manual implementation. Compared with other existing systems, NetQRE is 1.8 to 23 times faster. Such a difference is understandable because few tools are targeting its specific use case, and some of them are not even compiled into low-level code, that highlights the NetQRE usefulness.

Regarding memory consumption, NetQRE is significantly more demanding, up to 60% compared to a manual implementation. Even if it is not put forward by the authors, the dynamic parameter instantiation probably does not have a negligible impact on the overhead costs. However, this increase has no real impact on the necessary hardware.

In addition to performance testing in sandbox mode, multiple end-to-end validations have shown that NetQRE meets expectations in real use cases.



## 4.5 Road traffic monitoring use case: A StreamQRE solution

In Section 1.4, an implementation of a naive algorithm was built in order to estimate the maximum density of vehicles acceptable by a tunnel.

In this section, an similar algorithm is developed with an identical purpose, but this time using all the knowledge gathered in the last chapters. The algorithm is implemented with StreamQRE as a regular function. The design of the algorithm will be explained. Then, its quality will be estimated by evaluating the same criteria that had been used for the naive algorithm.

The choice of a reimplementaion with DReX or NetQRE was ruled out because these two languages, although having participated in the evolution of QRE language, are specific to a particular domain which is not compatible with the current example.

Before building the implementation of the algorithm, here is a summary of the context in which this processing algorithm takes part.

The maximum tunnel density is measured by the maximum number of vehicles capable of passing through the tunnel in one hour. The estimate is based on the processing of a data stream resulting from the observation of the vehicles passing through the tunnel. This data stream contains both the type of each vehicle and their speed. Additionally, it also contains tokens that mark the repetition of 15 minutes periods. Figure 4.19 represents motorcycles, cars, trucks and 15 minutes ending by the symbols M, C, T and  $end_{15}$ . Different parameters of the algorithm have to be adjustable in order to define the characteristics of the tunnel. That allows the algorithm to be used on different types of tunnel. Among these parameters is found the length of the slope of the tunnel, if there is one, and the level of habits of the drivers who pass through it.

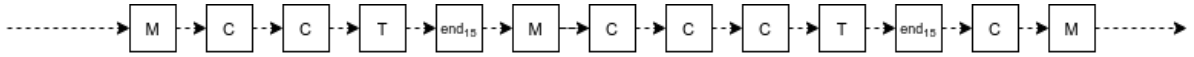


Figure 4.19: Representative example of the types of tokens making up the data stream

Then, the algorithm should be ran through a simulation testing using a stream of vehicles generated to reproduce a real situation. In this context, in the first sections of this thesis, the naive algorithm had shown inherent weaknesses in the way it was developed. These weaknesses have oriented the research on the characteristics necessary to efficiently process a data flow. Now is the time to see if there has been any improvement.

### 4.5.1 Algorithm

The processing of the data flow is partitioned into the computation of four factors:

- The theoretical capacity;
- The peak hour factor;
- The heavy vehicle factor;
- The type of driver factor.

Each of these factors was explained in Section 1.4. The first three will be implemented as QREs, then combined to be able to compute the practical capacity of the tunnel in real time. The fourth factor is a statically predefined parameter that can't be computed based on the stream.

#### Theoretical capacity

The following query is divided into two functions. The first one applies a transformation on the average speed to compute the theoretical capacity of the tunnel. The second one computes the average speed of the stream itself using the data of the vehicles observed.

In the second query, each 15 min token is considered to have zero speed, and each vehicle token returns its own speed. This is done in the combinators `is15MinToken` and `isVehicleSpeedToken`. They both filter the type of object received via the method `getItemType`. Inside the `isVehicleSpeedToken` combinator,

the lambda producing the result first casts the object to have access to the properties of its effective type, then gets the speed as a `Double` value. Then, the combinator `isSpeedSum` calculates the total sum of the speed values, switching between the vehicles and the timing tokens, depending on the nature of the elements, thanks to the combinator `isSpeed`. The sum is aggregated by the lambda function `Double::sum`, starting from an initial value of 0. The `isSpeed` combinator provides valid values by switching between its two inner combinators depending on whether the token is representing a 15 min signal or a vehicle.

According to the same logic, the count combinator calculates the number of vehicles observed. The difference lies in the presence of a new `isVehicleCountToken` combinator which, to count the vehicles, does not return their speed but just the value 1 as a `Double` number. The `isCountSum` combinator adds all the values received one after the other. The `isCount` combinator switches between its inner combinators to always produce a valid result, and reuses the `is15MinToken` combinator to ignore this type of token in the count.

The two combinators `isSpeedSum` and `isCountSum` are then combined to retrieve the average speed.

In the first query, a static computation is applied to the average speed to retrieve the theoretical capacity.

**Listing 10: A QRE implemented with StreamQRE to compute the theoretical capacity**

```

1  private static QReApply<StreamingDto, Double, Double>
   ↪  theoreticalCapacityPerTrafficLane() {
2      var isAverageSpeed = Algo.averageSpeedOfVehicles();
3      return new QReApply<>(isAverageSpeed, x -> x * 10.0 + 1200.0);
4  }
5
6  private static QReCombine<StreamingDto, Double, Double, Double>
   ↪  averageSpeedOfVehicles() {
7      var is15MinToken = new QReAtomic<StreamingDto, Double>(x ->
   ↪  Objects.equals(x.getItemType(), END15), x -> 0D);
8
9      var isVehicleSpeedToken = new QReAtomic<StreamingDto, Double>(x ->
   ↪  Objects.equals(x.getItemType(), VEHICLE), x ->
   ↪  ((VehicleDto)x).getSpeed().doubleValue());
10     var isSpeed = new QReElse<>(isVehicleSpeedToken, is15MinToken);
11     var isSpeedSum = new QReIter<>(isSpeed, 0D, Double::sum, x -> x);
12
13     var isVehicleCountToken = new QReAtomic<StreamingDto, Double>(x ->
   ↪  Objects.equals(x.getItemType(), VEHICLE), x -> 1D);
14     var isCount = new QReElse<>(isVehicleCountToken, is15MinToken);
15     var isCountSum = new QReIter<>(isCount, 0D, Double::sum, x -> x);
16
17     return new QReCombine<>(isSpeedSum, isCountSum, (x, y) -> x/y);
18 }

```

### Peak hour factor

This query computes the variance of traffic density based on the traffic passing through the tunnel. First, it calculates the density over a repeated 15-minute period with an interval of 15 minutes. The combinator `sumOfVehicle` aggregates the number of vehicles observed. For this purpose, it uses the combinator `isVehicleToken` which filters the streaming items to accept the ones representing vehicles, via the method `getItemType`. The `sumOfVehiclesDuring15Min` combinator limits this observation to a 15-minute period by stopping the iteration when a 15-minute token appears. This token is recognized by the combinator `is15MinToken`, via the method `getItemType`. The `is15MinToken` combinator constitutes the right part of the split concatenation.

The maximum density observed over repeated periods is then kept by the combinators `repeatOfSumOfVehiclesDuring15Min` and `sumOfVehiclesDuringLast15Min`. The first combinator repeats the detection

of successive 15-minute period. The second combinator concatenates this repetition with an indefinite vehicle iteration. This iteration contains the last vehicles observed while waiting for a new end of period token. These tokens being a series of vehicles without any 15-minute tokens, they can be recognized by the `sumOfVehicle` combinator. The two combinators keep only the maximum value among every aggregation period.

Similar logic is used to calculate the number of vehicles in an hour, except that the one hour period is limited by a sliding window. The combinator `sumOfVehicleDuring1Hour` aggregates the last four 15-minute periods and advances at 15-minute intervals to always keep the total number of vehicles observed over the previous hour. The method `Double::sum` computes this aggregation, and the parameter 4 limits the number of iteration inside the window. The combinator `sumOfVehiclesDuringLastHour` manages to emit the last hour count even if it's not the end of a 15-minute period. In this way, the queries have the same rate and produce a new result for each new token received. For this purpose, it uses the same technique as the combinator `sumOfVehiclesDuringLast15Min`, and recognizes both past iterations and the last vehicles by a concatenation with `sumOfVehicle` combinator.

The two results are then combined to calculate the variance using the combine combinator. The last hour density is divided by the maximum density observed over 15 minutes.

</> Listing 11: A QRE implemented with StreamQRE to compute the peak hour factor </>

```

1 private static QReCombine<StreamingDto, Double, Double, Double> peakHourFactor() {
2     var isVehicleToken = new QReAtomic<StreamingDto, Double>(x ->
3         ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1D);
4     var is15MinToken = new QReAtomic<StreamingDto, Double>(x ->
5         ↪ Objects.equals(x.getItemType(), END15), x -> 1D);
6
7     var sumOfVehicle = new QReIter<>(isVehicleToken, 0D, Double::sum, x -> x);
8     var sumOfVehiclesDuring15Min = new QReSplit<>(sumOfVehicle, is15MinToken, (x, y)
9         ↪ -> x);
10
11     var repeatOfSumOfVehiclesDuring15Min = new QReIter<>(sumOfVehiclesDuring15Min,
12         ↪ 0D, (x,y) -> x > y ? x : y, x -> x);
13     var sumOfVehiclesDuringLast15Min = new
14         ↪ QReSplit<>(repeatOfSumOfVehiclesDuring15Min, sumOfVehicle, (x, y) -> x > y ?
15         ↪ x : y);
16
17     var sumOfVehicleDuring1Hour = new QReWindow<>(sumOfVehiclesDuring15Min, 0D,
18         ↪ Double::sum, x -> x, 4);
19     var sumOfVehiclesDuringLastHour = new QReSplit<>(sumOfVehicleDuring1Hour,
20         ↪ sumOfVehicle, (x, y) -> x);
21
22     return new QReCombine<>(sumOfVehiclesDuringLastHour,
23         ↪ sumOfVehiclesDuringLast15Min, (x, y) -> x / (4 * y));
24 }

```

### Heavy vehicle factor

The heavy vehicle factor is computed based on the percentage of trucks in the targeted road traffic stream. The query `percentageOfTrucks` calculates it by aggregating on one hand the total number of vehicles with the combinator `isVehicleSum`, and on the other hand the number of trucks with the combinator `isTruckSum`. In each part, the predicates used in the nested combinators allow an unambiguous iteration on the whole stream while each proposing a different interpretation.

The combinator `isVehicleSum` iterates over each vehicle token and aggregates their sum, with the method `Long::sum`, starting from an initial value of 0. To omit all tokens that are not vehicles, it uses the inner combinator `isVehicleValue`, which switches between `is15MinToken` combinator and `isVehicleToken` combinator. Combinator `isVehicleToken` returns a value of 1 for each vehicle while the other always returns 0, so that only vehicles are taken into account in the aggregation.

The combinator `isTruckSum` works exactly in the same way. The difference lies in the count of vehicles, limited to trucks only with the combinator `isTruckToken`. The tokens which are not trucks are recognized by the other combinator, `isNotTruckToken`. This combinator accepts both tokens which are not vehicles, via the `getItemType` method, and those which are vehicles but not trucks, via the `getType` method. For all of them, the value 0 is produced, in opposite to the `isTruckToken` combinator which produces 1 values.

The count of vehicles and trucks is combined to calculate the ratio between all vehicles and trucks only, and thus know the truck percentage. Then, the percentage is transformed into the actual resulting factor within the `heavyVehicleFactor` query. The eventual length and slope grade of the tunnel can be passed as parameters. This allows to easily get the equivalence factor from an external matrix. This matrix is attached in Appendix C.1.1.

**Listing 12:** A QRE implemented with StreamQRE to compute the heavy vehicle factor

```

1  private static QReApply<StreamingDto, Double, Double> heavyVehicleFactor(
2      Double length,
3      Double slopeGrade
4  ) {
5      return new QReApply<>(percentageOfTrucks(), x -> 1.0 / (1.0 + x *
6          ↪ (matrix.getCoefficient(length, slopeGrade, x) - 1.0)));
7  }
8  private static QReCombine<StreamingDto, Long, Long, Double> percentageOfTrucks() {
9      var is15MinToken = new QReAtomic<StreamingDto, Long>(x ->
10         ↪ Objects.equals(x.getItemType(), END15), x -> 0L);
11      var isVehicleToken = new QReAtomic<StreamingDto, Long>(x ->
12         ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1L);
13      var isVehicleValue = new QReElse<>(isVehicleToken, is15MinToken);
14      var isVehicleSum = new QReIter<>(isVehicleValue, 0L, Long::sum, x -> x);
15
16      var isTruckToken = new QReAtomic<StreamingDto, Long>(x ->
17         ↪ Objects.equals(x.getItemType(), VEHICLE) && ((VehicleDto)x).getType() ==
18         ↪ VehicleType.TRUCK, x -> 1L);
19      var isNotTruckToken = new QReAtomic<StreamingDto, Long>(x ->
20         ↪ !Objects.equals(x.getItemType(), VEHICLE) || ((VehicleDto)x).getType() !=
21         ↪ VehicleType.TRUCK, x -> 0L);
22      var isTruckValue = new QReElse<>(isTruckToken, isNotTruckToken);
23      var isTruckSum = new QReIter<>(isTruckValue, 0L, Long::sum, x -> x);
24
25      return new QReCombine<>(isTruckSum, isVehicleSum, (x, y) ->
26         ↪ x.doubleValue()/y.doubleValue());
27  }

```

### Practical factor

Thanks to the recursive structure implemented by StreamQRE, the three previous factors can easily be combined and computed in parallel as is done in the following query `computeMaximumLaneCapacity`. This query will emit the final result giving the maximum practical lane capacity factor in a tunnel.

The combine combinator passes the streaming items to all inner queries at the same time. The `peakHourFactor` result is combined with the `heavyVehicleFactor` result. The lambda function multiplies the two results with a static `habitualUseFactor` argument. The combine combinator is itself nested inside another combine combinator, and its result is multiplied with the result of the `theoreticalCapacityPerTrafficLane` query.

</> Listing 13: A QRE implemented with StreamQRE to compute the maximum lane capacity </>

```

1 public static QReCombine<StreamingDto, Double, Double, Double>
   ↪ computeMaximumLaneCapacity(
2     Double length,
3     Double slopeGrade,
4     Double habitualUseFactor
5 ) {
6     return new QReCombine<> (
7         theoreticalCapacityPerTrafficLane(),
8         new QReCombine<> (
9             peakHourFactor(),
10            heavyVehicleFactor(length, slopeGrade),
11            (PHF, HVF) -> PHF * HVF * habitualUseFactor
12        ),
13        (TC, R) -> TC * R
14    );
15 }

```

### 4.5.2 Performance

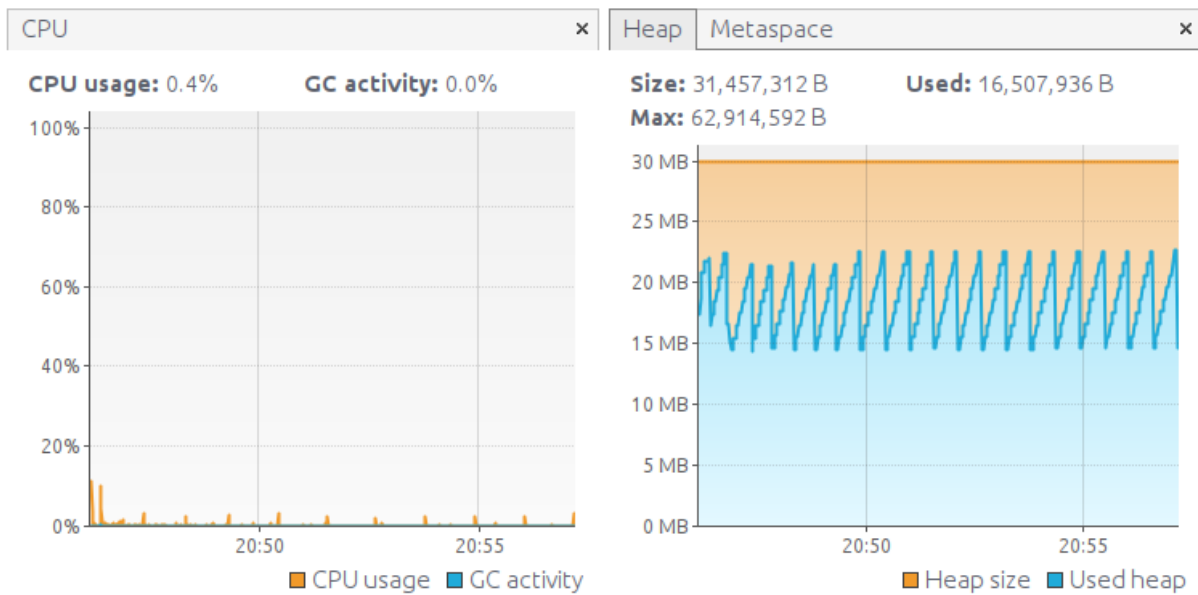


Figure 4.20: Performance of the StreamQRE solution

In Figure 5.14, the evolution of the performance of the application can be observed during a 10-minute simulation. This simulation has been done with the server and the web application developed for this thesis to generate data streams. Information on the application is attached in Appendix D.

In comparison with the very good performances which had been produced by the naive solution, the evaluation of the algorithm developed with StreamQRE reaches equivalent, or even almost identical metrics. The CPU load remains very low and the stream processing consumes a constant amount of memory which is freed at regular intervals by the JVM's garbage collector.

### 4.5.3 Modifiability

Focusing on the separation of concerns, thanks to the QRE recursive structure, the processing of the different factors are divided into several functions, while being able to be calculated in parallel. Each

part of the algorithm is isolated from the others, which makes it easy to modify the internal behavior of the algorithm.

Some factor computations require to be parameterized. Thanks to the use of lambda functions as operations for the combinators, external effective parameters are easily usable within the operations.

Inside each function, the structure, induced by the use of QREs, requires the separation of the operations relating to each part of the stream. A combinator isolate the transformations on a predefined part of the stream. Thanks to that, as long as each combinator is correctly named, it is easy enough during a proofreading to understand again how the implementation works, as well as to update it.

These different characteristics make this solution a much more easily maintainable implementation than the naive solution was.

#### 4.5.4 Testability

Regarding testability, the naive solution had showed a lack of modularity which prevented any isolated test on a part of the code during the development, or the execution of the application. This is no longer the case in this implementation. The implementation allows to isolate the tests on a particular factor, as it allows to easily make tests on the whole implementation.

#### 4.5.5 Conclusion

In conclusion, such an implementation makes it possible to continue to guarantee good performance even if the algorithm grows, while maintaining a good modifiability and a pleasant testability to develop even without knowing all the tricks of the Java language or the implementation of the combinators.

The size of the implementation is relatively concise. The different factors are easily identifiable and modifiable. Integration into any environment is easy due to the low constraints imposed by the combinators, and their relatively small number.

### 4.6 Conclusion

At the end of this chapter, all the existing implementations of QREs have been reviewed.

Starting with DReX, a first implementation which showed strong constraints on the domain on which the expressions can be applied. These constraints were present with a justification because the respect of the specifications established for the QREs was only partial. The importance of this implementation resides in its leading towards other implementations.

The second implementation, StreamQRE, meets specifications literally. It is a complete and generic implementation using stacks to manipulate the results received and emitted by the application of combinators. His analysis helped to define the transposition of the complexity constraints of QREs in a concrete implementation.

NetQRE, another already existing implementation, has shown the capacity of extension of the combinators, as well as the capacity of adaptation of the language for an application in a specific domain, in this case the network monitoring domain.

Finally, the use of the most suitable implementation for the field of road traffic monitoring, StreamQRE, has shown its efficiency through an example. A new implementation of a stream processing monitoring solution for computing the maximum density of a tunnel have been developed with this QRE implementation. The solution showed excellent performance, but also an ease of writing, a very good modifiability and testability, which differentiates it from the naive solution.

# Chapter 5

## LazyQRE

### 5.1 Introduction

In this chapter, a new implementation of quantitative regular expressions is proposed. It takes advantage of the functional programming paradigm and uses it through the functional features proposed by the Scala language. This new implementation is inspired by StreamQRE, in particular in its total compliance with the QRE specifications and its development oriented for generic use. The declaration of queries makes it easy to switch from StreamQRE implementation to this one, which is a great advantage. However, a clear evolution of the language is brought with some differences in the execution of the operations.

The operations carried out by LazyQRE are only executed when the user requests a result. Moreover, no partial operation is executed if it does not lead to a final result. This *lazy evaluation*, which gives its name to the library, saves many resources and gives more flexibility in the use of QREs. The implementation will be explained step by step by presenting some part of the code but the whole is attached in Appendix A.

The chapter begins with the descriptions of the features present in Scala which are useful for this new implementation. Then the explanations will mainly focus on the differences with StreamQRE, the advantages provided by this implementation but also the disadvantages that come with it. New solutions to some problems already analyzed in previous chapters, including the tunnel monitoring problem, will be exposed at the end of this chapter to give a concrete point of comparison.

### 5.2 Scala

Scala is a high level programming language which combines object-oriented and functional programming approaches. The language is statically typed, compiles, among others, to Java bytecode and run on a JVM (Java Virtual Machine).

Before proposing a new implementation of QREs in Scala, it is necessary to define the usefulness of this programming language. This section describes the Scala language, its functional approach, and the details of the implementation of some interesting features for our use case.

This section is written with a knowledge acquired while reading ‘*Functional Programming in Scala*’ [22] and ‘*Scala By Example*’ [42].

#### 5.2.1 Language flexibility

Recently, the Java language has evolved a lot by accelerating the rate of production of new versions. Twice a year, a new version comes out and, with it, new features are integrated into Java language capabilities. However, that has not always been the case. In 2003, when the Scala language was released, Java was quite poor and verbose in terms of syntax.

At this time, Scala is invented with the aim of creating a more user-friendly language, getting rid of restrictive or frustrating aspects in development. Despite the progress of Java since then, Scala is still

widely used today and has retained a guideline that allows it to meet different needs than its counterpart. Apart from functional programming, Scala is also known for its syntactic flexibility compared to Java.

Here are some advantages. The developer may omit semicolons for line endings. Functions benefit from syntactic sugar to be declared more succinctly. Scala automatically creates getters and setters for each declared field in a class. Beside that, a lot of other features have been integrated, but not all syntactic variations will be presented here. Overall the language has a more refined syntax compared to the Java language.

**Example 5.2.1.** As an example, here is how to declare a class in Java compared to Scala. The statement turns out to be much more concise in Scala. Both example classes are designed to build objects representing vehicles. They both contain a list of characteristics for the vehicle and its brand. In Java, to access the instance variables of an object, a `getXXX` method must be written, and a `setXXX` method must be used to be able to modify them. The `XXX` part is generally the name of the corresponding variable. In Scala, these methods are generated automatically by the compiler with the same name as the variables. A new brand can be assigned to a vehicle by the instruction `v.brand = "Toyota"`, with `v` being an object of type `Vehicle`.

To access the instance variables of an object outside of it, its access modifiers must be specified. The access modifiers set accessibility of classes, methods, and other members. In Java, the `public` keyword is put in front of each class, getter, setter that can be used by other classes, as is generally the case. To make things easier, in Scala, the default access modifier is `public`. So the developers don't have to explicitly specify this keyword.

</> Listing 14: An example of Java code </>

```

1  public class Vehicle {
2      private List<String> characteristics;
3      private String brand;
4
5      public Vehicle() {
6          characteristics = new ArrayList<String>();
7      }
8
9      public String getBrand() {
10         return brand;
11     }
12
13     public void setBrand(String name) {
14         this.brand = name;
15     }
16
17     public List<String> getCharacteristics() {
18         return characteristics;
19     }
20
21     public void setCharacteristics(List<String> characteristics) {
22         this.characteristics = characteristics;
23     }
24 }

```

</> Listing 15: An example of Scala code </>

```

1  class Vehicle {
2      var characteristics: List[String] = Nil
3      var brand: String = _
4  }

```

One difference that was present until Java 10 was the use of the `var` and `val` keywords which were only present in Scala. These two keywords allow to omit specifying the type of the variable in its declaration,



if and only if a value is assigned immediately to the variable.

Today in both languages, the compiler is smart enough to figure out the type of an expression at variable declaration. However, the keyword `val`, which creates an immutable variable (like `final` in Java), remains reserved for Scala.

### 5.2.2 Higher-order functions

In Scala, the functions are managed as first-class citizen, which earned them the name of higher-order functions. They can be used and manipulated like any other value. They can be assigned to a variable, but also passed as a parameter to a function. A function can even return other functions. Since version 8, some of these features are also available in the Java language. For example, Stream API works with a similar logic. However, the possibilities are still way more limited than in Scala.

**Example 5.2.2.** Let's explore the capabilities provided by higher-order functions. The following algorithm creates two custom functions, and pass them to `filter` and `map` functions, which are predefined higher-order functions for manipulating sequential data. The function `filterByCars` filters the list of vehicles to keep only cars, represented by `C`. The function `mapToOne` maps each element to the number 1. The whole query computes the sum of cars observed.

For each vehicle, its value will be checked as being equals to `"C"` with the method `.filter(filterCars)`. If so, the value goes to the next step. With the method `.map(mapToOne)`, the retrieved value is transformed from `"C"` to 1. Finally, with the `sum` method, each 1 value is added in the variable `count`.

</> Listing 16: Higher-order function example in Scala </>

```
1 val filterCars = (x:String) => x == "C"
2 val mapToOne = (x:String) => 1
3
4 val vehicles = List("C", "M", "T", "C", "T")
5 val count = vehicles
6   .filter(filterCars)
7   .map(mapToOne)
8   .sum
```

On top of that, the language gives the possibility to go even further and create custom higher-order functions. To do this, the parameter type, or return type, of the concerned function must be declared as being a function type definition itself. In the following code, a number is multiplied by 10, and then added to 5. The addition function is passed as a parameter to the multiplication function to allow the two functions to be performed successively in one call. Calling the `multiplyBy10` function on the value 3, as is done in the code, will give the result 35.

</> Listing 17: Custom higher-order function example in Scala </>

```
1 val addition5 = (x:Int) => x + 5
2 val multiplyBy10 = (fn: Int => Int, x: Int) => fn(x * 10)
3 val result = multiplyBy10(addition5, 3)
```

The use of functions within other functions is called *function composition*. To facilitate these patterns, which can sometimes become complicated to understand, the language gives the power of specific operators. The `compose` operator allows to easily create a new function resulting of the composition of two other functions. The following code gives exactly the same result as the previous one. The difference is that each function has been written separately before finally being combined to create a new one, on which is directly applied the value 3.

</> Listing 18: Function composition example in Scala </>

```
1 val addition5 = (x:Int) => x + 5
2 val multiplyBy10 = (x: Int) => x * 10
3 val result = (addition5 compose multiplyBy10)(3)
```

One last benefit of higher-order functions is *currying*. Currying is the transformation of a function which accepts several items as parameters into a partial function which accepts a single argument. This new function no longer returns the initially expected result. It returns a new function identically to the initial, but to which a parameter has already been pre-applied. This technique is called *partial application*.

In the following code snippet, the four results contain the value 5, resulting from the addition between 2 and 3. The differences lie in the application of this addition. The first function is a simple function which just applies the addition as it would have been possible to do in many other languages. The second and third functions are higher-order functions which use currying to apply the two arguments independently from each others. The third function shows the possibility of keeping in memory the partial application of the function to which an argument has already been applied. The last function shows the possibility of transforming a simple function into a curried function by calling the method `curried` on it.

</>
Listing 19: Currying example in Scala
</>

```

1  val addition1 = (x:Int,y:Int) => x + y
2  val result1 = addition1(2,3)
3
4  val addition2 = (x:Int) => (y:Int) => x + y
5  val result2 = addition2(2)(3)
6
7  val addition3 = (x:Int) => (y:Int) => x + y
8  val partialAddition = addition3(2)
9  val result3 = partialAddition(3)
10
11 val addition4 = addition1.curried
12 val result4 = addition4(2)(3)

```

### 5.2.3 Monads

Functional programming, put forward by Scala, is a paradigm relying on the principles stated in the precedent section. In this paradigm, the programs are built through the application and the composition of functions. The key feature of functional programming is pure functions. These functions are conceived to have no side effect, no external variable mutation, and therefore a result that will always remain the same according to the input arguments. Following this logic, one goal is to acquire easiness to isolate the developed features, to test them apart from the rest of the program, via formal verification for example, and accordingly to produce fewer bugs.

However, the constraints imposed by functional languages may seem highly restrictive and complicated to use to reproduce patterns that are easy to set up in an imperative language. For example, performing complicated sequences of functions on the same object can be tricky in functional languages due to the prohibition to modify the value of a variable. To prevent such constraints from bringing more problems than advantages, features have been designed to work around them in an elegant way. *Monads* are one of these features.

A monad is a construction for encapsulating an object and performing successive operations on it. In Scala, no class contains predefined logic to build a monad, it is a concept similar to a design pattern that can be set up by the developer. Monads are a very broad subject. Without claiming to explain their potential and functionalities in detail, this section defines their general operating mode. Certain predefined monads are already present in the language. This is the case of the `Option` type which encapsulates an object, or nothing, and then allows the developer to perform operations on its internal content, no matter if the option type contains an object or not.

**Example 5.2.3.** Here is a way to declare two optional variables. The `Option` data structure must be parametrized by the type of its content. The `Option[Double]` declaration indicates that the two variables can optionally contain a value of type `Double`. The `Some` and `None` sub-classes respectively define the presence and absence of a value.

In this case, the variable `num` contains the value 2, and the variable `empty` contains no value.

</>
Listing 20: Option type in practice
</>

```

1  val num:Option[Double] = Some(2)
2  val empty:Option[Double] = None

```

As said before, monads are tools that allow to easily chain operations. If a function is prone to returning sometimes a value, and sometimes not, this function can encapsulate each of its results into an `Option` monad. Without having the need to check what is contained by the monad, the data structure will guarantees to the application that the rest of the operations will run without any problems.

In the following code, a division function has been implemented. To avoid division by 0 but to be able to continue performing other operations without worrying about this issue, the monad provides an elegant way to encapsulate the problem.

The result of the division does not matter for being able to continue with the following multiplication. If a value is present, as in the result of the division between 4 and 2, multiplication is applied by the `map` function. If the result is already empty, it will remain empty for the rest of the following operations, none of which will actually be applied. Such a pattern avoids null checking between each operation.

The `map` function used here modifies the internal content of the data structure.

</>
Listing 21: Option type as a tool for chaining operations
</>

```

1  def divide(x:Double, y:Double) = if (y == 0) None else Some(x/y)
2
3  val div1 = divide(4,2) // Some(2)
4  val div2 = divide(4,0) // None
5
6  val mult1 = div1.map(x => x * 3) // Some(6)
7  val mult2 = div2.map(x => x * 3) // None

```

If the `map` function changes the value contained by the `Option`, it cannot switch from the presence of a value to its absence. Fortunately a `flatMap` function is present for this use case. In the next few lines of code, the goal is to filter the value to keep it only if it is greater than 10. If the mapping was done only with the `map` function, the resulting value would be of type `Option[Option[Double]]`, which would not be useful.

When an operation does not want to modify the value contained by the structure but wants to modify the structure itself, the `flatMap` operator is more appropriate. This operator applies the given function, and then flattens the result. That feature is able to change the effect the monad has. In the following `flatMap` operation, the original result of the previous multiplication is reused. This result being `Some(6)`, which has a value less than 10, the internal structure of the result will change to become `None`.

</>
Listing 22: Monads flattening operation
</>

```

1  val notFlattened:Option[Option[Double]] =
2    mult1.map(x => if (x > 10) Some(x) else None) // Some(None)
3  val flattened:Option[Double] =
4    mult1.flatMap(x => if (x > 10) Some(x) else None) // None

```

Without the `flatMap` operator a type cannot be named a monad, but is only considered to be what is called a *functor*.

#### 5.2.4 Case classes

The concept of immutability is central in a language like Scala, which wants to rely, at least partially, on the functional paradigm. The inclusion of the keyword `val` gives the capacity to easily declare immutable variables. However, support for immutability does not stop there, because the language must also be able to create immutable objects.

For this reason, Scala has introduced a variation of class definitions called *case classes*. The case classes allow to build classes whose parameters are all public and immutable. Each case class depends only on its constructor arguments. They have predefined constructors, which do not require the use of the word `new` at instantiation. They also have predefined equality operators that compare their internal structures and not their references.

**Example 5.2.4.** In this example, both objects are recognized equals because their constructor arguments are the same.

```
</> Listing 23: Case class in Scala </>
1 case class Vehicle(vehicleBrand: String, vehicleType: String)
2
3 val v1 = Vehicle("Mercedes", "Car")
4 val v2 = Vehicle("Mercedes", "Car")
5
6 println(v1 == v2) //true
```

Due to the immutability constraints imposed by case classes, the following statement will not compile. In case classes, a variable cannot be reassigned. This restriction can be bypassed but this is not recommended.

```
</> Listing 24: Wrong re-assignment statement in Scala </>
1 v2.vehicleBrand = "Audi"
```

### 5.2.5 Pattern matching

Another feature offered by Scala is pattern matching. It allows the implementation of more powerful comparison conditions than those present in Java. In Java, switch clauses are a good alternative to avoid the nesting of successive if else statements for executing instructions depending on the value of a variable.

Scala goes further by accepting more complex conditions within these clauses. These conditions integrate particularly well with case classes.

**Example 5.2.5.** It is possible to carry out instructions according to the effective type of each object, while using the values that it contains, as shown in the following example. The `printVehicle` function checks the effective type of each `Vehicle` to print their brands in a customized way.

```
</> Listing 25: Pattern matching on case classes </>
1 abstract class Vehicle(vehicleBrand: String)
2
3 case class Car(brand: String) extends Vehicle(brand)
4 case class Truck(brand: String, load:String) extends Vehicle(brand)
5 case class Moto(brand: String) extends Vehicle(brand)
6
7 def printVehicle(vehicle: Vehicle): Unit = {
8   vehicle match {
9     case Car(brand) => println(s"This is a car from the brand $brand")
10    case Truck(brand, _) => println(s"This is a truck from the brand $brand")
11    case Moto(brand) => println(s"This is a moto from the brand $brand")
12  }
13 }
```

This operation is also applicable to monads if they are constructed with case classes. That gives a fairly wide field of possibilities.

&lt;/&gt;

Listing 26: Pattern matching on monads

&lt;/&gt;

```

1 def printOption(option: Option[Int]): Unit = {
2   option match {
3     case Some(value) => println(s"The contained value is $value")
4     case None => println(s"There is no contained value")
5   }
6 }

```

The Scala language still contains many possibilities and differences with other languages but the essential functionalities to understand the new QRE implementation have now been reviewed.

## 5.3 Language

Only few things should be noted about the LazyQRE language because the way of writing QREs is very similar to that used by StreamQRE. Here are two QREs each counting the number of vehicles observed and producing exactly the same result. The first one is written with LazyQRE, while the second one is written with StreamQRE.

&lt;/&gt;

Listing 27: Extract of code with LazyQRE

&lt;/&gt;

```

1 var isVehicleSpeedToken = new AtomQRE<StreamingDto,Double>(x ->
2   ↳ Objects.equals(x.getItemType(),VEHICLE), x -> x.getSpeed());
3 var isSpeedSum = new IterQRE<>(isVehicleSpeedToken, 0D, Double::sum, x -> x);

```

&lt;/&gt;

Listing 28: Extract of code with StreamQRE

&lt;/&gt;

```

1 var isVehicleSpeedToken = new QREAtomic<StreamingDto,Double>(x ->
2   ↳ Objects.equals(x.getItemType(),VEHICLE), x -> x.getSpeed());
3 var isSpeedSum = new QREIter<>(isVehicleSpeedToken, 0D, Double::sum, x -> x);

```

In both queries, the `isSpeedSum` combinators add all the speed values received from their inner combinators with the `Double::sum` function, starting with the value 0 in the `Double` number format. The result is produced without any modification by the lambda function `x -> x`. The only filter performed is in the inner combinator, which limits the domain of the query to `VEHICLE` type objects via the function `getItemType`.

As can be seen the way of writing them is almost identical. This is an advantage that makes it easy to switch from StreamQRE to LazyQRE, and vice versa. There are many differences between StreamQRE and LazyQRE, but they reside in the implementation of the language.

## 5.4 Compilation

In the compilation of StreamQRE, the impact of the complexity bounds of regular functions have been transposed to the combinators implementation. These bounds induce the use of stateful combinators, which are split and iter. These combinators must keep intermediate states in memory to be able to perform future calculations. Iter must always keep the results of the current iteration, but also of the previous one. As for the split combinator, it must keep the states necessary for the concatenations susceptible to arrive in the future.

LazyQRE respects the same operating principles but will delay the execution of operations until they are needed or requested.

In the execution of an algorithm built with StreamQRE, at the arrival of each new element, the operations in which this element is involved are directly executed. However, these operations may have been performed to calculate an intermediate state which will never be used in the future.

**Example 5.4.1.** In Section 4.3.2, an example was explained to demonstrate the need for the split combinator to keep partial results. The purpose of the presented algorithm was to aggregate the average speed of the last ten vehicles which were observed on a road before the occurrence of an accident.

Coming back to this algorithm, it is possible to notice that many executed operations could have been avoided. Figure 5.1 shows a state of the stream in which the algorithm had already observed an accident in the data received and was waiting to see if another would occur. The whole query was explained in Section 4.3.2. The parts of this query, defined by the combinators `is10ItemsAvgSpeedBeforeAccident` and `isVehicleSpeedSum`, were partitioning the stream between the last accident observed and the vehicles that have been processed after.

On the image, the blue squares represent the two series of vehicles whose speeds have already been aggregated to extract the average speed observed.

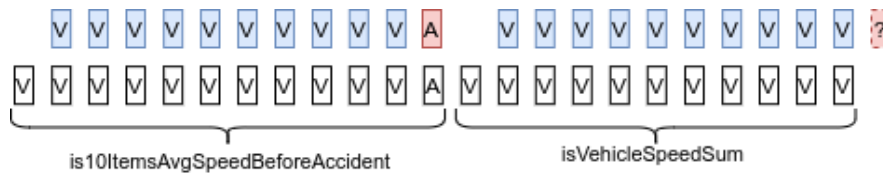


Figure 5.1: State of the stream when an accident have already occurred, but another can happen anytime soon

The first series of vehicle has been aggregated with good reason because an accident has indeed occurred subsequently. However, this is not the case for the following one. The second series of blue squares represents a set of vehicles whose speed was aggregated without knowing if an accident was going to occur or not. If the next item received does not reflect the occurrence of an accident, this operation will have been carried out without any real interest. Since the pattern used by the current combinator is a sliding window, this operation will be computed unnecessarily at the arrival of each new element which is not an accident.

The impact it has on program performance is minimal in this case. Here, the window combinator operation is inexpensive in terms of resources for the CPU since it is a simple sum. However, this will not always be the case for all operations. For example, in the network monitoring domain, which had been reviewed in Section 1.7.3 of the first chapter, encryption and decryption operations were defined as expensive operations in terms of resources, and were especially used in SSL renegotiation attacks for this purpose.

**Example 5.4.2.** To check the effective impact of the obligation of encrypting or decrypting each data received by the streaming processing algorithm, a new simulation was carried out. The QRE designed in Section 4.3.2 to monitor the occurrence of accidents on the road has been modified, and an intermediate operation have been set up.

The following apply combinator has been included. Its purpose is to simulate the encryption or decryption of each vehicle received before sending back the speed so that the algorithm can continue its average aggregation. The code of the encryption simulation is attached in Appendix C.2.1. The complete query will not be rewritten here but can be found in Appendix C.2.2.

</>
**Listing 29:** *Combinator applying a simulation of a vehicle speed encryption*
</>

```

1  var isVehicleSpeedToken = new QReApply<>(isVehicleToken, x -> {
2      GenerateEncryptionSimulation.generate();
3      return x.getSpeed().doubleValue();
4  });

```

Figure 5.2 represents the performance analysis of the algorithm over a period of 10 minutes, on the processing of a stream which contains no accident token. Since the algorithm calculates all the intermediate operations automatically, it is obliged to go through the encryption step for each vehicle. The performance requirements are blowing up, and the demand for CPU resources is constantly around 40%, while no end results should be produced.

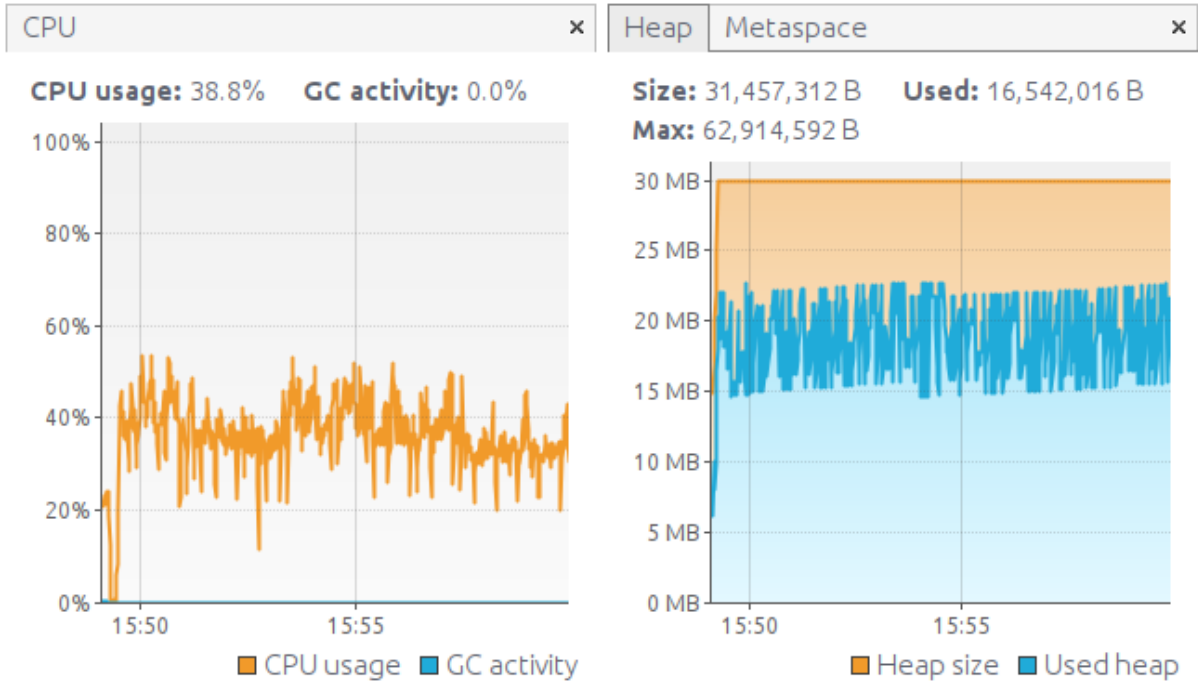


Figure 5.2: Performance of the StreamQRE solution with encrypted vehicles

To avoid this kind of problem, in the implementation of LazyQRE, *the choice was made to evaluate the operations only when a result is requested*. Even if an end result can be produced, all the operations to compute it are kept in memory without evaluating them. This feature, called *lazy evaluation*, being the key feature within the behavior of the LazyQRE implementation gave it its name.

Not only does this avoid the computation of some useless intermediate states, but it also leaves the user, or the developer, to choose if he wishes to extract the current result or not. In the case of accident monitoring solution, a developer could decide to evaluate the stream only at intervals of 5 or 10 minutes to retrieve the information. With a LazyQRE implementation, only at these times the operations would be effectively evaluated.

To find out how lazy evaluation of the constructed terms was possible, it will be required to deep dive into the implementation of the language.

## 5.5 Implementation

### 5.5.1 Structure

LazyQRE is entirely written in Scala. As for StreamQRE, combinators have been reified into object instances. However, the internal structure of the combinators works differently. In StreamQRE, each combinator contained a stack which was mutable and for which the values encapsulated were modified, added, or removed from the stack, when reading each new element in the stream. In this implementation, each object instance encapsulating the combinator logic is immutable. For this reason, they have all been implemented using case classes. When a combinator is modified, none of its variables change but it returns a new version of itself by creating a new object. Then, the previous combinator instance, which had become useless, will be deleted from memory by the garbage collector.

The advantage of this method is that there is no need to worry about what state each object contains, or to avoid any undesired effect caused by accessing data at the wrong time. Thus the risk of creating an object which could be ending in an invalid state is close to zero. Immutability also allows a good encapsulation, which guarantees the absence of side effects. In addition, in terms of ease of development, the implementation is easy to maintain. All classes have been written in less than 50 lines of code.

The major drawback could be the heavy use of memory. With such practices, the number of combinators



quickly becomes very large. High production of new combinators uses tons of memory. In fact, this is a recurring problem in functional programming. Since the variables cannot be changed, new ones are created all the time. Despite the fact that the garbage collector frees memory regularly, it is preferable to try to minimize memory overhead. In this case, to minimize as much as possible the memory occupied by each combinator, it was necessary to build an adequate structure. The combinator structure was inspired by the flyweight design pattern.

The flyweight pattern is useful when manipulating a large number of objects that have some properties in common. The goal is to centralize these properties in a single data structure to avoid deduplicating them in all objects. Factorization separates the identical properties across all objects from the specific ones, externalize them into a data structure, and share them between the instances of the different objects.

With this pattern being applied to the combinators, their state has been divided into two classes. An intrinsic state that is invariant, and which contains all the parameters required for configuring the behavior of the combinator. These are generally the operations to be applied to the elements of the stream. The second class is an extrinsic state that is variant, which contains the result, or the partial result, produced by the combinator. This class has a reference to its intrinsic state. For each combinator, only one intrinsic state is maintained, while new extrinsic states are created as the stream is read.

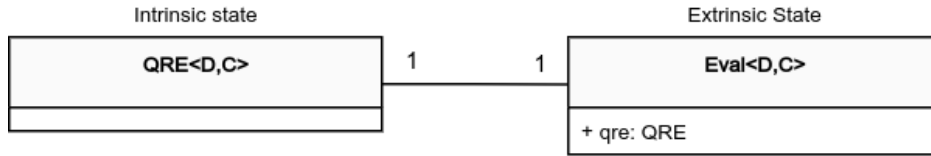


Figure 5.3: Partial class diagram of a combinator, separating its logic between two classes

Figure 5.3 represents the separation of the abstract structure of a combinator into the two classes, QRE and Eval. In Figure 5.4, the corresponding object diagram shows the unique link from the invariant state to its variant state. Despite that other variant states have been linked to it previously, they will no longer be used and are no longer referenced. The previous variant states are colored in red. The intrinsic state class is named QRE because of its role in the configuration of the query. The extrinsic state class is named Eval because of the evaluation logic it contains.

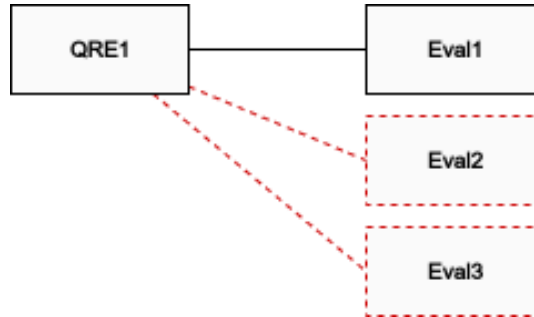


Figure 5.4: Partial object diagram of a combinator, separating its logic between two classes

### 5.5.2 Parameter types

In Figure 5.3, the classes have been parameterized with two types, D and C. These are analogous to the domain and the cost types used to define the logical properties of QREs in Chapter 3. The items read on the stream belongs to the type D and the processing produces a result of type C.

**Example 5.5.1.** Considering a flow of vehicles for which each item must be mapped to the value 1, relying on these types, the operation expressed by an atom combinator could be the following,

$$x \rightarrow 1 : D \rightarrow C$$

However, with such an operation, each element  $x$  on the stream would be evaluated instantaneously, and for each object received, the performance problem exposed in the previous section would come back. For



this reason, it was decided to use the features of Scala to set up the lazy evaluation. The operations are mapped as higher-order functions, and return new functions for the quantitative properties to be evaluated later. According to that principle, the previous operation is transformed in:

$$x \rightarrow (() \rightarrow 1) : D \rightarrow (() \rightarrow C)$$

The evaluation of the operation will not be executed when the element  $x$  is read, but the function  $() \rightarrow 1$  is kept in memory and can be executed if necessary to retrieve the result. The structure of this operation is valid for a single atom combinator but does not allow the creation of more complex operations. Up to now, the principles which makes it possible to compose the combinators with other combinators has not been approached yet. These principles are also based on the use of higher-order functions.

**Example 5.5.2.** For example, the goal of a stream processing algorithm could be to count the number of vehicles in a stream. In this case, an iter combinator should gathered the operations emitted while reading every streaming item and compose a new function with them. Let's imagine the function produced by this iter combinator using, as child, the atom combinator defined previously.

Considering that a stream  $w$  of three vehicles has been read, such that  $w = v, v, v$ , the result produced by the iter combinator after reading the third vehicle  $x$  should be the following:

$$x \rightarrow (() \rightarrow (((() \rightarrow 1)() + (() \rightarrow 1)() + (() \rightarrow 1)())) : D \rightarrow (() \rightarrow C)$$

This term can be decomposed as a tree to understand it more easily. This is done in Figure 5.5. Once again, this decomposition shows that the operation will not be evaluated when reading the streaming element  $x$ , but will compose a new function which will be kept in memory. When evaluating this resulting function, all the operations nested one inside the other will be executed.

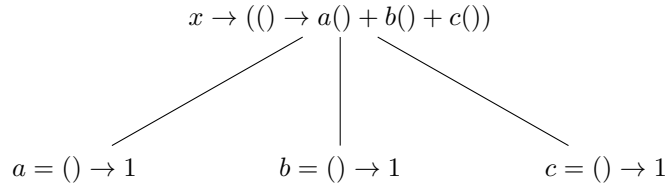


Figure 5.5: LazyQRE iter operation as a tree

The principles demonstrated in Section 5.2.2 regarding higher-order functions give the necessary expressiveness to design such functions. However, the domain and cost types,  $D$  and  $C$ , are not sufficient to assemble the functions together while ensuring the compatibility of the data types used. For example, if the iter counting operation expects to receive data of type `Integer`, it is necessary that the definition of its child combinator guarantees this type of data as output. Therefore, each parent combinator class must be parametrized by the type of data it expects to receive from its children. This guarantee will also help during the development process of the queries. The syntactic analysis, included in the programming language compilers and in some development tools (IDE), will ensure that the combinators are compatible during the creation of a QRE.

In Figure 5.6, a minimal definition of the classes specifying the iter and the atom combinators are represented. Two type parameters are highlighted. These parameters makes it possible to match the type of data received by a parent combinator from its child combinator. The output type of the atom combinator,  $C$ , must be the same than the type  $ChildC$  of the iter combinator for the two combinators to be nested together.

This type parameterization is not only one required for the implementation of LazyQRE. To create the term resulting from the evaluation of the stream, the partial functions must be kept in memory. In a first approach, it may seem sufficient to keep each partial function within the parent combinator and wait for the result of its children to complete it. Taking back the preceding iter example, here is the construction of such a function that could be kept between the arrival of the second and the third element of the stream:

$$(childoutput) \rightarrow (() \rightarrow a() + b() + childoutput)$$

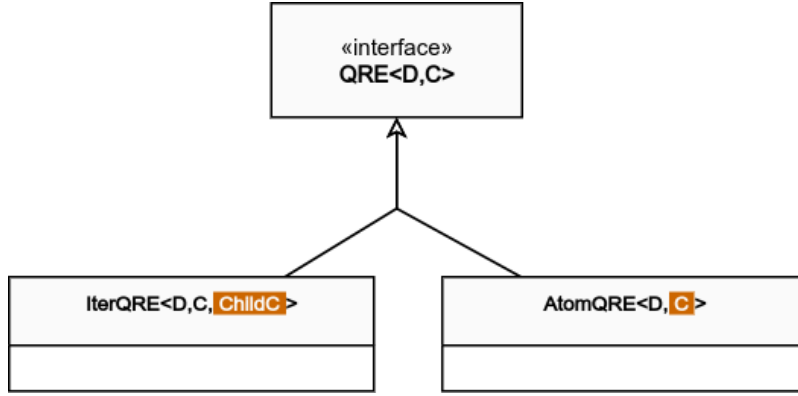


Figure 5.6: Class diagram defining the typing correspondence constraint between parent and child combinators

Note that to support non-commutative operations, the function can also be written in this way

$$(childoutput) \rightarrow ((\rightarrow childoutput + b()) + a())$$

However, the same combinator is required to be capable of keeping an indefinite number of partial functions. This is due to split combinator, as it was demonstrated in Section 4.3.2. To solve this problem, StreamQRE moved the stacks containing the partial results from combinator to combinator when new elements arrived. In the case of LazyQRE, it is the partial function that will be moved between the combinators.

In the case of the preceding iter combinator, its evaluation will create a partial function which will be passed as an argument to the atom combinator. This function expects the arrival of a new element  $x$ , then create a function  $c$ , which will compose the rest of the result. This is how it can be written:

$$x \rightarrow (c \rightarrow ((\rightarrow a()) + b()) + c())$$

The type of each function created by a parent combinator must be known by the child combinator to ensure that function composition can be performed during its evaluation. This is why an other type parameterization should be added to the combinator classes. Hopefully, thanks to type inference and to the way the combinator classes have been defined, the developer is not required to parameterize this last type by himself.

To summarize all these properties, in Figure 5.7, a diagram of the iter combinators structure is represented.

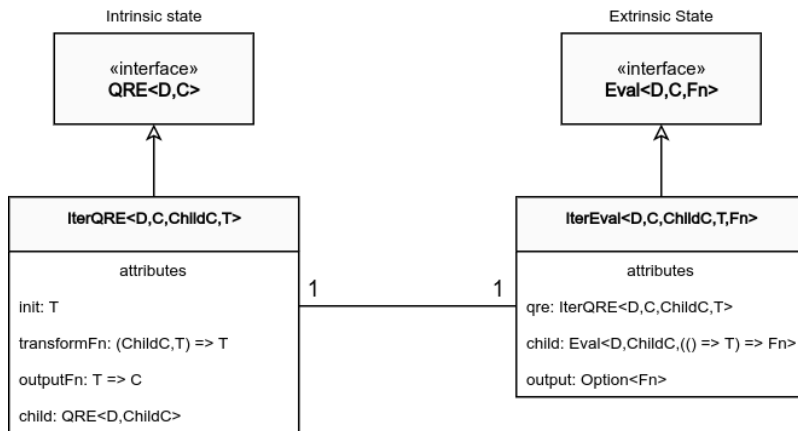


Figure 5.7: Class diagram of an iter combinator

In the IterEval class on the class diagram, the parameterization of the partial function, which should

be passed to the child combinator, can be found in the declaration of the child Eval class. This type is declared as  $((() \rightarrow T) \rightarrow Fn$ .

The output type of the child can also be found in the parameter types of the IterQRE and IterEval classes. This type is declared as *ChildC*.

Certain types of combinators may require the parameterization of some intermediate types. This is the case for the iter combinator which allows to parameterize the type of the values aggregated by its operations. This type is declared as *T*.

In comparison, here is the declaration of an iter combinator, specified to solve the preceding example problem, written with LazyQRE:

**Listing 30:** *Example of a simple QRE for explaining the LazyQRE combinator structure*

```
1 var init = 0;
2 var transformFn = (x,y) -> x + y;
3 var outputFn = x -> x;
4 var child = new AtomQRE<StreamingDto,Integer>(x -> true, x -> 1);
5 var isSpeedSum = new IterQRE<>(child, init, transformFn, outputFn);
```

It fits the structure defined by the class diagram. The parametrized types should only be specified in the atom combinator. The type inference system gives the opportunity to omit them in the parent combinator. Note that only the intrinsic state is configured in the declaration. The Eval part will be created automatically by the QRE class at the start of the evaluation.

### 5.5.3 Result

In Figure 5.7, in the IterEval class, the attribute containing the current result produced by the combinator is declared as *output*. The creation of results uses monads and pattern matching to handle the undefined values, as well as to handle the stream subsets that would not match the QRE domain.

When a parent combinator receives an output from a child combinator, this output is always encapsulated in an Option monad. In this way, the combinator can easily handle undefined outputs using pattern matching. The following extract of code shows how it's done.

**Listing 31:** *Monads integration for combinator results*

```
1 child.output match {
2   case Some => ...
3   case None => ...
4 }
```

The whole implementation of the structure and type parameters of combinators have now been defined. Every combinator type implementation will follow exactly the same model as the iter classes. It remains to define the methods which make it possible to execute the algorithm and to consume the elements of the stream.

### 5.5.4 Methods

The class diagram of the iter combinator presented previously can be completed with its available methods. Figure 5.8 displays it.

The configuration methods are set in the QRE class:

- **create:** Create the initial Eval object instance to start the algorithm. The Fn type is used to define the return type expected by the combinator from its child. This type is parameterized by the create method, and not at class level, because it makes it possible to change the expected type of the result according to the nesting of the combinators.

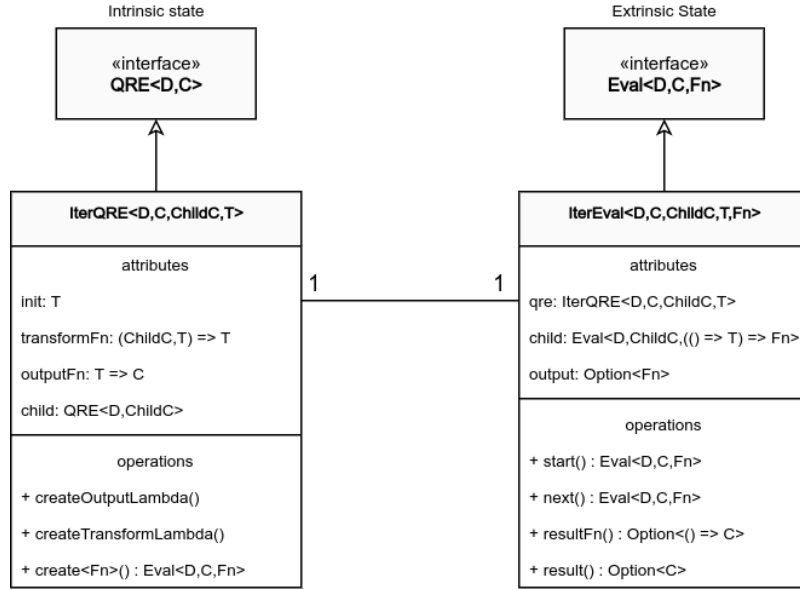


Figure 5.8: Class diagram of an iter combinator with its operations

- `createOutputLambda`, `createTransformLambda`, etc.: These functions vary depending on the type of the combinator. They are implemented to create the necessary higher-order functions for the evaluation of the stream.

The evaluation methods are set in the `Eval` class. These methods are inspired by the same model as `StreamQRE`:

- `start`: Initializes the internal structure for the combinator evaluation. This method is called at the launch of the evaluation of the algorithm.
- `next`: Consume the data items. On receipt of each new item, the method evaluates the stream on the regular pattern represented by the combinator. It keeps the result if the concatenation of the previous items with the one newly received is recognized. Note that it will return a new version of the `Eval` class with a new version of the output variable each time, and will not mutate any variables in the current class instance.
- `resultFn`: Returns the result function without evaluating the output. This function is useful to check if there is an actual output, or if the result is currently undefined, without evaluating any operation on the stream.
- `result`: Returns the eager result by evaluating of the output function.

### 5.5.5 Advantages

This section aims to review the different advantages of the `LazyQRE` implementation.

In the previous example, Example 5.4.2, an algorithm aggregating the speed of the observed vehicles had been supplemented by the need to encrypt or decrypt each speed before having the capacity to use it. This algorithm, built with `StreamQRE`, had then shown a significant demand of resources from the CPU monitoring. This demand was not without reasons. All operations were evaluated even if they did not necessarily lead to a final result.

This algorithm has been rebuilt, in the exact same way, but with the `LazyQRE` implementation. Figure 5.9 shows the performance report that was observed while running this new algorithm. The entire query will not be rewritten here, but it can be viewed in Appendix C.2.3.

During this observation period no accident occurred. Therefore, thanks to lazy evaluation, the algorithm was not required to evaluate any element of the stream. Unlike the algorithm built with `StreamQRE`, the resource consumption that was observed from the CPU remained very low.

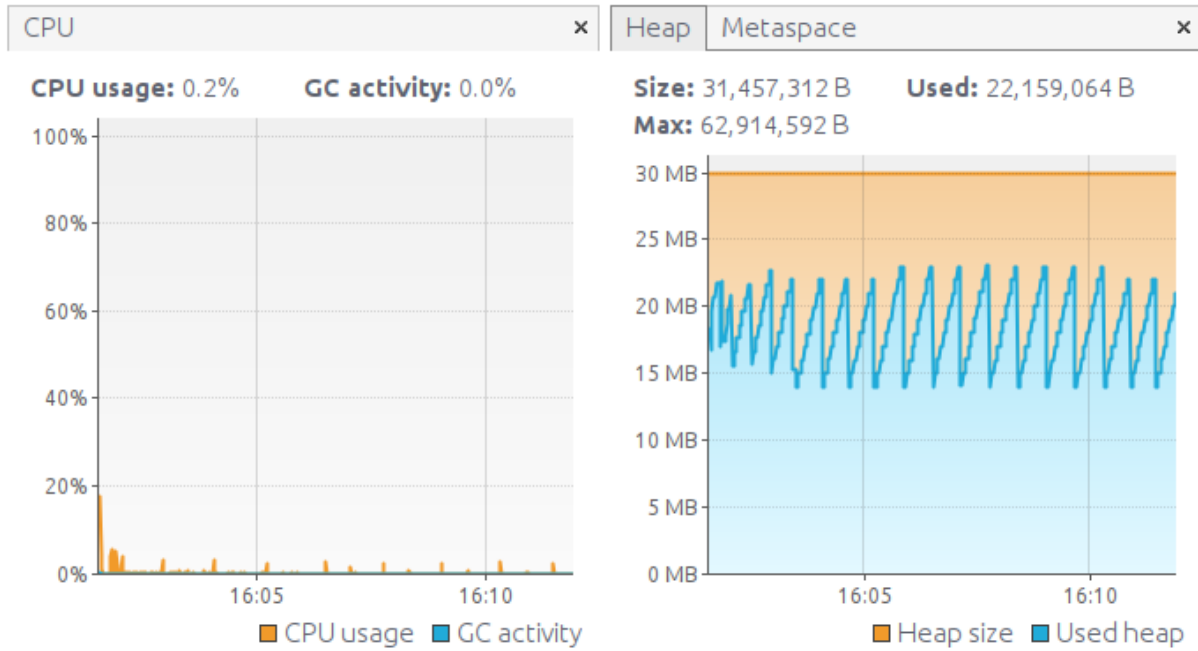


Figure 5.9: Performance of the LazyQRE solution with encrypted vehicles data, no accident detected

Another test was carried out, still over a period of 10 minutes, with the same algorithm. In this one, accidents were simulated every 30 seconds. Figure 5.10 shows the performance report that was observed while running the algorithm on this simulation. Although these accidents occur at regular intervals, as long as no result is requested, no operation is performed. That is another advantage of lazy evaluation. In this simulation, only one result was requested at 13:35, the graphs translate it by showing an increase in resource consumption at that time. All the rest of the time, the resource consumption remained stable.

Within this simulation, the result was evaluated when it was requested. However, nothing prevents the user to get the resulting function, using the method `resultFn`, and to evaluate it later. Such practices can be useful when the algorithm runs on a system that is known to have resource-intensive operations at scheduled periods.

Another advantage of LazyQRE compared to StreamQRE is the type parameter system. In StreamQRE, the stacks, as well as every result, are typed as `Object`. `Object` is the root of the class hierarchy in Java. That thus obliges the user to cast the result in its effective type, which can cause errors. By using LazyQRE, the result is automatically typed according to the operations executed to obtain it. This prevents the user from making mistakes about future operations on the result.

### 5.5.6 Disadvantages

This section aims to review the different disadvantages of the LazyQRE implementation.

The composition of functions kept in memory, and the lazy evaluation, does not only have advantages. The memory consumed depends on the size of the query. So if the domain of an QRE is very large, for example the aggregation of a very large number of items, the memory consumed by the algorithm can quickly become quite high. Since the evaluation of the operations is delayed, the algorithms will consume more memory than with StreamQRE.

To avoid this problem a maximum iteration count has been added to the iter combinator. If this number of successive iterations is reached during the evaluation, all the previous iterations are automatically executed and aggregated. This number is optional, and can be configured in the declaration of the combinator. As long as this parameter is not set, the iterations will never be automatically aggregated.

**Example 5.5.3.** For example, the following algorithm aims to count all the vehicles observed in a data stream indefinitely. Without aggregating the data automatically, the number of compound operations would only grow and would end up taking all the space in memory. To avoid this situation, the declaration

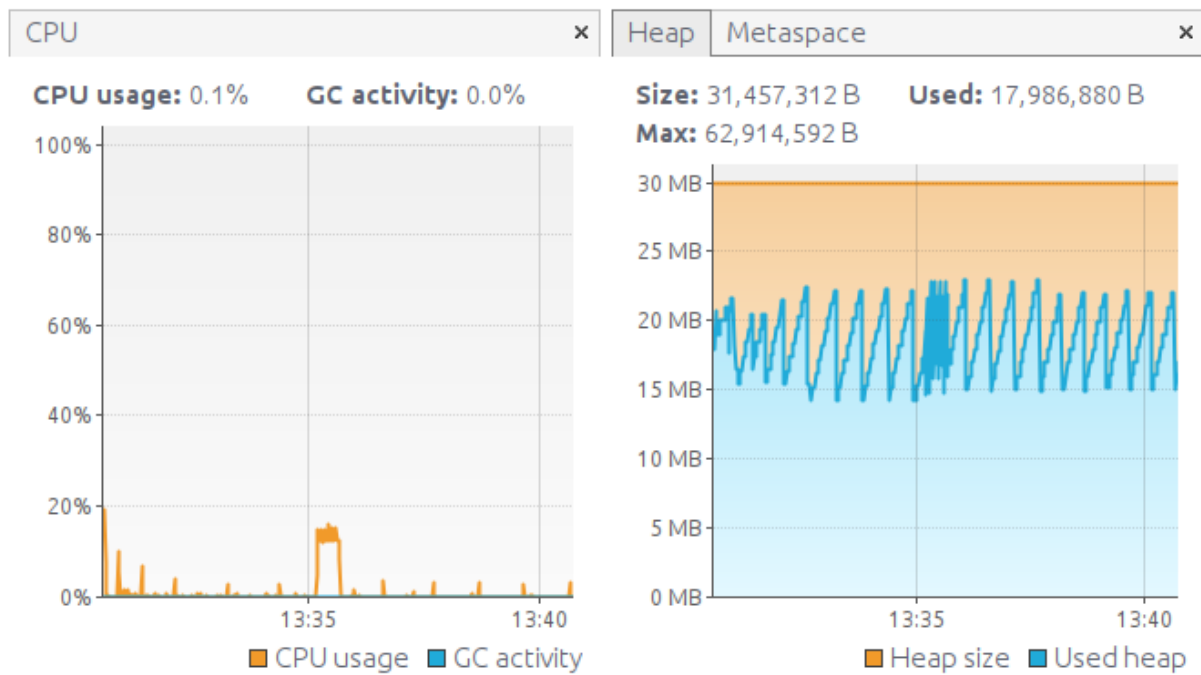


Figure 5.10: Performance of the LazyQRE solution with encrypted vehicles data, an accident detected

of the iter combinator forces the automatic execution of the operations each times 1000 iterations have been reached.

**Listing 32:** *Example of an iter combinator automatically aggregating the results after 1000 iterations*

```
1 var isVehicleToken = new AtomQRE<StreamingDto,VehicleDto>(x -> true, x -> 1);
2 var isVehicleCount= new IterQRE<>(isVehicleToken, 0D, (x,y) -> x + y, x -> x, 1000);
```

Note that, by following this logic, configuring the iter combinator with a maximum of iterations set to 1 is equivalent to reproducing the behavior of its counterpart in StreamQRE.

**Listing 33:** *Example of an iter combinator automatically aggregating the results after 1 iteration*

```
1 var isVehicleCount= new IterQRE<>(isVehicleToken, 0D, (x,y) -> x + y, x -> x, 1);
```

Another problem that can be caused by the creation of queries having a large domain, and aggregating a lot of items, is the nesting of too many operations. If too many operations are nested, when they will be evaluated, the Java Virtual Machine will throw a stack overflow exception because the amount of call stack memory allocated is exceeded. This number of operations depends on the configuration of the JVM. Limiting the number of iterations within this iter combinator is also a solution in this case.

Another solution, which is not currently implemented, and which could be a way of improving LazyQRE, is the use of tail call recursion implementation for the iter combinator. By using tail call for combining the operations, an indefinite number of operation could be nested [45]. The tail call optimization gives the capacity of calling a nested function without adding a new stack frame to the call stack. The tail call must be the last instruction of a function, and because it is, it frees or replaces the memory space reserved for the current function on the stack.

An additional solution could be to use variables to retrieve intermediate results when performing a series of operations before reapplying them to another series of nested functions. Saving intermediate results would prevent the nesting of too many functions.

Another disadvantage can be observed, although it is more of a suggested improvement than a real problem. Generally, the terms constructed by function composition contain at least one concatenation, using `split`, which divides them in two parts. When a new element of the stream arrives, it is likely that only one part, left or right, will change. However, when evaluating the whole function, the unchanged part will still be evaluated a second time even if the same result was already computed at the previous state. One way of improving the solution would be to reuse the results of the different parts for future streaming iterations, if they were already computed before. Note that this may add an additional cost in memory depending on the domain of the results.

A final disadvantage is the size of the LazyQRE package, which is around 150kb, compared to the size of the StreamQRE package, which is around 80kb, thus practically twice as small. However this must be put in perspective compared to the average size of binaries. In comparison, the Java application package used for the simulation testing of the algorithms present in this thesis weighs 28mb. Comparatively, the two packages containing the QREs implementations both remain very light.

### 5.5.7 Example

In Section 4.3.4, a step-by-step algorithm demonstration was detailed and showed the behaviour of the combinators while evaluating a QRE using StreamQRE. The example query was designed to count the number of vehicles behind a sequence of two trucks.

In this section, an identical demonstration is detailed but this time using LazyQRE. The sample query is configured exactly in the same way as the StreamQRE sample query, and is evaluated on the same word. This word is shown in Figure 5.11.

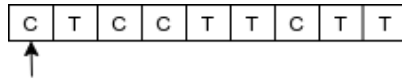


Figure 5.11: The word to parse starting at the left.

Figures 5.12 and 5.13 show the demonstration. Unlike StreamQRE, in which stacks were moved from a combinator to another, in this case, partial functions take their place. The nested calls of functions between them can easily be observed. At each step, new functions are created and call the functions from the previous steps.

Regarding stream recognition, the behavior of this implementation is exactly the same as for StreamQRE. Likewise, the results are also identical.

This demonstration includes the presence of a `split` combinator. The higher-order functions created by this combinator are similar to those created by an `iter` combinator, except that they will not be combined over and over again, but will always be created on two levels. Since the `split` combinator express a concatenation, their first evaluation returns a new function with the left split argument set. Then, on the evaluation of this new function, the `split` function is returned with both its arguments set.

The analysis of the LazyQRE implementation is now completed.

## 5.6 Road traffic monitoring use case: A LazyQRE solution

In Section 1.4, an implementation of a naive algorithm was built in order to estimate the maximum density of vehicles acceptable by a tunnel. In Section 4.5, this algorithm was rebuilt using the StreamQRE implementation.

A new implementation has been developed with LazyQRE. The whole implementation of the tested algorithm is not listed here, but has been attached in Appendix C.1.4. In the context of LazyQRE, the differences with the algorithm built using StreamQRE are minimal. As it was shown in the previous sections, the language used by LazyQRE is very similar to that of StreamQRE. The declaration of the algorithm, its modifiability and testability are therefore practically identical.

Given that this algorithm does not require significant resource consumption, both for memory and algorithm complexity, the performances observed are very similar to those of StreamQRE in this case. Figure 5.14 shows a performance report on a test identical to that performed for the other two solutions.

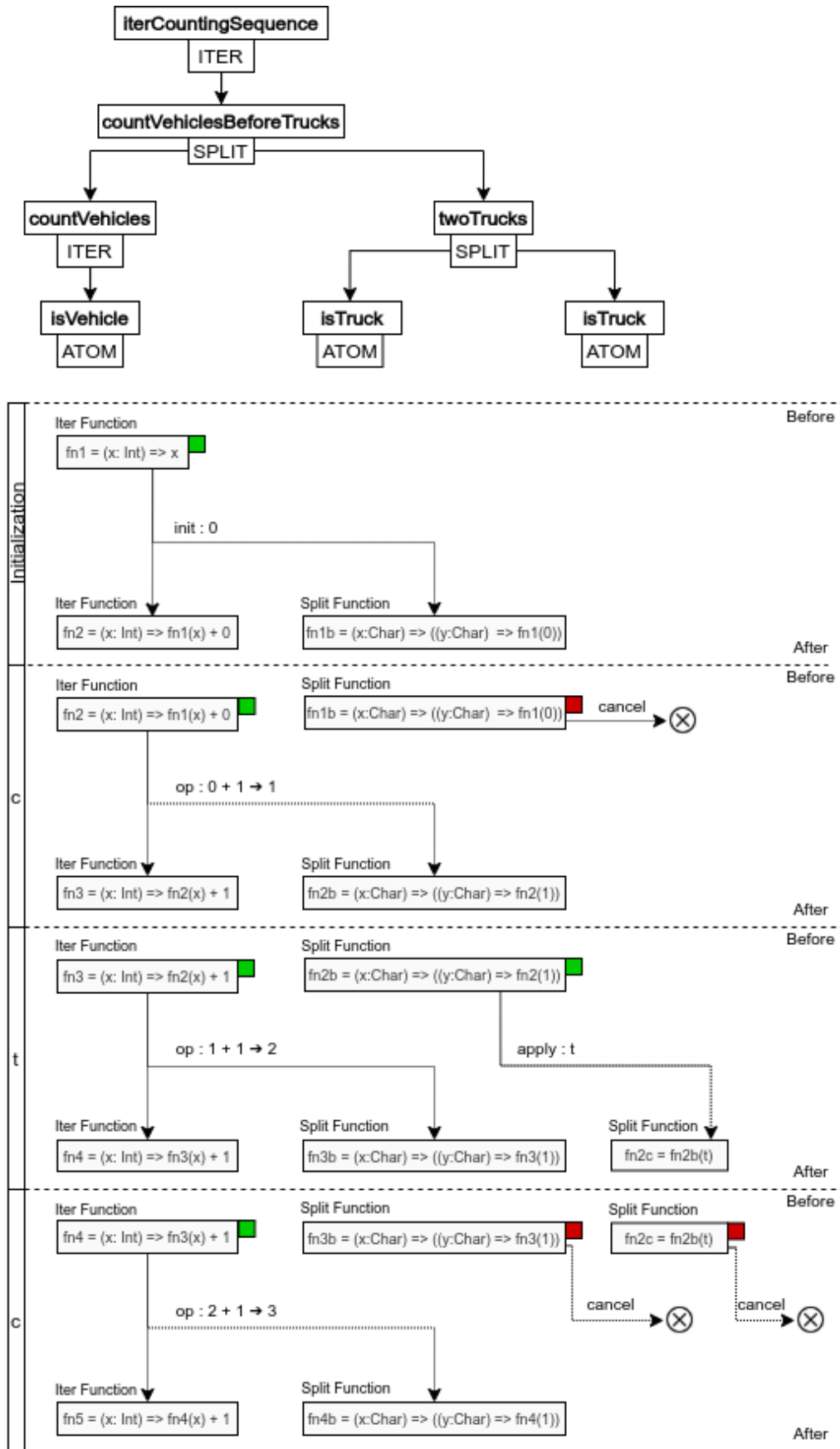


Figure 5.12: First part of the LazyQRE demonstration



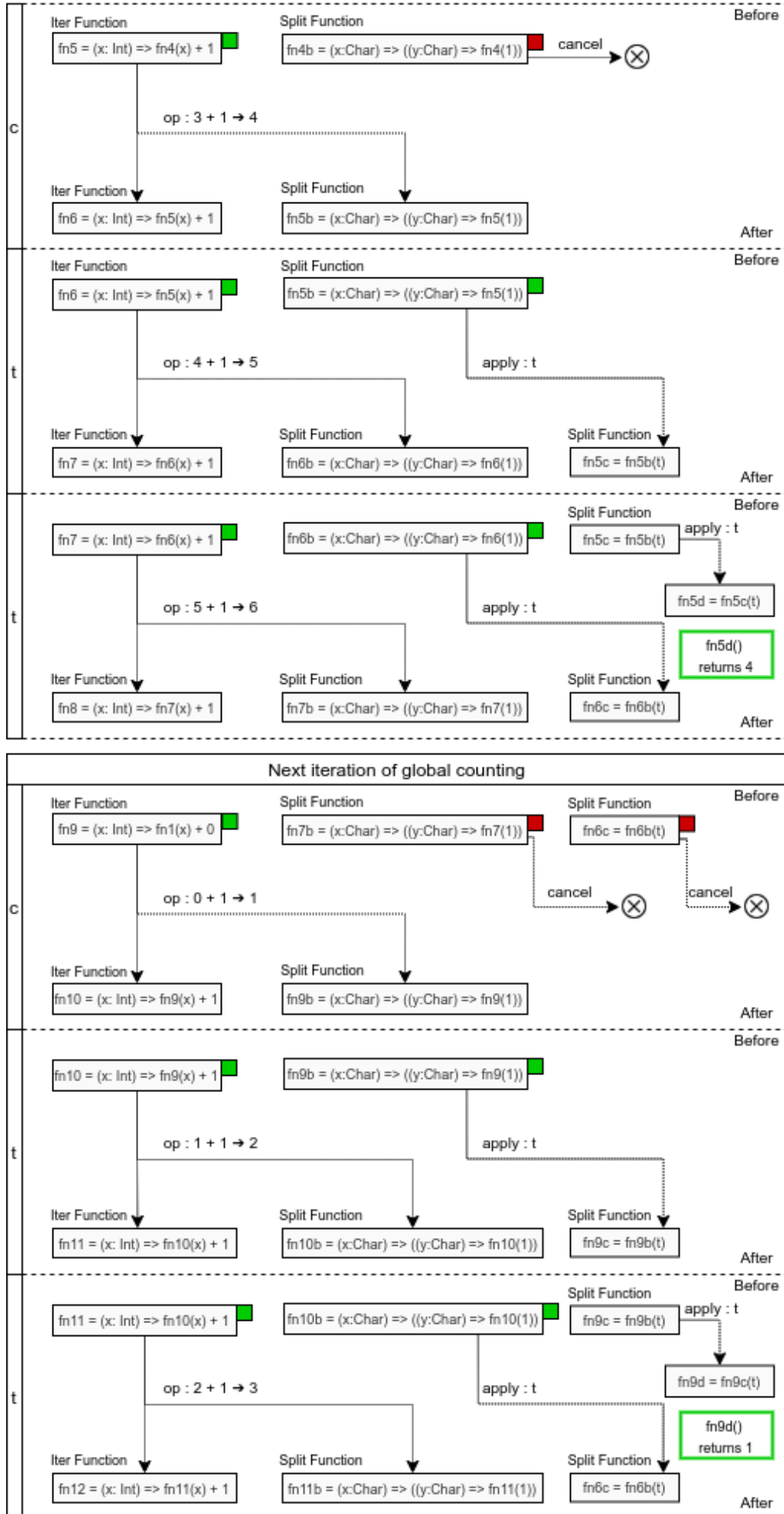


Figure 5.13: Second part of the LazyQRE demonstration

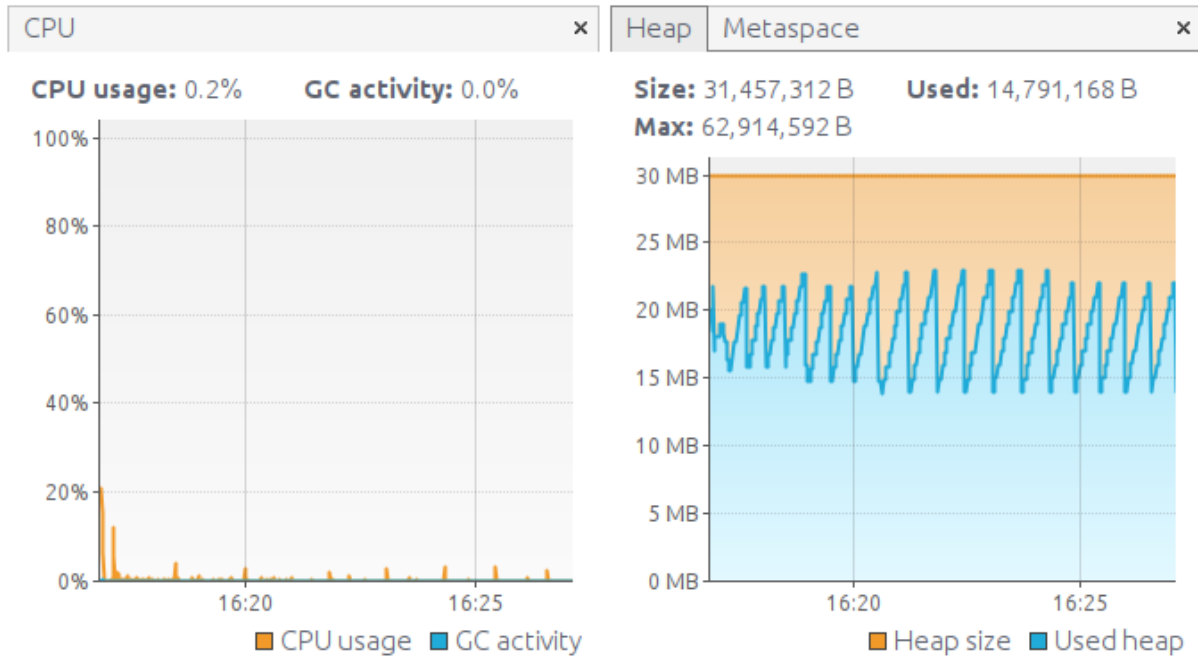


Figure 5.14: Performance of the StreamQRE solution

To perform all the tests that were presented in this thesis, a simulation application had to be developed. The technical specifications of this application have been explained in Section 1.5. The frontend of this application built to configure each of these tests for the different implementations of traffic monitoring algorithm, including this one, is displayed in Figure 5.15.

The tests were carried out on the largest tunnel in Europe, the l erdal tunnel. In this image, the green, black, and red dots represent motorcycles, cars, and trucks, respectively. All the parameters displayed on the left make it possible to control the density of the simulated traffic, but also the characteristics of the tunnel such as the length of the slope and its percentage.

For each type of vehicle, its density can be defined as well as its average speed. In the tests, the distribution was generally divided between 10% of motorcycle, 20% of trucks and 70% of cars. The average speed of each vehicle type is randomly generated around a configurable average speed and a configurable standard deviation. In this way, the speeds generated for motorcycles will generally be higher than those for cars or trucks.

## 5.7 Conclusion

To conclude, this chapter has presented in its entirety LazyQRE, a new incarnation of quantitative regular expressions written in Scala.

Scala was put forward as a useful language for a new implementation. Thanks to functional programming structures, which are not available in some other languages, it gives the capacity to achieve different logical operations. At the heart of these new features are higher-order functions and function composition.

Using higher-order functions, LazyQRE aims to create terms based on the theory defined by the QREs while observing a data stream. These operations are constructed in a similar way to those produced by StreamQRE, with the main difference that they are not evaluated immediately. This practice is called lazy evaluation, the terms are built by a composition of functions and are kept in memory to be evaluated when necessary. This allows significant performance gains by considering that not all operations always need to be evaluated. In addition, it gives the capacity the user, or the developer, to choose when they want to run them.

This implementation also comes with some tradeoffs in return for what it offers. In particular, it requires a bit more memory to store the computed operations, and the implementation is limited in terms of the

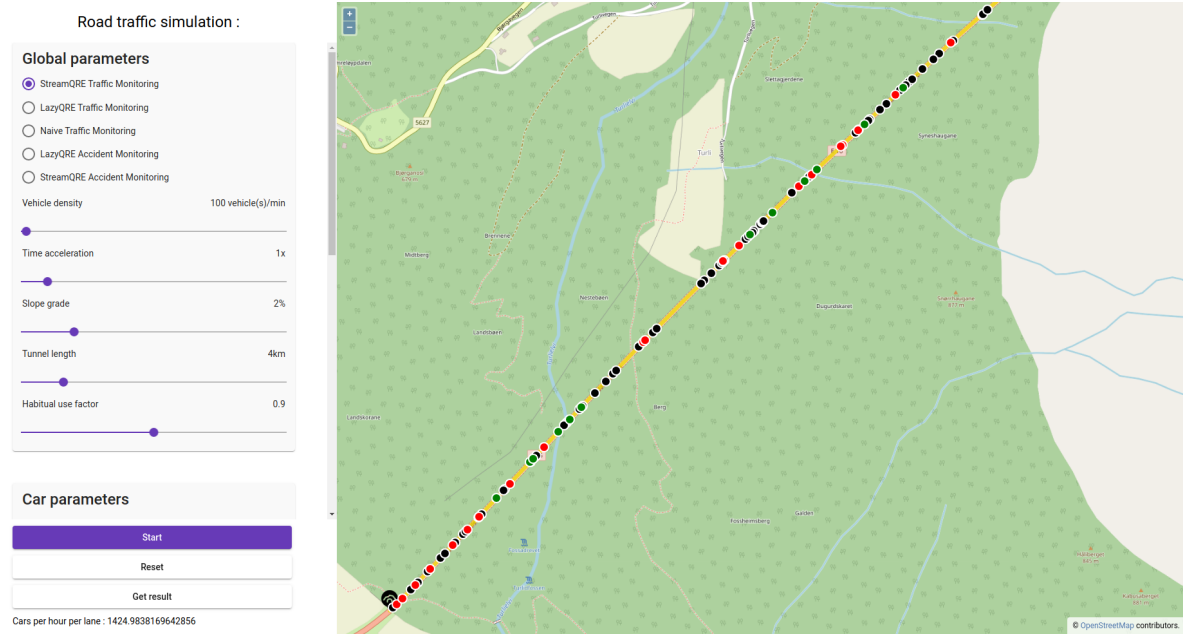


Figure 5.15: Application that was built to do every simulation testing

number of nested functions which can be lazy evaluated.



# Conclusion

This thesis presents techniques for processing data streams to calculate quantitative properties. The declaration of queries specifying the data to be aggregated, and the operations to be performed on that data, must be concise and flexible. In addition, the properties are intended to be computed in an efficient manner. By combining these two qualities, people who are not experts in the field of data stream processing can easily apply efficient streaming algorithms within their field of application. That's why such a language is required to be small and flexible to easily adapt to different use cases, and be interfaced with any type of application, even existing ones.

By studying low to high level streaming languages within the scope defined by these constraints, research has shown that a language combining performance, expressiveness and flexibility is complicated to find. Low level languages lack flexibility and are difficult to write without possessing a profound knowledge in computer science. On the other hand, high-level languages don't guarantee predictable performance, or are not efficient enough.

Nonetheless, a language called Quantitative Regular Expressions (QREs) stands out. Through multiple examples, this language is presented and analyzed from bottom to top. These investigations give the necessary background to understand what makes it so appealing. The QREs have the advantage of being of a high level language. That ensures them the ability to be easy to learn and apply. At the same time, the designed queries rely on formal language theory, and more precisely on a new type of automaton called cost register automata. This finite state machine was created in order to apply a function to a term in accordance with regular language properties to guarantee good performance. Applied to a data stream, this technique has proven to be plainly adequate for efficiently aggregating quantitative properties.

The restrictions imposed by this language are put in perspective to the performance it can provide. While maintaining the theoretical properties of state machines, logical components, called combinators, have been designed with the objective of developing implementations in general purpose programming languages. These combinators are a small predefined set of tools that mimic regular expression operators to provide a modular structure for building concise queries.

Several existing implementations had already been designed by using these small software blocks. Three implementations are studied in this thesis. They have each been set up in the past with different objectives. The first implementation, DRex, being the ancestor of QREs, is first analyzed to understand the differences with their actual other implementations. A second implementation, called StreamQRE, is reviewed more in depth because it perfectly reflects the image of the theoretical specifications of QREs. A last implementation, called NetQRE, shows the ability of the language to adapt to the specific domain of network monitoring. The analysis of the different implementations made it possible to identify areas for improvement.

All the examples exposed during the various chapters have at the heart of their use cases the same area of interest, the road traffic monitoring. This not only makes it easy to compare algorithms with each other, but also gives the opportunity to solve real-life problems. Among these use cases, a tunnel traffic monitoring algorithm is created to compute the maximum capacity in vehicles by hour of a tunnel. Being implemented on the one hand with a naive solution, and on the other with StreamQRE, the differences between the two approaches are highlighted, in particular the ease to modularize, modify and test the StreamQRE queries.

An identical comparative approach, is used to elaborate the last chapter. Within it, a new implementation is developed. Named LazyQRE, this new creation leverages the functional programming paradigm by

taking advantage of the Scala language.

Both in the naive approach and the StreamQRE approach, the designed algorithms performed the aggregation operations on the stream as the items are received. Regularly, the produced results were only partial and therefore had no interest in being used. LazyQRE opts for storing the operations in memory waiting for the guarantee of a final result instead of directly executing them on the stream. The user then has the choice of evaluating them to compute the result, or not. Such practices prove the ability of this implementation to bring a net gain in performance.

Dynamic function creation for lazy evaluating operations on a stream are achievable by the means of function composition and higher-order functions features. They make it possible to partially or fully apply arguments to a function while delaying its evaluation. The more resource-intensive the operations to be performed on the stream, the greater the performance gain will potentially be.

This implementation would not be conceivable without any tradeoffs. In particular, this tool needs to keep partial operations in memory, and it therefore consumes more memory than other implementations. It is also limited in the number of nested aggregated operations by the maximum call stack memory allocated in the runtime environment. Some configurable parameters have been provided to avoid these problems. However, other potential solutions still leave room for improvement.

In the future, the nesting of too many operations could be solved by the implementation of tail recursion optimization to avoid stack overflow errors, or by executing functions in a more iterative way. A partial iterative execution, keeping some intermediate results in memory, can be a way to limit too large function composition. Another performance optimization that could be achieved is the reuse of partial results common to several stages of the stream evaluation. By keeping in memory the parts of the stream already evaluated, and by replacing them with their results for future calculations, this optimization consists in ensuring that the same term will never be evaluated more than once. Although a slight cost in memory might come with it, a lot of CPU resources could be saved for a whole category of problems.

This thesis had the advantage of offering an important knowledge in streaming processing problems, in language theories, and have given a wide experience of the impact, and the importance, of combining these two concepts.

# Bibliography

- [1] H. Abbas, R. Alur, K. Mamouras, R. Mangharam, and A. Rodionova. Real-time decision policies with predictable performance. *Proceedings of the IEEE*, 106(9):1593–1615, 2018.
- [2] H. Abbas, A. Rodionova, E. Bartocci, S. A. Smolka, and R. Grosu. Regular expressions for irregular rhythms. *arXiv preprint arXiv:1612.07770*, 2016.
- [3] H. Abbas, A. Rodionova, E. Bartocci, S. A. Smolka, and R. Grosu. Quantitative regular expressions for arrhythmia detection algorithms. In *International Conference on Computational Methods in Systems Biology*, pages 23–39. Springer, 2017.
- [4] H. Abbas, A. Rodionova, K. Mamouras, E. Bartocci, S. A. Smolka, and R. Grosu. Quantitative regular expressions for arrhythmia detection. *IEEE/ACM transactions on computational biology and bioinformatics*, 16(5):1586–1597, 2018.
- [5] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, may 1993.
- [6] R. Alur and L. D’Antoni. Streaming tree transducers. In *Automata, Languages, and Programming*, pages 42–53. Springer Berlin Heidelberg, 2012.
- [7] R. Alur, L. D’Antoni, J. V. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions, cost register automata, and generalized min-cost problems. *arXiv preprint arXiv:1111.0670*, 2011.
- [8] R. Alur, L. D’Antoni, and M. Raghothaman. DReX: A Declarative Language for Efficiently Evaluating Regular String Transformations. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, jan 2015.
- [9] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, apr 1994.
- [10] R. Alur, D. Fisman, and M. Raghothaman. Regular programming for quantitative properties of data streams. In *European Symposium on Programming*, pages 15–40. Springer, 2016.
- [11] R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) - CSL-LICS ’14*. ACM Press, 2014.
- [12] R. Alur and K. Mamouras. An introduction to the streamqre language. *Dependable Software Systems Engineering*, 50(1), 2017.
- [13] R. Alur, K. Mamouras, and D. Ulus. Derivatives of quantitative regular expressions. In *Models, algorithms, logics and tools*, pages 75–95. Springer, 2017.
- [14] M. A. Amin, M. Kashif, M. Umer, M. Kamran, A. Kanwal, and M. Azeem. Video streaming analytics for traffic monitoring systems. *International Journal of Advanced Computer Science and Applications*, 9(11), 01 2018.
- [15] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, jul 2005.
- [16] I. R. M. Association, editor. *Climate Change and Environmental Concerns*. IGI Global, 2018.

- [17] K. Bae and J. Lee. Bounded model checking of signal temporal logic properties using syntactic separation. 3(POPL):1–30, jan 2019.
- [18] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.
- [19] K. Chatterjee, L. Doyen, and T. A. Henzinger. Quantitative languages. *ACM Transactions on Computational Logic*, 11(4):1–38, jul 2010.
- [20] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. Quantitative verification of implantable cardiac pacemakers over hybrid heart models. *Information and Computation*, 236:87–101, aug 2014.
- [21] Y. Chen, S. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006.
- [22] P. Chiusano and R. Bjarnason. *Functional Programming in Scala*. Manning Publications Co., USA, 1st edition, 2014.
- [23] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, 2007. release October, 12th 2007.
- [24] B. Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53–75, apr 1994.
- [25] L. D’Antoni and M. Veanes. Static analysis of string encoders and decoders. In *Lecture Notes in Computer Science*, pages 209–228. Springer Berlin Heidelberg, 2013.
- [26] L. D’Antoni and M. Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification*, pages 47–67. Springer International Publishing, 2017.
- [27] C. Doblander, T. Rabl, and H.-A. Jacobsen. Processing big events with showers and streams. In *Specifying Big Data Benchmarks*, pages 60–71. Springer Berlin Heidelberg, 2014.
- [28] M. Droste, W. Kuich, and H. Vogler, editors. *Handbook of Weighted Automata*. Springer Berlin Heidelberg, 2009.
- [29] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. One-pass wavelet decompositions of data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):541–554, may 2003.
- [30] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases, VLDB ’01*, page 79–88, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
- [31] GoodVision. Goodvision life traffic : Real-time traffic video analytics for road-side sensors.
- [32] S. Greenhut, J. Jenkins, and R. MacDonald. A stochastic network model of the interaction between cardiac rhythm and artificial pacemaker. *IEEE Transactions on Biomedical Engineering*, 40(9):845–858, 1993.
- [33] T. A. Henzinger. The theory of hybrid automata. pages 265–292. Springer Berlin Heidelberg, 2000.
- [34] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: a hybrid methodology for the performance analysis of embedded real-time systems. 14(3):193–227, jun 2010.
- [35] A. Maglie. ReactiveX and RxJava, Reactive Java Programming. In *Reactive Java Programming*, pages 1–9. Apress, 2016.
- [36] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer Berlin Heidelberg, 2004.
- [37] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna. Streamqre: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 693–708, 2017.



- [38] A. Margara and T. Rabl. Definition of data streams. In *Encyclopedia of Big Data Technologies*, pages 648–652. Springer International Publishing, 2019.
- [39] Microsoft. Reactivex - an api for asynchronous programming with observable streams.
- [40] L. Moreira-Matias, J. Moreira, J. Gama, and M. Ferreira. On improving operational planning and control in public transportation networks using streaming data: A machine learning approach. In *Proceedings of ECML/PKDD 2014 PhD Session at Nancy, France*, 01 2014.
- [41] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.
- [42] M. Odersky. Scala by example. <http://scala.epfl.ch/docu/files/ScalaByExample.pdf>, October 2005.
- [43] J. A. Orchilles. Ssl renegotiation dos.
- [44] J.-F. Raskin. An introduction to hybrid automata. In *Handbook of Networked and Embedded Control Systems*, pages 491–517. Birkhäuser Boston, 2005.
- [45] G. L. Steele. Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, LAMBDA. In *Proceedings of the 1977 annual conference on - ACM '77*. ACM Press, 1977.
- [46] TrafficVision. Trafficvision : Traffic intelligence from video.
- [47] T. C. . R. Tunnels. Cross section design of bidirectional road tunnels, 2004.
- [48] M. Vaziri, O. Tardieu, R. Rabbah, P. Suter, and M. Hirzel. Stream processing with a spreadsheet. In *ECOOOP 2014 – Object-Oriented Programming*, pages 360–384. Springer Berlin Heidelberg, 2014.
- [49] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Presentation of quantitative network monitoring with netqre at acm sigcomm 2017. ACM SIGCOMM, 2017.
- [50] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with netqre. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 99–112, 2017.



# Appendix A

## LazyQRE

LazyQRE is a brand new implementation of quantitative regular expressions that has been fully developed for this thesis. The library is written in Scala and can be executed on the Java Virtual Machine. It is developed using the functional programming paradigm. The methodology employed is based on the functional programming paradigm. The source code is available in the repository <https://github.com/rombru/LazyQRE>.

The code is divided into two parts:

- The combinators evaluation classes;
- The combinators configuration classes.

### A.1 Combinators evaluation classes

These classes define the logic of the combinators. When reading the stream, they are recreated each time the subset of the stream satisfying the domain of combinators changes. They are initialized and reinitialized by the `start` method, and they receive the new elements of the stream by the `next` method.

The first two files, `Eval` and `EvalExtension`, factorize a logic common to all combinators.

#### Contents

File program 1: Eval trait . . . . .	109
File program 2: Eval extension object . . . . .	110
File program 3: Apply Combinator . . . . .	110
File program 4: Atom Combinator . . . . .	110
File program 5: Combine Combinator . . . . .	111
File program 6: Combine3 Combinator . . . . .	111
File program 7: Combine4 Combinator . . . . .	112
File program 8: Else Combinator . . . . .	113
File program 9: Iter Combinator . . . . .	113
File program 10: Split Combinator . . . . .	114
File program 11: Streaming Compose Combinator . . . . .	115
File program 12: Window Combinator . . . . .	115
File program 13: Tumbling Window Combinator . . . . .	116

#### Code

```
</> File program 1: Eval trait </>
1 package be.bruyere.romain.eval
2
3 trait Eval[In, Out, Fn] {
4   val output: Option[Fn]
```

```

5
6   def next(item: In): Eval[In, Out, Fn]
7
8   def start(current: (() => Out) => Fn): Eval[In, Out, Fn]
9 }

```

&lt;/&gt;

File program 2: *Eval extension object*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 object EvalExtension {
4   implicit class StartEval[In, Out](eval: Eval[In, Out, () => Out]) {
5     def next(item: In): StartEval[In, Out] = eval.next(item)
6
7     def result(): Option[Out] = eval.output map (o => o())
8
9     def resultFn(): Option[() => Out] = eval.output
10  }
11 }

```

&lt;/&gt;

File program 3: *Apply Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.ApplyQRE
4
5 case class Apply[In, Out, ChildOut, Fn] private(child: Eval[In, ChildOut, Fn], qre:
6   ↳ ApplyQRE[In, Out, ChildOut], output: Option[Fn]) extends Eval[In, Out, Fn] {
7   override def next(item: In): Eval[In, Out, Fn] = {
8     val newChild = child.next(item)
9     Apply(newChild, qre, newChild.output)
10  }
11
12  override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
13    val newFn = qre.createNewF(fn)
14    val newChild = child.start(newFn)
15    Apply(newChild, qre, newChild.output)
16  }
17 }

```

&lt;/&gt;

File program 4: *Atom Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.AtomQRE
4
5 case class Atom[In, Out, Fn] private(predicate: In => Boolean, qre: AtomQRE[In, Out],
6   ↳ current: Option[In => Fn], output: Option[Fn])
7   extends Eval[In, Out, Fn] {
8
9   def start(fn: (() => Out) => Fn): Atom[In, Out, Fn] = {
10     val newFn = qre.createNewF(fn)
11     Atom(predicate, qre, Some(newFn), None)
12   }
13
14   def next(item: In): Atom[In, Out, Fn] = {
15     if (predicate(item) && current.isDefined) {

```

```

15     Atom(predicate, qre, None, current map (c => c(item)))
16   } else {
17     Atom( predicate, qre, None, None)
18   }
19 }
20 }

```

&lt;/&gt;

File program 5: *Combine Combinator*

&lt;/&gt;

```

1  package be.bruyere.romain.eval
2
3  import be.bruyere.romain.qre.CombineQRE
4
5  case class Combine[In, Out, Child1Out, Child2Out, Fn] private
6  (
7    child1: Eval[In, Child1Out, (() => Child2Out) => Fn],
8    child2: Eval[In, Child2Out, () => Child2Out],
9    qre: CombineQRE[In, Out, Child1Out, Child2Out],
10   output: Option[Fn]
11 ) extends Eval[In, Out, Fn] {
12
13   override def next(item: In): Eval[In, Out, Fn] = {
14     val newChild1 = child1.next(item)
15     val newChild2 = child2.next(item)
16     Combine(newChild1, newChild2, qre, getOutput(newChild1, newChild2))
17   }
18
19   override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
20     val newFn = qre.createNewF(fn)
21     val newChild1 = child1.start(newFn)
22     val newChild2 = child2.start(identity)
23     Combine(newChild1, newChild2, qre, getOutput(newChild1, newChild2))
24   }
25
26   private def getOutput(newChild1: Eval[In, Child1Out, (() => Child2Out) => Fn],
27     ↪ newChild2: Eval[In, Child2Out, () => Child2Out]) = {
28     for {out1 <- newChild1.output; out2 <- newChild2.output}
29       yield out1.apply(out2)
30   }
31 }

```

&lt;/&gt;

File program 6: *Combine3 Combinator*

&lt;/&gt;

```

1  package be.bruyere.romain.eval
2
3  import be.bruyere.romain.qre.Combine3QRE
4
5  case class Combine3[In, Out, Child1Out, Child2Out, Child3Out, Fn] private
6  (
7    child1: Eval[In, Child1Out, (() => Child2Out, () => Child3Out) => Fn],
8    child2: Eval[In, Child2Out, () => Child2Out],
9    child3: Eval[In, Child3Out, () => Child3Out],
10   qre: Combine3QRE[In, Out, Child1Out, Child2Out, Child3Out],
11   output: Option[Fn]
12 ) extends Eval[In, Out, Fn] {
13   override def next(item: In): Eval[In, Out, Fn] = {
14     val newChild1 = child1.next(item)

```

```

15     val newChild2 = child2.next(item)
16     val newChild3 = child3.next(item)
17     Combine3(newChild1, newChild2, newChild3, qre, getOutput(newChild1, newChild2,
    ↪ newChild3))
18 }
19
20 override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
21     val newFn = qre.createNewF(fn)
22     val newChild1 = child1.start(newFn)
23     val newChild2 = child2.start(identity)
24     val newChild3 = child3.start(identity)
25     Combine3(newChild1, newChild2, newChild3, qre, getOutput(newChild1, newChild2,
    ↪ newChild3))
26 }
27
28 private def getOutput
29 (
30     child1: Eval[In, Child1Out, (() => Child2Out, () => Child3Out) => Fn],
31     child2: Eval[In, Child2Out, () => Child2Out],
32     child3: Eval[In, Child3Out, () => Child3Out],
33 ) = {
34     for {out1 <- child1.output; out2 <- child2.output; out3 <- child3.output}
35         yield out1.apply(out2, out3)
36 }
37 }

```

</> File program 7: *Combine4 Combinator* </>

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.Combine4QRE
4
5 case class Combine4[In, Out, Child1Out, Child2Out, Child3Out, Child4Out, Fn] private
6 (
7     child1: Eval[In, Child1Out, (() => Child2Out, () => Child3Out, () => Child4Out) =>
    ↪ Fn],
8     child2: Eval[In, Child2Out, () => Child2Out],
9     child3: Eval[In, Child3Out, () => Child3Out],
10    child4: Eval[In, Child4Out, () => Child4Out],
11    qre: Combine4QRE[In, Out, Child1Out, Child2Out, Child3Out, Child4Out],
12    output: Option[Fn]
13 ) extends Eval[In, Out, Fn] {
14     override def next(item: In): Eval[In, Out, Fn] = {
15         val newChild1 = child1.next(item)
16         val newChild2 = child2.next(item)
17         val newChild3 = child3.next(item)
18         val newChild4 = child4.next(item)
19         Combine4(newChild1, newChild2, newChild3, newChild4, qre, getOutput(newChild1,
    ↪ newChild2, newChild3, newChild4))
20     }
21
22     override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
23         val newFn = qre.createNewF(fn)
24         val newChild1 = child1.start(newFn)
25         val newChild2 = child2.start(identity)
26         val newChild3 = child3.start(identity)
27         val newChild4 = child4.start(identity)

```

```

28   Combine4(newChild1, newChild2, newChild3, newChild4, qre, getOutput(newChild1,
29   ↪ newChild2, newChild3, newChild4))
30 }
31 private def getOutput
32 (
33   child1: Eval[In, Child1Out, (() => Child2Out, () => Child3Out, () => Child4Out) =>
34   ↪ Fn],
35   child2: Eval[In, Child2Out, () => Child2Out],
36   child3: Eval[In, Child3Out, () => Child3Out],
37   child4: Eval[In, Child4Out, () => Child4Out],
38 ) = {
39   for {out1 <- child1.output; out2 <- child2.output; out3 <- child3.output; out4 <-
40   ↪ child4.output}
41     yield out1.apply(out2, out3, out4)
42 }

```

&lt;/&gt;

File program 8: *Else Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 case class Else[In, Out, Fn] private(child1: Eval[In, Out, Fn], child2: Eval[In, Out,
4 ↪ Fn], output: Option[Fn]) extends Eval[In, Out, Fn] {
5   override def next(item: In): Eval[In, Out, Fn] = {
6     val newChild1 = child1.next(item)
7     val newChild2 = child2.next(item)
8     Else(newChild1, newChild2, newChild1.output orElse newChild2.output)
9   }
10
11   override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
12     val newChild1 = child1.start(fn)
13     val newChild2 = child2.start(fn)
14     Else(newChild1, newChild2, newChild1.output orElse newChild2.output)
15   }
16 }

```

&lt;/&gt;

File program 9: *Iter Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.IterQRE
4
5 case class Iter[In, Out, ChildOut, Agg, Fn] private(child: Eval[In, ChildOut, (() =>
6 ↪ Agg, (() => Agg) => Fn, Long)], qre: IterQRE[In, Out, ChildOut, Agg], output:
7 ↪ Option[Fn]) extends Eval[In, Out, Fn] {
8   override def next(item: In): Eval[In, Out, Fn] = {
9
10     val newChild = child.next(item)
11     newChild.output match {
12       case Some((trans, out, max)) =>
13         max match {
14           case 0 =>
15             val t = trans()
16             val newFn = qre.createNewF(t, out, qre.iterLimit)
17             Iter(newChild.start(newFn), qre, Some(out(() => t)))
18           case _ =>

```

```

17     val newFn = qre.createNewF(trans, out, max-1)
18     Iter(newChild.start(newFn), qre, Some(out(trans)))
19   }
20   case None => Iter(newChild, qre, None)
21 }
22 }
23
24 override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
25   val out = qre.createOutF(fn)
26   val newFn = qre.createNewF(qre.init, out, qre.iterLimit)
27   Iter(child.start(newFn), qre, Some(fn(() => qre.outputF(qre.init))))
28 }
29 }

```

&lt;/&gt;

File program 10: *Split Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.SplitQRE
4
5 case class Split[In, Out, ChildLOut, ChildROut, Agg, Fn] private
6 (childL: Eval[In, ChildLOut, (() => ChildROut) => Fn], childR: Eval[In, ChildROut,
7   ↪ Fn], qre: SplitQRE[In, Out, ChildLOut, ChildROut, Agg], output: Option[Fn])
8 extends Eval[In, Out, Fn] {
9
10  def start(fn: (() => Out) => Fn): Split[In, Out, ChildLOut, ChildROut, Agg, Fn] = {
11    val newFn = qre.createNewF(fn)
12
13    val newChildL = childL.start(newFn)
14    newChildL.output match {
15      case Some(childOutput) =>
16        val fn = qre.createOutputLeftF(childOutput)
17
18        val newChildR = childR.start(fn)
19
20        newChildR.output match {
21          case Some(_) => Split[In, Out, ChildLOut, ChildROut, Agg, Fn](newChildL,
22            ↪ newChildR, qre, newChildR.output)
23          case None => Split(newChildL, newChildR, qre, None)
24        }
25      case None => Split(newChildL, childR, qre, None)
26    }
27  }
28
29  def next(item: In): Split[In, Out, ChildLOut, ChildROut, Agg, Fn] = {
30    val newChildL = childL.next(item)
31    val newChildR = childR.next(item)
32
33    val (newChildR2, output) = newChildL.output match {
34      case Some(childOutput) => restartRight(childOutput, newChildR)
35      case None => (newChildR, newChildR.output)
36    }
37    Split(newChildL, newChildR2, qre, output)
38  }

```



```

39 private def restartRight(outputL: (() => ChildROut) => Fn, newChildR: Eval[In,
    ↪ ChildROut, Fn]) = {
40     val fn = qre.createOutputLeftF(outputL)
41
42     val newChildR2 = newChildR.start(fn)
43
44     newChildR2.output match {
45         case Some(_) => (newChildR2, newChildR2.output)
46         case None => (newChildR2, newChildR.output)
47     }
48 }
49 }

```

&lt;/&gt;

File program 11: *Streaming Compose Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 case class StreamingCompose[In, Out, ChildOut, Fn] private(child1: Eval[In,
    ↪ ChildOut, () => ChildOut], child2: Eval[ChildOut, Out, Fn], output: Option[Fn])
    ↪ extends Eval[In, Out, Fn] {
4     override def next(item: In): Eval[In, Out, Fn] = {
5         val newChild1 = child1.next(item)
6         strcompose(newChild1, child2)
7     }
8
9     override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
10         val newChild1 = child1.start(identity)
11         val newChild2 = child2.start(fn)
12         strcompose(newChild1, newChild2)
13     }
14
15 private def strcompose(child1: Eval[In, ChildOut, () => ChildOut], child2:
    ↪ Eval[ChildOut, Out, Fn]) = {
16     child1.output match {
17         case Some(out) =>
18             val newChild2 = child2.next(out())
19             StreamingCompose(child1, newChild2, newChild2.output)
20         case None =>
21             StreamingCompose(child1, child2, None)
22     }
23 }
24 }

```

&lt;/&gt;

File program 12: *Window Combinator*

&lt;/&gt;

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.WindowQRE
4 import be.bruyere.romain.struct.AggregateQueue
5
6 case class Window[In, Out, ChildOut, Agg, Fn] private(child: Eval[In, ChildOut, (()
    ↪ => Agg) => Fn, AggregateQueue[ChildOut, Agg]]], qre: WindowQRE[In, Out, ChildOut,
    ↪ Agg], output: Option[Fn]) extends Eval[In, Out, Fn] {
7     override def next(item: In): Eval[In, Out, Fn] = {
8
9         val newChild = child.next(item)
10         newChild.output match {

```

```

11     case Some((out, queue)) =>
12         if(queue.size == qre.winSize) {
13             val result = queue.aggregate(qre.init, qre.transformF)
14             val newFn = qre.createNewF(out, queue.dequeue())
15             Window(newChild.start(newFn), qre, Some(out(() => result())))
16         } else {
17             val newFn = qre.createNewF(out, queue)
18             Window(newChild.start(newFn), qre, None)
19         }
20     case None => Window(newChild, qre, None)
21 }
22 }
23
24 override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
25     val out = qre.createOutF(fn)
26     val newFn = qre.createNewF(out, new AggQueue[ChildOut, Agg]())
27     Window(child.start(newFn), qre, None)
28 }
29 }

```

File program 13: *Tumbling Window Combinator*

```

1 package be.bruyere.romain.eval
2
3 import be.bruyere.romain.qre.TumblingWindowQRE
4 import be.bruyere.romain.struct.AggQueue
5
6 case class TumblingWindow[In, Out, ChildOut, Agg, Fn] private(child: Eval[In,
7     ↳ ChildOut, ((() => Agg) => Fn, AggQueue[ChildOut, Agg])], qre:
8     ↳ TumblingWindowQRE[In, Out, ChildOut, Agg], output: Option[Fn]) extends Eval[In,
9     ↳ Out, Fn] {
10     override def next(item: In): Eval[In, Out, Fn] = {
11
12         val newChild = child.next(item)
13         newChild.output match {
14             case Some((out, queue)) =>
15                 if(queue.size == qre.winSize) {
16                     val result = queue.aggregate(qre.init, qre.transformF)
17                     val newFn = qre.createNewF(out, new AggQueue[ChildOut, Agg]())
18                     TumblingWindow(newChild.start(newFn), qre, Some(out(() => result())))
19                 } else {
20                     val newFn = qre.createNewF(out, queue)
21                     TumblingWindow(newChild.start(newFn), qre, None)
22                 }
23             case None => TumblingWindow(newChild, qre, None)
24         }
25     }
26 }
27
28 override def start(fn: (() => Out) => Fn): Eval[In, Out, Fn] = {
29     val out = qre.createOutF(fn)
30     val newFn = qre.createNewF(out, new AggQueue[ChildOut, Agg]())
31     TumblingWindow(child.start(newFn), qre, None)
32 }
33 }

```

## A.2 Combinators configuration classes

These classes define the configuration of the combinators. They create the evaluation classes for the first time, according to the parameters specified by the users in the queries. They also contain the logic to create higher-order functions. This logic is called by the evaluation classes when creating a new term.

The first file, QRE, factorizes a logic common to all combinators.

### Contents

File program 1: QRE trait . . . . .	117
File program 2: Apply QRE . . . . .	117
File program 3: Atom QRE . . . . .	118
File program 4: Combine QRE . . . . .	118
File program 5: Combine3 QRE . . . . .	118
File program 6: Combine4 QRE . . . . .	119
File program 7: Else QRE . . . . .	119
File program 8: Iter QRE . . . . .	119
File program 9: Split QRE . . . . .	120
File program 10: Streaming Compose QRE . . . . .	121
File program 11: Window QRE . . . . .	121
File program 12: Tumbling Window QRE . . . . .	121

### Code

```

</> File program 1: QRE trait </>
1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.Eval
4 import be.bruyere.romain.eval.EvalExtension.StartEval
5
6 trait QRE[In, Out] {
7   def start(): StartEval[In, Out] = {
8     create[() => Out]() .start(identity)
9   }
10
11   protected[qre] def create[Fn]() : Eval[In, Out, Fn]
12 }

```

```

</> File program 2: Apply QRE </>
1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Apply, Eval}
4
5 case class ApplyQRE[In, Out, ChildOut] (child: QRE[In, ChildOut], outputF: ChildOut
  ↳ => Out) extends QRE[In, Out] {
6
7   protected[qre] override def create[Fn]() : Eval[In, Out, Fn] = {
8     Apply[In, Out, ChildOut, Fn](child.create(), this, None)
9   }
10
11   def createNewF[Fn](fn: (() => Out) => Fn): (() => ChildOut) => Fn = {
12     (x: () => ChildOut) => fn(() => outputF(x()))
13   }
14 }

```

&lt;/&gt;

File program 3: *Atom QRE*

&lt;/&gt;

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Atom, Eval}
4
5 case class AtomQRE[In, Out] (predicate: In => Boolean, outputF: In => Out) extends
6   ↳ QRE[In, Out] {
7
8   protected[qre] override def create[Fn](): Eval[In, Out, Fn] = {
9     Atom[In, Out, Fn](predicate, this, None, None)
10  }
11
12  def createNewF[Fn](fn: (() => Out) => Fn): In => Fn = {
13    (x: In) => fn(() => outputF(x))
14  }
15 }

```

&lt;/&gt;

File program 4: *Combine QRE*

&lt;/&gt;

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Combine, Eval}
4
5 case class CombineQRE[In, Out, Child1Out, Child2Out] (child1: QRE[In, Child1Out],
6   ↳ child2: QRE[In, Child2Out], transformF: (Child1Out, Child2Out) => Out) extends
7   ↳ QRE[In, Out] {
8
9   protected[qre] override def create[Fn](): Eval[In, Out, Fn] = {
10     Combine[In, Out, Child1Out, Child2Out, Fn](child1.create(), child2.create(),
11       ↳ this, None)
12   }
13
14   def createNewF[Fn](fn: (() => Out) => Fn): (() => Child1Out) => (() => Child2Out) =>
15     ↳ Fn = {
16       (x: () => Child1Out) => {
17         (y: () => Child2Out) => fn(() => transformF.curried(x())(y()))
18       }
19     }
20 }

```

&lt;/&gt;

File program 5: *Combine3 QRE*

&lt;/&gt;

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Combine3, Eval}
4
5 case class Combine3QRE[In, Out, Child1Out, Child2Out, Child3Out] (child1: QRE[In,
6   ↳ Child1Out], child2: QRE[In, Child2Out], child3: QRE[In, Child3Out], transformF:
7   ↳ (Child1Out, Child2Out, Child3Out) => Out) extends QRE[In, Out] {
8
9   protected[qre] override def create[Fn](): Eval[In, Out, Fn] = {
10     Combine3[In, Out, Child1Out, Child2Out, Child3Out, Fn](child1.create(),
11       ↳ child2.create(), child3.create(), this, None)
12   }
13 }

```

```

11 def createNewF[Fn](fn: (() => Out) => Fn): (() => Child1Out) => (() => Child2Out, ()
    ↪ => Child3Out) => Fn = {
12   (x: () => Child1Out) => {
13     (y: () => Child2Out, z: () => Child3Out) => fn(() =>
    ↪ transformF.curried(x())(y())(z()))
14   }
15 }
16 }

```

</> File program 6: *Combine4 QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Combine4, Eval}
4
5 case class Combine4QRE[In, Out, Child1Out, Child2Out, Child3Out, Child4Out](child1:
    ↪ QRE[In, Child1Out], child2: QRE[In, Child2Out], child3: QRE[In, Child3Out],
    ↪ child4: QRE[In, Child4Out], transformF:
    ↪ (Child1Out, Child2Out, Child3Out, Child4Out) => Out) extends QRE[In, Out] {
6
7   protected[qre] override def create[Fn]() : Eval[In, Out, Fn] = {
8     Combine4[In, Out, Child1Out, Child2Out, Child3Out, Child4Out,
    ↪ Fn](child1.create(), child2.create(), child3.create(), child4.create(),
    ↪ this, None)
9   }
10
11 def createNewF[Fn](fn: (() => Out) => Fn): (() => Child1Out) => (() => Child2Out, ()
    ↪ => Child3Out, () => Child4Out) => Fn = {
12   (x: () => Child1Out) => {
13     (y: () => Child2Out, z: () => Child3Out, w: () => Child4Out) => fn(() =>
    ↪ transformF.curried(x())(y())(z())(w()))
14   }
15 }
16 }

```

</> File program 7: *Else QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Else, Eval}
4
5 case class ElseQRE[In, Out] (child1: QRE[In, Out], child2: QRE[In, Out]) extends
    ↪ QRE[In, Out] {
6
7   protected[qre] override def create[Fn]() : Eval[In, Out, Fn] = {
8     Else[In, Out, Fn](child1.create(), child2.create(), None)
9   }
10 }

```

</> File program 8: *Iter QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Eval, Iter}
4

```

```

5 case class IterQRE[In, Out, ChildOut, Agg] (child: QRE[In, ChildOut], init: Agg,
  ↪ transformF: (Agg, ChildOut) => Agg, outputF: Agg => Out, iterLimit: Long) extends
  ↪ QRE[In, Out] {
6
7   def this(child: QRE[In, ChildOut], init: Agg, transformF: (Agg, ChildOut) => Agg,
  ↪ outputF: Agg => Out) = this(child, init, transformF, outputF, 1)
8
9   protected[qre] override def create[Fn](): Eval[In, Out, Fn] = {
10     Iter[In, Out, ChildOut, Agg, Fn](child.create(), this, None)
11   }
12
13   def createNewF[Fn](trans: Agg, out: (() => Agg) => Fn, max: Long): (() => ChildOut)
  ↪ => (() => Agg, (() => Agg) => Fn, Long) = createNewF(() => trans, out, max)
14
15   def createNewF[Fn](trans: () => Agg, out: (() => Agg) => Fn, max: Long): (() =>
  ↪ ChildOut) => (() => Agg, (() => Agg) => Fn, Long) = {
16     (x: () => ChildOut) => {
17       val newTrans = () => transformF.curried(trans())(x())
18       (newTrans, out, max)
19     }
20   }
21
22   def createOutF[Fn](fn: (() => Out) => Fn): (() => Agg) => Fn = {
23     (y: () => Agg) => fn(() => outputF(y()))
24   }
25 }

```

</> File program 9: *Split QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Eval, Split}
4
5 case class SplitQRE[In, Out, ChildLOut, ChildROut, Agg] (childL: QRE[In, ChildLOut],
  ↪ childR: QRE[In, ChildROut], transformF: (ChildLOut, ChildROut) => Agg, outputF:
  ↪ Agg => Out) extends QRE[In, Out] {
6
7   protected[qre] override def create[Fn](): Eval[In, Out, Fn] = {
8     Split[In, Out, ChildLOut, ChildROut, Agg, Fn](childL.create(), childR.create(),
  ↪ this, None)
9   }
10
11   def createNewF[Fn](fn: (() => Out) => Fn): (() => ChildLOut) => (() => ChildROut) =>
  ↪ Fn = {
12     (x: () => ChildLOut) => {
13       (y: () => ChildROut) => fn(() => outputF(transformF.curried(x())(y())))
14     }
15   }
16
17   def createOutputLeftF[Fn](outputL: (() => ChildROut) => Fn): (() => ChildROut) =>
  ↪ Fn = {
18     (x: () => ChildROut) => outputL(x)
19   }
20 }

```

</> File program 10: *Streaming Compose QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Eval, StreamingCompose}
4
5 case class StreamingComposeQRE[In, Out, ChildOut](child1: QRE[In, ChildOut], child2:
6   ↳ QRE[ChildOut, Out]) extends QRE[In, Out] {
7
8   protected[qre] override def create[Fn]() : Eval[In, Out, Fn] = {
9     StreamingCompose[In, Out, ChildOut, Fn](child1.create(), child2.create(), None)
10  }
11 }

```

</> File program 11: *Window QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Eval, Window}
4 import be.bruyere.romain.struct.AggregateQueue
5
6 case class WindowQRE[In, Out, ChildOut, Agg](child: QRE[In, ChildOut], init: Agg,
7   ↳ transformF: (Agg, ChildOut) => Agg, outputF: Agg => Out, winSize: Int) extends
8   ↳ QRE[In, Out] {
9
10  protected[qre] override def create[Fn]() : Eval[In, Out, Fn] = {
11    Window[In, Out, ChildOut, Agg, Fn](child.create(), this, None)
12  }
13
14  def createNewF[Fn](out: (() => Agg) => Fn, queue: AggregateQueue[ChildOut, Agg]): (() =>
15    ↳ ChildOut) => (() => Agg) => Fn, AggregateQueue[ChildOut, Agg]) = {
16    (x: () => ChildOut) => (out, queue.enqueue(x))
17  }
18
19  def createOutF[Fn](fn: (() => Out) => Fn): (() => Agg) => Fn = {
20    (y: () => Agg) => fn(() => outputF(y))
21  }
22 }

```

</> File program 12: *Tumbling Window QRE* </>

```

1 package be.bruyere.romain.qre
2
3 import be.bruyere.romain.eval.{Eval, TumblingWindow}
4 import be.bruyere.romain.struct.AggregateQueue
5
6 case class TumblingWindowQRE[In, Out, ChildOut, Agg](child: QRE[In, ChildOut], init:
7   ↳ Agg, transformF: (Agg, ChildOut) => Agg, outputF: Agg => Out, winSize: Int)
8   ↳ extends QRE[In, Out] {
9
10  protected[qre] override def create[Fn]() : Eval[In, Out, Fn] = {
11    TumblingWindow[In, Out, ChildOut, Agg, Fn](child.create(), this, None)
12  }
13
14  def createNewF[Fn](out: (() => Agg) => Fn, queue: AggregateQueue[ChildOut, Agg]): (() =>
15    ↳ ChildOut) => (() => Agg) => Fn, AggregateQueue[ChildOut, Agg]) = {
16    (x: () => ChildOut) => (out, queue.enqueue(x))
17  }

```

```

14   }
15
16   def createOutF[Fn](fn: (() => Out) => Fn): (() => Agg) => Fn = {
17     (y: () => Agg) => fn(() => outputF(y()))
18   }
19 }

```

### A.3 Structure class

A minimal implementation of a Queue was necessary to manage the sliding window pattern. This is the next class.

```

</> File program 1: Aggregation Queue </>
1  package be.bruyere.romain.struct
2
3  import scala.collection.immutable.Queue
4
5  case class AggQueue[A,B](queue: Queue[() => A] = Queue[() => A]() {
6
7    def this(item: () => A) {
8      this(Queue[() => A](item))
9    }
10
11    def enqueue(item: () => A): AggQueue[A,B] = {
12      AggQueue(queue.enqueue(item))
13    }
14
15    def dequeue(): AggQueue[A,B] = {
16      AggQueue(queue.dequeue._2)
17    }
18
19    def aggregate(init: B, trans: (B, A) => B): () => B = {
20      val fn = (y: A) => trans.curried(init)(y)
21      aggregateRec(() => fn, trans, queue)
22    }
23
24    def aggregateRec(transCurr: () => A => B, trans: (B, A) => B, queue: Queue[() =>
25      ↪ A]): () => B = {
26      if(queue.size == 1) {
27        () => transCurr()(queue.head())
28      } else {
29        aggregateRec(() => trans.curried(transCurr()(queue.head())), trans, queue.tail)
30      }
31    }
32
33    def head: () => A = queue.head
34    def size: Int = queue.size
35  }

```



## Appendix B

# StreamQRE, NetQRE, DReX

### B.1 StreamQRE

StreamQRE is the first true implementation of quantitative regular expressions. This implementation was not developed in this thesis, but was used for demonstration and explanation purposes as well as for comparison with LazyQRE. The code can be downloaded at this url, <https://mamouras.web.rice.edu/StreamQRE/index.html>.

### B.2 NetQRE

NetQRE is a C++ implementation of quantitative regular expressions. The implementation is specialized in the network monitoring domain. The code has not been presented as is in this thesis because it quite different from a desired generic implementation. Source code can be found at this url, <https://github.com/SleepyToDeath/NetQRE>.

### B.3 DReX

DReX is the ancestor of the QRE language. The source code of an implementation have not been found.



# Appendix C

## Case studies

The following case studies have been designed to compare languages on different aspects. They have been integrated into the simulation application described in the next appendix. All the algorithms have been entirely designed within the framework of this thesis.

They are divided into two categories, each grouping together the algorithms solving a same problem:

- Traffic density monitoring;
- Accident monitoring.

The source code of these algorithms, as well as the simulation application, can be downloaded at this url, <https://github.com/rombru/VehicleStreamingApp>.

### C.1 Traffic density monitoring

These algorithms were designed to calculate the density of traffic on an incoming stream. The calculations are configured by the data of the matrix written in the first file. The source code specific to these algorithms can be downloaded from this url, <https://github.com/rombru/VehicleStreamingApp/tree/main/backend/src/main/java/be/bruyere/vehiclegrounding/algo/tunnel>.

#### C.1.1 Equivalence factor matrix

```
</> File program 1: Equivalence factor matrix </>
1 package be.bruyere.vehiclegrounding.algo.tunnel;
2
3 public class EquivalenceFactorMatrix {
4
5     private final Double[] r2_0_400 = new Double[]{1.5, 1.5, 1.5, 1.5, 1.5, 1.5};
6     private final Double[] r2_400_800 = new Double[]{1.5, 1.5, 1.5, 1.5, 1.5, 1.5};
7     private final Double[] r2_800_1200 = new Double[]{1.5, 1.5, 1.5, 1.5, 1.5, 1.5};
8     private final Double[] r2_1200_1600 = new Double[]{2.0, 2.0, 1.5, 1.5, 1.5, 1.5};
9     private final Double[] r2_1600_2400 = new Double[]{3.0, 3.0, 1.5, 1.5, 1.5, 1.5};
10    private final Double[] r2_2400 = new Double[]{3.5, 3.0, 2.5, 2.5, 2.0, 2.0};
11
12    private final Double[] r3_0_400 = new Double[]{1.5, 1.5, 1.5, 1.5, 1.5, 1.5};
13    private final Double[] r3_400_800 = new Double[]{2.5, 2.0, 2.0, 2.0, 2.0, 1.5};
14    private final Double[] r3_800_1200 = new Double[]{4.0, 3.5, 3.5, 3.0, 2.5, 2.0};
15    private final Double[] r3_1200_1600 = new Double[]{5.5, 4.5, 4.0, 4.0, 3.5, 3.0};
16    private final Double[] r3_1600_2400 = new Double[]{6.0, 5.0, 4.5, 4.0, 4.0, 3.0};
17    private final Double[] r3_2400 = new Double[]{6.0, 5.0, 4.5, 4.5, 4.0, 3.0};
18
19    private final Double[] r4_0_400 = new Double[]{1.5, 1.5, 1.5, 1.5, 1.5, 1.5};
```

```

20 private final Double[] r4_400_800 = new Double[]{4.0, 3.5, 3.0, 3.0, 3.0, 2.5};
21 private final Double[] r4_800_1200 = new Double[]{7.0, 6.0, 5.5, 5.0, 4.5, 4.0};
22 private final Double[] r4_1200_1600 = new Double[]{8.0, 6.5, 6.0, 5.5, 4.0, 4.5};
23 private final Double[] r4_1600_2400 = new Double[]{8.0, 7.0, 6.0, 6.0, 5.0, 5.0};
24 private final Double[] r4_2400 = new Double[]{8.0, 7.0, 6.0, 6.0, 5.0, 5.0};
25
26 private final Double[] r5_0_400 = new Double[]{2.0, 1.5, 1.5, 1.5, 1.5, 1.5};
27 private final Double[] r5_400_800 = new Double[]{4.5, 4.0, 3.5, 3.0, 3.0, 2.5};
28 private final Double[] r5_800_1200 = new Double[]{7.0, 6.0, 5.5, 5.0, 4.5, 4.0};
29 private final Double[] r5_1200_1600 = new Double[]{9.0, 8.0, 7.0, 7.0, 6.0, 6.0};
30 private final Double[] r5_1600_2400 = new Double[]{9.5, 8.0, 7.5, 7.0, 6.5, 6.0};
31 private final Double[] r5_2400 = new Double[]{9.5, 8.0, 7.5, 7.0, 6.5, 6.0};
32
33 public Double getCoefficient(
34     Double length,
35     Double slopeGrade,
36     Double truckPercentage
37 ) {
38     if(truckPercentage < 4) {
39         return 1.0;
40     }
41     if(slopeGrade < 2) {
42         return 1.5;
43     } else if(slopeGrade <= 3) {
44         return getCoefficientByLength(length, truckPercentage, r2_0_400,
45             ↪ r2_400_800, r2_800_1200, r2_1200_1600, r2_1600_2400, r2_2400);
46     } else if(slopeGrade <= 4) {
47         return getCoefficientByLength(length, truckPercentage, r3_0_400,
48             ↪ r3_400_800, r3_800_1200, r3_1200_1600, r3_1600_2400, r3_2400);
49     } else if(slopeGrade <= 5) {
50         return getCoefficientByLength(length, truckPercentage, r4_0_400,
51             ↪ r4_400_800, r4_800_1200, r4_1200_1600, r4_1600_2400, r4_2400);
52     } else {
53         return getCoefficientByLength(length, truckPercentage, r5_0_400,
54             ↪ r5_400_800, r5_800_1200, r5_1200_1600, r5_1600_2400, r5_2400);
55     }
56 }
57
58 private Double getCoefficientByLength(
59     Double length,
60     Double truckPercentage,
61     Double[] r_0_400,
62     Double[] r_400_800,
63     Double[] r_800_1200,
64     Double[] r_1200_1600,
65     Double[] r_1600_2400,
66     Double[] r_2400
67 ) {
68     if(length < 400) {
69         return getCoefficientByTruckPercentage(truckPercentage, r_0_400);
70     } else if(length < 800) {
71         return getCoefficientByTruckPercentage(truckPercentage, r_400_800);
72     } else if(length < 1200) {
73         return getCoefficientByTruckPercentage(truckPercentage, r_800_1200);
74     } else if(length < 1600) {
75         return getCoefficientByTruckPercentage(truckPercentage, r_1200_1600);
76     } else if(length < 2400) {

```

```

73         return getCoefficientByTruckPercentage(truckPercentage, r_1600_2400);
74     } else {
75         return getCoefficientByTruckPercentage(truckPercentage, r_2400);
76     }
77 }
78
79 private Double getCoefficientByTruckPercentage(
80     Double truckPercentage,
81     Double[] coefs
82 ) {
83     if(truckPercentage <= 4) {
84         return coefs[0];
85     } else if(truckPercentage <= 6) {
86         return coefs[1];
87     } else if(truckPercentage <= 8) {
88         return coefs[2];
89     } else if(truckPercentage <= 10) {
90         return coefs[3];
91     } else if(truckPercentage <= 15) {
92         return coefs[4];
93     } else {
94         return coefs[5];
95     }
96 }
97 }

```

### C.1.2 Naive algorithm

</>
File program 2: *Naive traffic density algorithm*
</>

```

1  package be.bruyere.vehiclestreaming.algo.tunnel;
2
3  import be.bruyere.vehiclestreaming.service.dto.StreamingDto;
4  import be.bruyere.vehiclestreaming.service.dto.VehicleDto;
5
6  import java.util.Objects;
7
8  import static be.bruyere.vehiclestreaming.service.dto.ItemType.VEHICLE;
9  import static be.bruyere.vehiclestreaming.service.dto.VehicleType.TRUCK;
10
11 public class NaiveAlgo {
12
13     private static final EquivalenceFactorMatrix matrix = new
14         ↪ EquivalenceFactorMatrix();
15
16     private static Double total = 0D;
17     private static Double totalTrucks = 0D;
18     private static Double length = 0D;
19     private static Double slopeGrade = 0D;
20     private static Double habitualUseFactor = 0D;
21
22     private static Integer indexVehicleBy1Hour = 0;
23     private static Double[] vehicleBy1Hour = {0D,0D,0D,0D,0D};
24     private static Integer indexVehicleBy15Min = 0;
25     private static Double[] vehicleBy15Min = {0D, 0D};
26     private static Double totalBy1Hour = null;

```

```

27 private static Double totalBy15Min = null;
28 private static Double speedSum = 0D;
29
30 /**
31  * Configuration of the parameters for the algorithm
32  * @param length the road length
33  * @param slopeGrade the slope grade
34  * @param habitualUseFactor the habitual use factor
35  */
36 public static void configure(Double length, Double slopeGrade, Double
    ↪ habitualUseFactor) {
37     NaiveAlgo.length = length;
38     NaiveAlgo.slopeGrade = slopeGrade;
39     NaiveAlgo.habitualUseFactor = habitualUseFactor;
40
41     NaiveAlgo.indexVehicleBy1Hour = 0;
42     NaiveAlgo.vehicleBy1Hour = new Double[]{0D, 0D, 0D, 0D, 0D};
43     NaiveAlgo.indexVehicleBy15Min = 0;
44     NaiveAlgo.vehicleBy15Min = new Double[]{0D, 0D};
45
46     NaiveAlgo.totalBy1Hour = null;
47     NaiveAlgo.totalBy15Min = null;
48
49     NaiveAlgo.speedSum = 0D;
50 }
51
52 /**
53  * Determine the lane capacity in tunnels with cars per hour per lane
54  * @param item the next item
55  * @return the capacity
56  */
57 public static Double computeNextVehicle(StreamingDto item) {
58     if(Objects.equals(item.getItemType(), VEHICLE)) {
59         total = total + 1;
60         vehicleBy1Hour[indexVehicleBy1Hour] = vehicleBy1Hour[indexVehicleBy1Hour]
    ↪ + 1;
61         vehicleBy15Min[indexVehicleBy15Min] = vehicleBy15Min[indexVehicleBy15Min]
    ↪ + 1;
62
63         var vehicle = (VehicleDto)item;
64         speedSum = speedSum + vehicle.getSpeed();
65         if(Objects.equals(vehicle.getType(), TRUCK)) {
66             totalTrucks = totalTrucks + 1;
67         }
68     } else {
69         if(indexVehicleBy1Hour < 4) {
70             indexVehicleBy1Hour++;
71         } else {
72             vehicleBy1Hour[0] = vehicleBy1Hour[1];
73             vehicleBy1Hour[1] = vehicleBy1Hour[2];
74             vehicleBy1Hour[2] = vehicleBy1Hour[3];
75             vehicleBy1Hour[3] = vehicleBy1Hour[4];
76             vehicleBy1Hour[4] = 0D;
77             totalBy1Hour = vehicleBy1Hour[0] + vehicleBy1Hour[1] +
    ↪ vehicleBy1Hour[2] + vehicleBy1Hour[3];
78         }
79         if(indexVehicleBy15Min < 1) {

```

```

80         indexVehicleBy15Min++;
81     } else {
82         vehicleBy15Min[0] = vehicleBy15Min[1];
83         vehicleBy15Min[1] = 0D;
84         if(totalBy15Min == null || totalBy15Min < vehicleBy15Min[0]) {
85             totalBy15Min = vehicleBy15Min[0];
86         }
87     }
88 }
89
90 var percentage = totalTrucks / total;
91 var heavyVehicleFactor = 1.0 / (1.0 + percentage *
92     ↪ (matrix.getCoefficient(length, slopeGrade, percentage) - 1.0));
93 var averageSpeed = speedSum / total ;
94 var theoreticalCapacityPerTrafficLane = averageSpeed * 10.0 + 1200.0;
95
96 if(totalBy1Hour != null && totalBy15Min != null) {
97     var peakHourFactor = totalBy1Hour / (4 * totalBy15Min);
98     return theoreticalCapacityPerTrafficLane * peakHourFactor *
99     ↪ heavyVehicleFactor * habitualUseFactor;
100 } else {
101     return null;
102 }

```

### C.1.3 StreamQRE algorithm

```

</> File program 3: StreamQRE traffic density algorithm </>
1 package be.bruyere.vehiclestreaming.algo.tunnel;
2
3 import StreamQRE.*;
4 import be.bruyere.vehiclestreaming.service.dto.StreamingDto;
5 import be.bruyere.vehiclestreaming.service.dto.VehicleDto;
6 import be.bruyere.vehiclestreaming.service.dto.VehicleType;
7
8 import java.util.Objects;
9
10 import static be.bruyere.vehiclestreaming.service.dto.ItemType.END15;
11 import static be.bruyere.vehiclestreaming.service.dto.ItemType.VEHICLE;
12
13 public class StreamQREAlgo {
14
15     private static final EquivalenceFactorMatrix matrix = new
16     ↪ EquivalenceFactorMatrix();
17
18     /**
19      * Determine the lane capacity in tunnels with cars per hour per lane
20      * @param length the tunnel length
21      * @param slopeGrade the slope grade
22      * @param habitualUseFactor the habitual use factor of drivers
23      * @return the computing QRE
24      */
25     public static QReCombine<StreamingDto, Double, Double, Double>
26     ↪ computeLaneCapacity(
27         Double length,

```

```

26     Double slopeGrade,
27     Double habitualUseFactor
28 ) {
29     return new QReCombine<>(
30         theoreticalCapacityPerTrafficLane(),
31         new QReCombine<>(
32             peakHourFactor(),
33             heavyVehicleFactor(length, slopeGrade),
34             (PHF, HVF) -> PHF * HVF * habitualUseFactor
35         ),
36         (TC, R) -> TC * R
37     );
38 }
39
40 /**
41  * Compute the theoretical capacity in cars per hour per lane
42  * Use the average speed of vehicles in km/h
43  * @return the QRE
44  */
45 private static QReApply<StreamingDto, Double, Double>
46     ↪ theoreticalCapacityPerTrafficLane() {
47     var isAverageSpeed = StreamQREAlgo.averageSpeedOfVehicles();
48     return new QReApply<>(isAverageSpeed, x -> x * 10.0 + 1200.0);
49 }
50
51 /**
52  * Compute the average speed of vehicles
53  * @return the QRE
54  */
55 private static QReCombine<StreamingDto, Double, Double, Double>
56     ↪ averageSpeedOfVehicles() {
57     var is15MinToken = new QReAtomic<StreamingDto, Double>(x ->
58         ↪ Objects.equals(x.getItemType(), END15), x -> 0D);
59
60     var isVehicleSpeedToken = new QReAtomic<StreamingDto, Double>(x ->
61         ↪ Objects.equals(x.getItemType(), VEHICLE), x ->
62         ↪ ((VehicleDto)x).getSpeed().doubleValue());
63     var isSpeed = new QReElse<>(isVehicleSpeedToken, is15MinToken);
64     var isSpeedSum = new QReIter<>(isSpeed, 0D, Double::sum, x -> x);
65
66     var isVehicleCountToken = new QReAtomic<StreamingDto, Double>(x ->
67         ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1D);
68     var isCount = new QReElse<>(isVehicleCountToken, is15MinToken);
69     var isCountSum = new QReIter<>(isCount, 0D, Double::sum, x -> x);
70
71     return new QReCombine<>(isSpeedSum, isCountSum, (x, y) -> x/y);
72 }
73
74 /**
75  * Compute the peak hour factor (PHF), which represents the relationship of
76  * ↪ the hourly intensity
77  * ↪ in capacity divided by four times the maximum number of vehicles in a
78  * ↪ period of
79  * ↪ fifteen minutes during peak hour.
80  * @return the QRE
81  */

```



```

74 private static QReCombine<StreamingDto, Double, Double, Double> peakHourFactor()
75 ↪ {
76     var isVehicleToken = new QReAtomic<StreamingDto, Double>(x ->
77         ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1D);
78     var is15MinToken = new QReAtomic<StreamingDto, Double>(x ->
79         ↪ Objects.equals(x.getItemType(), END15), x -> 1D);
80
81     var sumOfVehicle = new QReIter<>(isVehicleToken, 0D, Double::sum, x -> x);
82     var sumOfVehiclesDuring15Min = new QReSplit<>(sumOfVehicle, is15MinToken, (x,
83         ↪ y) -> x);
84
85     var repeatOfSumOfVehiclesDuring15Min = new
86         ↪ QReIter<>(sumOfVehiclesDuring15Min, 0D, (x, y) -> x > y ? x : y, x -> x);
87     var sumOfVehiclesDuringLast15Min = new
88         ↪ QReSplit<>(repeatOfSumOfVehiclesDuring15Min, sumOfVehicle, (x, y) -> x >
89         ↪ y ? x : y);
90
91     var sumOfVehicleDuring1Hour = new QReWindow<>(sumOfVehiclesDuring15Min, 0D,
92         ↪ Double::sum, x -> x, 4);
93     var sumOfVehiclesDuringLastHour = new QReSplit<>(sumOfVehicleDuring1Hour,
94         ↪ sumOfVehicle, (x, y) -> x);
95
96     return new QReCombine<>(sumOfVehiclesDuringLastHour,
97         ↪ sumOfVehiclesDuringLast15Min, (x, y) -> x / (4 * y));
98 }
99
100 /**
101  * Compute the factor of heavy vehicles, which indicates the effect of the
102  * ↪ types of slow
103  * vehicles on the flow of light vehicles.
104  * It depends on the percentage of heavy vehicles
105  * @param length the tunnel length
106  * @param slopeGrade the slope grade
107  * @return the QRE
108  */
109 private static QReApply<StreamingDto, Double, Double> heavyVehicleFactor(
110     Double length,
111     Double slopeGrade
112 ) {
113     return new QReApply<>(percentageOfTrucks(), x -> 1.0 / (1.0 + x *
114         ↪ (matrix.getCoefficient(length, slopeGrade, x) - 1.0)));
115 }
116
117 /**
118  * Compute the percentage of trucks
119  * ↪ @return the QRE
120  */
121 private static QReCombine<StreamingDto, Long, Long, Double> percentageOfTrucks()
122 ↪ {
123     var is15MinToken = new QReAtomic<StreamingDto, Long>(x ->
124         ↪ Objects.equals(x.getItemType(), END15), x -> 0L);
125     var isVehicleToken = new QReAtomic<StreamingDto, Long>(x ->
126         ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1L);
127     var isVehicleValue = new QReElse<>(isVehicleToken, is15MinToken);
128     var isVehicleSum = new QReIter<>(isVehicleValue, 0L, Long::sum, x -> x);
129 }

```

```

115     var isTruckToken = new QReAtomic<StreamingDto,Long>(x ->
        ↪ Objects.equals(x.getItemType(),VEHICLE) && ((VehicleDto)x).getType() ==
        ↪ VehicleType.TRUCK, x -> 1L);
116     var isNotTruckToken = new QReAtomic<StreamingDto,Long>(x ->
        ↪ !Objects.equals(x.getItemType(),VEHICLE) || ((VehicleDto)x).getType() !=
        ↪ VehicleType.TRUCK, x -> 0L);
117     var isTruckValue = new QReElse<>(isTruckToken, isNotTruckToken);
118     var isTruckSum = new QReIter<>(isTruckValue, 0L, Long::sum, x -> x);
119
120     return new QReCombine<>(isTruckSum, isVehicleSum, (x, y) ->
        ↪ x.doubleValue()/y.doubleValue());
121 }
122 }

```

### C.1.4 LazyQRE algorithm

```

</> File program 4: LazyQRE traffic density algorithm </>
1 package be.bruyere.vehiclestreaming.algo.tunnel;
2
3 import be.bruyere.romain.qre.*;
4 import be.bruyere.vehiclestreaming.service.dto.StreamingDto;
5 import be.bruyere.vehiclestreaming.service.dto.VehicleDto;
6 import be.bruyere.vehiclestreaming.service.dto.VehicleType;
7
8 import java.util.Objects;
9
10 import static be.bruyere.vehiclestreaming.service.dto.ItemType.END15;
11 import static be.bruyere.vehiclestreaming.service.dto.ItemType.VEHICLE;
12
13 public class LazyQREAlgo {
14
15     private static final EquivalenceFactorMatrix matrix = new
        ↪ EquivalenceFactorMatrix();
16
17     /**
18      * Determine the lane capacity in tunnels with cars per hour per lane
19      * @param length the tunnel length
20      * @param slopeGrade the slope grade
21      * @param habitualUseFactor the habitual use factor of drivers
22      * @return the computing QRE
23      */
24     public static Combine3QRE<StreamingDto, Double, Double, Double, Double>
        ↪ computeLaneCapacity(
25         Double length,
26         Double slopeGrade,
27         Double habitualUseFactor
28     ) {
29         return new Combine3QRE<>(
30             theoreticalCapacityPerTrafficLane(),
31             peakHourFactor(),
32             heavyVehicleFactor(length, slopeGrade),
33             (TC, PHF, HVF) -> TC * PHF * HVF * habitualUseFactor
34         );
35     }
36
37     /**

```

```

38      * Compute the theoretical capacity in cars per hour per lane
39      * Use the average speed of vehicles in km/h
40      * @return the QRE
41      */
42  private static ApplyQRE<StreamingDto, Double, Double>
43      ↪ theoreticalCapacityPerTrafficLane() {
44      var isAverageSpeed = LazyQREAlgo.averageSpeedOfVehicles();
45      return new ApplyQRE<>(isAverageSpeed, x -> x * 10.0 + 1200.0);
46  }
47
48  /**
49   * Compute the average speed of vehicles
50   * @return the QRE
51   */
52  private static CombineQRE<StreamingDto, Double, Double, Double>
53      ↪ averageSpeedOfVehicles() {
54      var is15MinToken = new AtomQRE<StreamingDto, Double>(x ->
55      ↪ Objects.equals(x.getItemType(), END15), x -> 0D);
56
57      var isVehicleSpeedToken = new AtomQRE<StreamingDto, Double>(x ->
58      ↪ Objects.equals(x.getItemType(), VEHICLE), x ->
59      ↪ ((VehicleDto)x).getSpeed().doubleValue());
60      var isSpeed = new ElseQRE<>(isVehicleSpeedToken, is15MinToken);
61      var isSpeedSum = new IterQRE<>(isSpeed, 0D, Double::sum, x -> x);
62
63      var isVehicleCountToken = new AtomQRE<StreamingDto, Double>(x ->
64      ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1D);
65      var isCount = new ElseQRE<>(isVehicleCountToken, is15MinToken);
66      var isCountSum = new IterQRE<>(isCount, 0D, Double::sum, x -> x);
67
68      return new CombineQRE<>(isSpeedSum, isCountSum, (x, y) -> x/y);
69  }
70
71  /**
72   * Compute the peak hour factor (PHF), which represents the relationship of
73   ↪ the hourly intensity
74   ↪ * in capacity divided by four times the maximum number of vehicles in a
75   ↪ period of
76   ↪ * fifteen minutes during peak hour.
77   * @return the QRE
78   */
79  public static CombineQRE<StreamingDto, Double, Double, Double> peakHourFactor() {
80      var isVehicleToken = new AtomQRE<StreamingDto, Double>(x ->
81      ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1D);
82      var is15MinToken = new AtomQRE<StreamingDto, Double>(x ->
83      ↪ Objects.equals(x.getItemType(), END15), x -> 1D);
84
85      var sumOfVehicle = new IterQRE<>(isVehicleToken, 0D, Double::sum, x -> x);
86      var sumOfVehiclesDuring15Min = new SplitQRE<>(sumOfVehicle, is15MinToken, (x,
87      ↪ y) -> x, x -> x);
88
89      var repeatOfSumOfVehiclesDuring15Min = new
90      ↪ IterQRE<>(sumOfVehiclesDuring15Min, 0D, (x,y) -> x > y ? x : y, x -> x);
91      var sumOfVehiclesDuringLast15Min = new
92      ↪ SplitQRE<>(repeatOfSumOfVehiclesDuring15Min, sumOfVehicle, (x, y) -> x >
93      ↪ y ? x : y, x -> x);

```

```

81     var sumOfVehicleDuring1Hour = new WindowQRE<>(sumOfVehiclesDuring15Min, 0D,
82         ↪ Double::sum, x -> x, 4);
83     var sumOfVehiclesDuringLastHour = new SplitQRE<>(sumOfVehicleDuring1Hour,
84         ↪ sumOfVehicle, (x, y) -> x, x -> x);
85
86     return new CombineQRE<>(sumOfVehiclesDuringLastHour,
87         ↪ sumOfVehiclesDuringLast15Min, (x, y) -> x / (4 * y));
88 }
89
90 /**
91  * Compute the factor of heavy vehicles, which indicates the effect of the
92  * types of slow
93  * ↪ vehicles on the flow of light vehicles.
94  * It depends on the percentage of heavy vehicles
95  * @param length the tunnel length
96  * @param slopeGrade the slope grade
97  * @return the QRE
98  */
99 public static ApplyQRE<StreamingDto, Double, Double> heavyVehicleFactor(
100     Double length,
101     Double slopeGrade
102 ) {
103     return new ApplyQRE<>(percentageOfTrucks(), x -> 1.0 / (1.0 + x *
104         ↪ (matrix.getCoefficient(length, slopeGrade, x) - 1.0)));
105 }
106
107 /**
108  * Compute the percentage of trucks
109  * @return the QRE
110  */
111 public static CombineQRE<StreamingDto, Double, Long, Long> percentageOfTrucks() {
112     var is15MinToken = new AtomQRE<StreamingDto, Long>(x ->
113         ↪ Objects.equals(x.getItemType(), END15), x -> 0L);
114     var isVehicleToken = new AtomQRE<StreamingDto, Long>(x ->
115         ↪ Objects.equals(x.getItemType(), VEHICLE), x -> 1L);
116     var isVehicle = new ElseQRE<>(isVehicleToken, is15MinToken);
117     var isVehicleSum = new IterQRE<>(isVehicle, 0L, Long::sum, x -> x);
118
119     var isTruckToken = new AtomQRE<StreamingDto, Long>(x ->
120         ↪ Objects.equals(x.getItemType(), VEHICLE) && ((VehicleDto)x).getType() ==
121         ↪ VehicleType.TRUCK, x -> 1L);
122     var isNotTruckToken = new AtomQRE<StreamingDto, Long>(x ->
123         ↪ !Objects.equals(x.getItemType(), VEHICLE) || ((VehicleDto)x).getType() !=
124         ↪ VehicleType.TRUCK, x -> 0L);
125     var isTruck = new ElseQRE<>(isTruckToken, isNotTruckToken);
126     var isTruckSum = new IterQRE<>(isTruck, 0L, Long::sum, x -> x);
127
128     return new CombineQRE<>(isTruckSum, isVehicleSum, (x, y) ->
129         ↪ x.doubleValue()/y.doubleValue());
130 }
131 }

```

## C.2 Accident monitoring

These algorithms were designed to calculate the average vehicle speed of the 10 vehicles observed before each accident on an incoming stream. The algorithms simulate having to encrypt/decrypt data before it can be read. The encryption simulator is in the first file. The source code specific to these algo-

gorithms can be downloaded from this url, <https://github.com/rombru/VehicleStreamingApp/tree/main/backend/src/main/java/be/bruyere/vehiclestreaming/algo/accident>.

### C.2.1 Encryption simulation

```

</> File program 1: Encryption simulation </>
1  package be.bruyere.vehiclestreaming.algo.accident;
2
3  import javax.crypto.*;
4  import javax.crypto.spec.PBEKeySpec;
5  import javax.crypto.spec.SecretKeySpec;
6  import java.security.InvalidKeyException;
7  import java.security.NoSuchAlgorithmException;
8  import java.security.SecureRandom;
9  import java.security.spec.InvalidKeySpecException;
10 import java.security.spec.KeySpec;
11
12 public class GenerateEncryptionSimulation {
13     public static void generate() {
14         try {
15             SecureRandom random = new SecureRandom();
16             byte[] salt = new byte[16];
17             random.nextBytes(salt);
18
19             KeySpec spec = new PBEKeySpec("password".toCharArray(), salt, 65536, 256);
20             ↪ // AES-256
21             SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
22             byte[] key = f.generateSecret(spec).getEncoded();
23
24             String message = "message";
25
26             Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
27             SecretKey secretKey = new SecretKeySpec(key, "AES");
28             cipher.init(Cipher.ENCRYPT_MODE, secretKey);
29             cipher.doFinal(message.getBytes());
30         } catch (NoSuchAlgorithmException | NoSuchPaddingException |
31             ↪ InvalidKeyException | IllegalBlockSizeException | BadPaddingException |
32             ↪ InvalidKeySpecException e) {
33             e.printStackTrace();
34         }
35     }
36 }

```

### C.2.2 StreamQRE algorithm

```

</> File program 2: StreamQRE accident algorithm </>
1  package be.bruyere.vehiclestreaming.algo.accident;
2
3  import StreamQRE.*;
4  import be.bruyere.vehiclestreaming.service.dto.StreamingDto;
5  import be.bruyere.vehiclestreaming.service.dto.VehicleDto;
6
7  import java.util.Objects;
8
9  import static be.bruyere.vehiclestreaming.service.dto.ItemType.ACCIDENT;

```

```

10 import static be.bruyere.vehiclestreaming.service.dto.ItemType.VEHICLE;
11
12 public class StreamQREAccidentAlgo {
13
14     public static QReSplit<StreamingDto, Double, Double, Double>
15         ↪ averageSpeedOfVehiclesBeforeLastAccident() {
16         var isAccidentToken = new QReAtomic<StreamingDto,Double>(x ->
17             ↪ Objects.equals(x.getItemType(),ACCIDENT), x -> 0D);
18         var isVehicleToken = new QReAtomic<StreamingDto,VehicleDto>(x ->
19             ↪ Objects.equals(x.getItemType(),VEHICLE), x -> (VehicleDto) x);
20         var isVehicleSpeedToken = new QReApply<>(isVehicleToken, x -> {
21             GenerateEncryptionSimulation.generate();
22             return x.getSpeed().doubleValue();
23         });
24         var isVehicleSpeedOrAccidentToken = new
25             ↪ QReElse<>(isAccidentToken,isVehicleSpeedToken);
26         var is10ItemsSpeedSum = new QReWindow<>(isVehicleSpeedOrAccidentToken, 0D,
27             ↪ Double::sum, x -> x, 10);
28         var is10ItemsAvgSpeedBeforeAccident = new QReSplit<>(is10ItemsSpeedSum,
29             ↪ isAccidentToken, (x,y) -> x/10D);
30         var isVehicleSpeedSum = new QReIter<>(isVehicleToken, 0D, (x,y) -> 0D, x -> x);
31         return new QReSplit<>(is10ItemsAvgSpeedBeforeAccident, isVehicleSpeedSum, (x,
32             ↪ y) -> x);
33     }
34 }

```

### C.2.3 LazyQRE algorithm

```

</> File program 3: LazyQRE accident algorithm </>
1 package be.bruyere.vehiclestreaming.algo.accident;
2
3 import be.bruyere.romain.qre.*;
4 import be.bruyere.vehiclestreaming.service.dto.StreamingDto;
5 import be.bruyere.vehiclestreaming.service.dto.VehicleDto;
6
7 import java.util.Objects;
8
9 import static be.bruyere.vehiclestreaming.service.dto.ItemType.ACCIDENT;
10 import static be.bruyere.vehiclestreaming.service.dto.ItemType.VEHICLE;
11
12 public class LazyQREAccidentAlgo {
13
14     public static SplitQRE<StreamingDto, Double, Double, Double, Double>
15         ↪ averageSpeedOfVehiclesBeforeLastAccident() {
16         var isAccidentToken = new AtomQRE<StreamingDto,Double>(x ->
17             ↪ Objects.equals(x.getItemType(),ACCIDENT), x -> 0D);
18         var isVehicleToken = new AtomQRE<StreamingDto,VehicleDto>(x ->
19             ↪ Objects.equals(x.getItemType(),VEHICLE), x -> (VehicleDto) x);
20         var isVehicleSpeedToken = new ApplyQRE<>(isVehicleToken, x -> {
21             GenerateEncryptionSimulation.generate();
22             return x.getSpeed().doubleValue();
23         });
24         var isVehicleSpeedOrAccidentToken = new
25             ↪ ElseQRE<>(isAccidentToken,isVehicleSpeedToken);
26         var is10ItemsSpeedSum = new WindowQRE<>(isVehicleSpeedOrAccidentToken, 0D,
27             ↪ Double::sum, x -> x, 10);

```

```
23     var is10ItemsAvgSpeedBeforeAccident = new SplitQRE<>(is10ItemsSpeedSum,  
24         ↪ isAccidentToken, (x,y) -> x/10D, x -> x);  
24     var isVehicleSpeedSum = new IterQRE<>(isVehicleToken, OD, (x,y) -> OD, x -> x);  
25     return new SplitQRE<>(is10ItemsAvgSpeedBeforeAccident, isVehicleSpeedSum,  
26         ↪ (x,y) -> x, x -> x);  
26 }  
27 }
```





## Appendix D

# Vehicle Stream Simulation App

A simulation application was created in order to produce a data stream and to test the algorithms, which was appended in the previous appendix. The application is composed of a frontend and a backend.

The frontend is a form to encode the parameters of generation of the stream and a map to display a live representation of the stream on a route. The programming language used is the Angular javascript framework (<https://angular.io/>) with the OpenLayers library (<https://openlayers.org/>).

The backend is a Java web server written with the Spring Boot framework (<https://spring.io/projects/spring-boot>). It receives requests from the frontend. Depending on the requests, it starts the algorithms, modifies the received stream to add temporal data to it, and returns the results.

The entire backend and frontend code for all algorithms would have been too large to be added as an appendix. Only the logic used to interface with the density calculation algorithm using LazyQRE, from Appendix C.1.4, has been appended. The complete source code of the simulation application, as well as the algorithms, can be downloaded at this url, <https://github.com/rombru/VehicleStreamingApp>.

### D.1 Frontend

```
</> File program 1: Form component for specifying the stream generation parameters </>
1  import {Component, OnInit} from '@angular/core';
2  import {FormBuilder, FormGroup} from "@angular/forms";
3  import {ParametersModel} from "../../models/parameters.model";
4  import {AppState} from "../app.state";
5  import {AlgoTypeEnum} from "../../models/algo-type.enum";
6
7  @Component({
8    selector: 'app-form',
9    templateUrl: './form.component.html',
10   styleUrls: ['./form.component.scss']
11 })
12 export class FormComponent implements OnInit {
13
14   public readonly AlgoTypeEnum = AlgoTypeEnum;
15   public readonly Color = "primary";
16
17   public form: FormGroup;
18   public result: number;
19
20   constructor(
21     private readonly fb: FormBuilder,
22     private readonly appState: AppState,
```

```

23   ) {
24       this.form = this.fb.group({
25           algo: [AlgoTypeEnum.STREAMQRE],
26           density: [100],
27           acceleration: [1],
28           slopeGrade: [2],
29           length: [4],
30           habitualUseFactor: [0.90],
31           rateCar: [0.7],
32           meanSpeedCar: [110],
33           sdSpeedCar: [10],
34           rateTruck: [0.2],
35           meanSpeedTruck: [90],
36           sdSpeedTruck: [7],
37           rateMotorbike: [0.1],
38           meanSpeedMotorbike: [110],
39           sdSpeedMotorbike: [10],
40       });
41   }
42
43   ngOnInit(): void {}
44
45   onSubmit(): void {
46       const value : ParametersModel = this.form.getRawValue();
47       this.appState.setServiceType(value.algo);
48       this.appState.setParameters(value);
49   }
50
51   onReset(): void {
52       this.form.enable();
53       this.appState.setStop(true);
54   }
55
56   onGetResult(): void {
57       this.appState.getOutput().subscribe(o => this.result = o);
58   }
59
60 }

```

&lt;/&gt;

File program 2: Singleton maintaining the state of the application

&lt;/&gt;

```

1  import {Injectable} from '@angular/core';
2  import {HttpClient} from "@angular/common/http";
3  import {BehaviorSubject, Observable} from "rxjs";
4  import {first} from "rxjs/operators";
5  import {VehicleModel} from "../models/vehicle.model";
6  import {ParametersModel} from "../models/parameters.model";
7  import {VehicleService} from "../services/vehicle.service";
8  import {LazyVehicleService} from "../services/lazy-vehicle.service";
9  import {AbstractVehicleService} from "../services/abstract-vehicle.service";
10 import {AlgoTypeEnum} from "../models/algo-type.enum";
11 import {NaiveVehicleService} from "../services/naive-vehicle.service";
12 import {VehicleAccidentService} from "../services/vehicle-accident.service";
13 import {LazyVehicleAccidentService} from
    ↪  "../services/lazy-vehicle-accident.service";
14
15 @Injectable({

```

```

16     providedIn: 'root'
17   })
18   export class AppState {
19
20     private readonly parameters = new BehaviorSubject<ParametersModel>(null);
21     public readonly parameters$ = this.parameters.asObservable();
22
23     private readonly stop = new BehaviorSubject<boolean>(null);
24     public readonly stop$ = this.stop.asObservable();
25
26     private service: AbstractVehicleService;
27
28     constructor(
29       private vehicleService: VehicleService,
30       private vehicleAccidentService: VehicleAccidentService,
31       private lazyVehicleService: LazyVehicleService,
32       private lazyVehicleAccidentService: LazyVehicleAccidentService,
33       private naiveVehicleService: NaiveVehicleService
34     ) {
35     }
36
37     public setServiceType(type: AlgoTypeEnum) {
38       switch (type) {
39         case AlgoTypeEnum.LAZYQRE:
40           this.service = this.lazyVehicleService;
41           break;
42         case AlgoTypeEnum.STREAMQRE:
43           this.service = this.vehicleService;
44           break;
45         case AlgoTypeEnum.NAIVE:
46           this.service = this.naiveVehicleService;
47           break;
48         case AlgoTypeEnum.STREAMQREACCIDENT:
49           this.service = this.vehicleAccidentService;
50           break;
51         case AlgoTypeEnum.LAZYQREACCIDENT:
52           this.service = this.lazyVehicleAccidentService;
53           break;
54       }
55     }
56
57     public setParameters(parameters: ParametersModel) {
58       this.parameters.next(parameters);
59     }
60
61     public setStop(stop: boolean) {
62       this.stop.next(stop);
63     }
64
65     public getOutput(): Observable<number> {
66       return this.service.getOutput();
67     }
68
69     public start(param: ParametersModel): Observable<void> {
70       return this.service.start(param);
71     }
72

```

```

73 public next(vehicle: VehicleModel): Observable<void> {
74     return this.service.next(vehicle);
75 }
76
77 public reset(): Observable<void> {
78     return this.service.reset();
79 }
80 }

```

</> File program 3: Interface for communicating with the web server </>

```

1  import {Injectable} from '@angular/core';
2  import {HttpClient} from '@angular/common/http';
3  import {Observable} from "rxjs";
4  import {first} from "rxjs/operators";
5  import {VehicleModel} from "../models/vehicle.model";
6  import {AbstractVehicleService} from "../abstract-vehicle.service";
7  import {ParametersModel} from "../models/parameters.model";
8
9  @Injectable({
10     providedIn: 'root'
11 })
12 export class LazyVehicleService extends AbstractVehicleService {
13
14     constructor(
15         private readonly http: HttpClient
16     ) {
17         super()
18     }
19
20     public getOutput(): Observable<number> {
21         return this.http.get<number>('/api/vehicle/lazy/output').pipe(first());
22     }
23
24     public start(parameter: ParametersModel): Observable<void> {
25         return this.http.post<void>('/api/vehicle/lazy/start', parameter).pipe(first());
26     }
27
28     public next(vehicle: VehicleModel): Observable<void> {
29         return this.http.post<void>('/api/vehicle/lazy/next', vehicle).pipe(first());
30     }
31
32     public reset(): Observable<void> {
33         return this.http.get<void>('/api/vehicle/lazy/reset').pipe(first());
34     }
35 }

```

## D.2 Backend

</> File program 1: Controller receiving the requests </>

```

1  package be.bruyere.vehiclestreaming.web.rest.tunnel;
2
3  import be.bruyere.vehiclestreaming.service.tunnel.LazyVehicleService;
4  import be.bruyere.vehiclestreaming.service.dto.ParameterDto;
5  import be.bruyere.vehiclestreaming.service.dto.VehicleDto;

```

```

6  import lombok.RequiredArgsConstructor;
7  import org.springframework.http.ResponseEntity;
8  import org.springframework.web.bind.annotation.*;
9
10 @RestController
11 @RequestMapping("/api/vehicle/lazy")
12 @RequiredArgsConstructor
13 public class LazyVehicleResource {
14
15     private final LazyVehicleService lazyVehicleService;
16
17     @GetMapping("/output")
18     public ResponseEntity<Double> getOutput() {
19         return ResponseEntity.ok(lazyVehicleService.getOutput());
20     }
21
22     @PostMapping("/start")
23     public ResponseEntity<Void> start(@RequestBody ParameterDto parameter){
24         lazyVehicleService.start(parameter);
25         return ResponseEntity.ok().build();
26     }
27
28     @PostMapping("/next")
29     public ResponseEntity<Void> nextVehicle(@RequestBody VehicleDto vehicle) {
30         lazyVehicleService.next(vehicle);
31         return ResponseEntity.ok().build();
32     }
33
34     @GetMapping("/reset")
35     public ResponseEntity<Void> reset() {
36         lazyVehicleService.reset();
37         return ResponseEntity.ok().build();
38     }
39 }

```

</> File program 2: *Component managing the interactions with the algorithm* </>

```

1  package be.bruyere.vehiclestreaming.service.tunnel;
2
3  import be.bruyere.romain.eval.EvalExtension;
4  import be.bruyere.vehiclestreaming.algo.accident.LazyQREAccidentAlgo;
5  import be.bruyere.vehiclestreaming.algo.tunnel.LazyQREAlgo;
6  import be.bruyere.vehiclestreaming.service.ScheduleTaskService;
7  import be.bruyere.vehiclestreaming.service.dto.*;
8  import lombok.RequiredArgsConstructor;
9  import org.springframework.stereotype.Service;
10
11 import java.time.Duration;
12 import java.time.Instant;
13 import java.time.temporal.ChronoUnit;
14
15 @Service
16 @RequiredArgsConstructor
17 public class LazyVehicleService {
18
19     private final ScheduleTaskService scheduleTaskService;
20

```

```

21 private EvalExtension.StartEval<StreamingDto, Double> eval;
22
23 private void configure(ParameterDto parameter) {
24     System.out.println("Configured");
25     eval = LazyQREAlgo.computeLaneCapacity(
26         parameter.getLength(),
27         parameter.getSlopeGrade(),
28         parameter.getHabitualUseFactor()).start();
29 }
30
31 public Double getOutput() {
32     if(eval.result().nonEmpty()) {
33         return eval.result().get();
34     } else {
35         return null;
36     }
37 }
38
39 public void start(ParameterDto parameters) {
40     this.configure(parameters);
41     this.startScheduler();
42 }
43
44 public void next(VehicleDto vehicle) {
45     this.eval = eval.next(vehicle);
46 }
47
48 public void reset() {
49     eval = null;
50     scheduleTaskService.removeTaskFromScheduler(1);
51 }
52
53 public void startScheduler() {
54     scheduleTaskService.addTaskToScheduler(
55         1,
56         () -> {
57             System.out.println("Minute");
58             eval = eval.next(new TimerDto());
59         },
60         Instant.now().plus(15, ChronoUnit.SECONDS),
61         Duration.ofSeconds(15));
62 }
63 }

```

</> File program 3: Interface implemented by each item of the stream </>

```

1 package be.bruyere.vehiclestreaming.service.dto;
2
3 public interface StreamingDto {
4     ItemType getItemType();
5 }

```

</> File program 4: Enumeration of streaming types </>

```

1 package be.bruyere.vehiclestreaming.service.dto;
2
3 import com.fasterxml.jackson.annotation.JsonCreator;

```

```

4  import com.fasterxml.jackson.annotation.JsonValue;
5  import lombok.AllArgsConstructor;
6  import lombok.Getter;
7
8  import java.util.stream.Stream;
9
10 @Getter(onMethod = @__({@JsonValue}))
11 @AllArgsConstructor
12 public enum ItemType {
13     ACCIDENT("ACCIDENT"),
14     END15("TIMER"),
15     VEHICLE("VEHICLE");
16
17     private String value;
18
19     @JsonCreator
20     public static ItemType decode(final String value) {
21         return Stream.of(ItemType.values())
22             .filter(targetEnum -> targetEnum.value.equals(value))
23             .findFirst()
24             .orElse(null);
25     }
26 }

```

&lt;/&gt;

File program 5: *Streaming item representing a vehicle*

&lt;/&gt;

```

1  package be.bruyere.vehiclestreaming.service.dto;
2
3  import lombok.*;
4
5  @AllArgsConstructor
6  @NoArgsConstructor
7  @Getter
8  @Setter
9  @ToString
10 public class VehicleDto implements StreamingDto {
11     private Long id;
12     private Long speed;
13     private VehicleType type;
14
15     @Override
16     public ItemType getItemType() {
17         return ItemType.VEHICLE;
18     }
19 }

```

&lt;/&gt;

File program 6: *Enumeration of vehicle types*

&lt;/&gt;

```

1  package be.bruyere.vehiclestreaming.service.dto;
2
3  import com.fasterxml.jackson.annotation.JsonCreator;
4  import com.fasterxml.jackson.annotation.JsonValue;
5  import lombok.AllArgsConstructor;
6  import lombok.Getter;
7
8  import java.util.stream.Stream;
9

```

```

10 @Getter(onMethod = @__({@JsonValue}))
11 @AllArgsConstructor
12 public enum VehicleType {
13     CAR("C"),
14     TRUCK("T"),
15     MOTORBIKE("M");
16
17     private String value;
18
19     @JsonCreator
20     public static VehicleType decode(final String value) {
21         return Stream.of(VehicleType.values())
22             .filter(targetEnum -> targetEnum.value.equals(value))
23             .findFirst()
24             .orElse(null);
25     }
26 }

```

</> File program 7: Streaming item representing a new time period signal </>

```

1 package be.bruyere.vehiclestreaming.service.dto;
2
3 import lombok.*;
4
5 @NoArgsConstructor
6 @Getter
7 @Setter
8 @ToString
9 public class TimerDto implements StreamingDto {
10     @Override
11     public ItemType getItemType() {
12         return ItemType.END15;
13     }
14 }

```

</> File program 8: Configuration streaming object </>

```

1 package be.bruyere.vehiclestreaming.service.dto;
2
3 import lombok.*;
4
5 @AllArgsConstructor
6 @NoArgsConstructor
7 @Getter
8 @Setter
9 @ToString
10 public class ParameterDto {
11     private Double length;
12     private Double slopeGrade;
13     private Double habitualUseFactor;
14 }

```