

DATA STREAMS IN SPREADSHEETS WITH REACTIVE EXTENSIONS

by

Lars Willems



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

Student number:	4256247
Master program:	Computer Science
Specialization:	Software Technology
Thesis committee:	Prof. dr. H.J.M. Meijer, TU Delft Dr. ir. F.F.J. Hermans, TU Delft Prof. dr. A. van Deursen, TU Delft
Thesis defense:	31 May 2016

Preface

This document represents the thesis report that was written as the final part of my master's programme in Computer Science at the Delft University of Technology. The research project was supervised by Prof. dr. H.J.M. Meijer and was conducted partially internally at the Software Engineering Research Group at the Delft University of Technology and partially at Applied Duality Inc.

The amount of collected data increases exponentially and this exponential growth comes with a demand to analyze big volumes of data and to process the data fast and in real time. Business analysts have the expertise in the domain of the desired stream processing applications, but often lack the technical skills to create these kind of applications. Advanced software engineering skills are required in the development of these applications. This thesis aims to make real time stream processing available for the millions of spreadsheets users around the world. The thesis presents a novel spreadsheet model, that turns a spreadsheet into an effective integrated development environment for stream processing. The spreadsheet model is built upon the reactive extensions framework, a library for asynchronous programming with push based data streams. The spreadsheet model is implemented both as add-in for Microsoft Excel and as underlying part of a newly created spreadsheet application that was written in Scala.

Acknowledgements

First of all, I would like to express my gratitude to Erik Meijer for supervising the project. His guidance and expertise were a great help in the process of conducting the research and writing the report. Besides my supervisor, I would like to thank Felienne Hermans and Arie van Deursen for participating in the thesis committee. And last but not least, I want to thank my fellow students, who were working on their thesis projects at the same time, for the inspiring and motivating discussions and their insightful comments and feedback.

Lars Willems
Delft, the Netherlands
May 24, 2016

TABLE OF CONTENTS

1	Introduction.....	5
1.1	The shortcoming of a Spreadsheet.....	5
1.2	Contributions.....	6
1.3	Overview	6
2	Introduction to Reactive Extensions	7
2.1	Push vs. Pull	7
2.2	Observables, Observers and Subjects.....	7
2.3	Operators	8
3	A model for stream processing in spreadsheets.....	10
3.1	Internal Data Streams.....	10
3.2	External Data Streams	13
4	ExcelStreams	16
4.1	Model	16
4.2	List of Rx functions.....	21
4.3	Lists in Excel	22
4.4	Data Streams and Server Sent Events	24
4.5	Examples	26
4.6	Limitations.....	30
5	StreamSheets.....	33
5.1	Advantages and disadvantages.....	33
5.2	Model	34
5.3	Parsing Formulas.....	36
5.4	Evaluating Formulas.....	37
5.5	List of Functions.....	39
5.6	Charts and Lists	40
5.7	Example	41
5.8	External Data Streams and Published Sheets	42
6	Google Sheets	44
6.1	Google Sheets API.....	44
6.2	Google Sheets Add-On.....	45
7	Related Work.....	47
8	Conclusion and Future Work.....	48
	References.....	50

1 INTRODUCTION

Spreadsheets are among the most used programming tools in the world. They are used a lot, both in business and by individuals, for data storage and data analysis in a broad range of applications. Spreadsheet applications are integrated development environments that brought functional programming to the masses. They are easy to use and you don't have to be a professional programmer to be able to create a spreadsheet application. A great benefit of a spreadsheet is that the data and the logic of the program are not separated and that the spreadsheet provides continuous feedback to the user by showing the result of a formula in a cell instantly. Moreover, a spreadsheet program is a reactive program, that propagates value changes in a sheet directly to the dependent cells. Since spreadsheets are easy to use, a business analyst can quickly create an application without having to involve an IT specialist that has less expertise in the domain of the application. Even if a spreadsheet is not completely sufficient, it can be used to rapidly create a prototype that clarifies the requirements for the IT developers.

1.1 THE SHORTCOMING OF A SPREADSHEET

Spreadsheets are widely used to perform calculations and analysis on data stored in a two-dimensional grid of cells. These are some situations where a spreadsheet is an efficient tool to use:

- A teacher registers the grades of the students and calculates their averages.
- An accountant records all the income and expenses of a company and calculates the profit that the company made over the last months.
- A meteorologist keeps track of the temperature measurements for a given time and analyzes the data for long-term weather forecasts.

In all these examples, the data is first collected and then processed. Nowadays, big data applications are becoming more and more popular and the amount of available data is growing exponentially, not only because people store and share more data online, but also because more and more devices with many sensors are used to collect big volumes of data. This exponential growth increases the demand for real time stream processing. Real time stream processing is a programming paradigm used to analyze vast amounts of data and to process the data fast, so that a company can react directly to changing conditions or circumstances. Stream processing is a scalable way of analyzing data in motion. Operations and calculations are performed continuously as the data flows through the system. It also allows the user to include external data sources. Some examples of situations where one would benefit from stream processing are:

- Analyzing real time stock exchanges
- Predicting the temperature based on the measurements of a temperature sensor
- Analyzing real time traffic data and predicting congestions or travel times

Despite the reactive nature of spreadsheets, it is hard to implement these examples with today's spreadsheet applications. They are not suitable for real time stream processing, because there is no functionality to work with data streams and historical data is not preserved. Moreover, there is no simple way to connect a spreadsheet to live external data streams by only using spreadsheet formulas. For this kind of applications, a spreadsheet user needs to rely on IT specialists to build a complex solution that collects, combines and manipulates the data streams.

1.2 CONTRIBUTIONS

The goal of this research is to unify the prevalent model of spreadsheet applications with the stream processing paradigm using the Reactive Extensions framework (Rx). The main research question of this thesis project is:

"Can we develop a spreadsheet model with Rx that makes a spreadsheet application an effective IDE for real time stream processing?"

This research question is divided in the following three sub-questions:

1. How can we introduce higher order streams in spreadsheets, i.e. how can we represent data streams as native data types, so that they can be used as arguments to spreadsheet functions and so that they can be returned by spreadsheet functions?
2. How can we preserve historical data of a spreadsheet without storing large amounts of superfluous data or how can we add an extra time dimension to a two-dimensional tabular sheet?
3. How can we import external data streams live from the internet by only using spreadsheet formulas and without the need to continuously check for updates?

This report presents a novel spreadsheet model that provides an answer to the aforementioned questions. The foundation of this model is the Reactive Extensions framework, a library for composing asynchronous and event-based programs. A short introduction to the Reactive Extensions framework is given in chapter 2. Additionally, we explore to which extent this model is applicable in practice, both as an extension to two popular spreadsheet applications, Microsoft Excel and Google Sheets, and as underlying model for a newly created spreadsheet application. The result is an add-in for Microsoft Excel with the name ExcelStreams and a new spreadsheet application written in Scala named StreamSheets. Applying this model to Google Sheets turned out to be infeasible, for which a motivation is included in the report as well.

1.3 OVERVIEW

A short introduction to the Reactive Extensions framework on which the spreadsheet model is built is given in chapter 2. The conceptual spreadsheet model with support for stream processing is presented in chapter 3. Chapter 4 gives a description of ExcelStreams, an add-in that implements this spreadsheet model for Microsoft Excel. StreamSheets is a newly created application based on this spreadsheet model and is documented in chapter 5. Chapter 6 explains the difficulties of applying this model to Google Sheets. Chapter 7 gives an overview of other research efforts in the field of stream processing with spreadsheets and how it relates to the work performed in this thesis project. And finally chapter 8 provides a conclusion of the work and the opportunities for future research.

2 INTRODUCTION TO REACTIVE EXTENSIONS

Reactive Extensions (Rx) is a library for asynchronous programming with data streams that are represented as observable sequences.^[3] With the Rx library one can easily create data streams, combine and transform them with Rx operators and subscribe to these observable sequences to get notified of new events in the data stream. Rx was originally developed for .NET, but is now available for many other programming languages, including Java, Scala, Python and JavaScript.

2.1 PUSH VS. PULL

In Rx a data stream is represented as an observable. Observables can be regarded as the counterpart or dual of iterables. An iterable is pull based, which means that the consumer of the data sequence pulls the values from the producer. The consumer is in charge and decides when to retrieve the next value. When the consumer pulls a value from the producer the thread blocks until the producer returns the result, because the process to retrieve a value is synchronous. On the contrary, an observable is completely push based. Whenever new data is available, the producer pushes the new value directly to the consumer. The consumer does not poll the data source, but reacts to the data that it receives. This process can be either synchronous or asynchronous. The following table summarizes how the observable relates to the iterable.

	Iterable (pull based)	Observable (push based)
retrieve data	T next()	onNext(T)
discover error	throws Exception	onError(Exception)
complete	! hasNext()	onCompleted()

The table shows that the observable is an extension of the observer pattern from the Gang of Four. An observable notifies the consumer when new data is available with the `onNext` function, but in addition it notifies the consumer if an error occurs and when there is no more data available with the functions `onError` and `onCompleted` respectively.

2.2 OBSERVABLES, OBSERVERS AND SUBJECTS

The observable and the observer are the most essential objects in Rx. The observable represents a push based data stream that can be observed by zero or more observers. An observer subscribes to an observable that it is interested in to get notified of events that occur in the data stream. The observer implements the `onNext`, `onError` and `onCompleted` functions that may be called by the observable it is subscribed to. The Observable Contract^[4] states that an observable will call the `onNext` function zero or more times, optionally followed by a call to either the `onError` or `onCompleted` function. When an observer subscribes to an observable, a subscription object is returned, that can be used to unsubscribe from the observable when the observer is not interested anymore to receive new values or notifications.

The moment that an observable starts emitting its items depends on the observable. There are hot and cold observables. A hot observable starts emitting items directly from the moment that it is created. An observer that later subscribes to a hot observable will only receive the items emitted after the event of subscription. On the other hand, a cold observable waits until an observer subscribes before it starts publishing items. An observer that subscribes to a cold

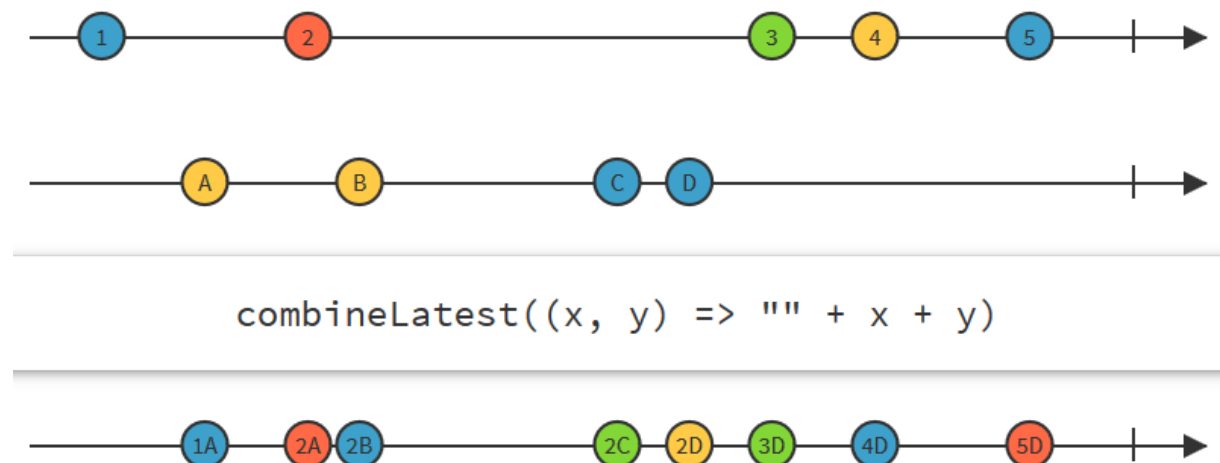
observable will always receive the complete sequence, regardless of the time of subscription. Values are shared among subscribers of a hot observable, but they are not among subscribers of a cold observable.

A subject is an object in Rx that acts both as an observer and as an observable. As observer a subject can subscribe to another observable. As observable it can reemit the observed items or emit new items by calling the `onNext`, `onError` and `onCompleted` methods of the subscribed observers. A subject makes a cold observable hot, since the cold observable starts to emit items at the moment that the subject subscribes to it. Nevertheless, there are four different types of subjects with different functionality:

- An observer that subscribes to a **PublishSubject** will only receive the items emitted after the event of subscription.
- A **ReplaySubject** stores all values that it has published and an observer receives all values, regardless of the moment of subscription.
- The **BehaviorSubject** is similar to the `ReplaySubject`, but only stores the last emitted value.
- An **AsyncSubject** emits only the last item and only at the moment that the source observable completes.

2.3 OPERATORS

The Reactive Extensions library provides a lot of operators that can be used to transform, filter or combine observables. `CombineLatest` is an example of an operator that will be used a lot in the spreadsheet model for stream processing. It combines two observables into one by applying a specified function.



This `combineLatest` operator takes two source observables and returns a new observable that concatenates the values of the source observables. When one of the source observables emits a new item, the operator combines it with the latest value from the other source observable and

the resulting observable emits the value returned by the concatenation function that was specified as argument to the `combineLatest` operator.

An example of a transforming operator is *map*, that transforms every emitted item by applying the specified function. *Filter* is an example of a filtering operator, that emits only the items from the source observable for which a given predicate function returns true. Since many operators return an observable themselves, multiple operators can also be applied in sequence. A complete list of Rx operators can be found on the ReactiveX website.^[3]

3 A MODEL FOR STREAM PROCESSING IN SPREADSHEETS

This chapter describes the principles of a novel spreadsheet model with support for stream processing. The implementation details of the model are explicated in chapter 4 and 5, which document the Excel add-in and a newly created spreadsheet application that are both based on this model.

3.1 INTERNAL DATA STREAMS

The starting point of this spreadsheet model is that a data stream is implemented as an observable from the Reactive Extensions library. To be able to process and perform calculations on these data streams, we want to apply some of the Rx operators in the context of a spreadsheet. Therefore, we should define spreadsheet functions that take observables as arguments, perform an Rx operation in the background, and return another observable as the result. In a spreadsheet the result of a function is stored in a cell. As a consequence a cell must represent an observable of values instead of a single value. Typically, an argument of a spreadsheet function can be replaced by a reference to another cell. With a cell representing an observable of values, this property still holds for functions that take observables as arguments.

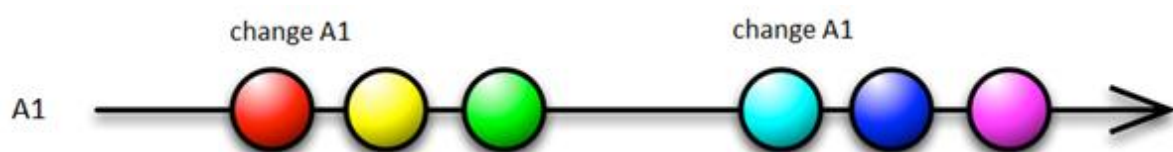
Let's assume every cell object stores an observable of values instead of a single value and we take the following example. First we enter a formula in cell A1 that returns an observable that emits three items. This observable is stored in the cell object of A1.



After the third item is emitted, we change the formula of cell A1. It returns another observable that replaces the previous one.

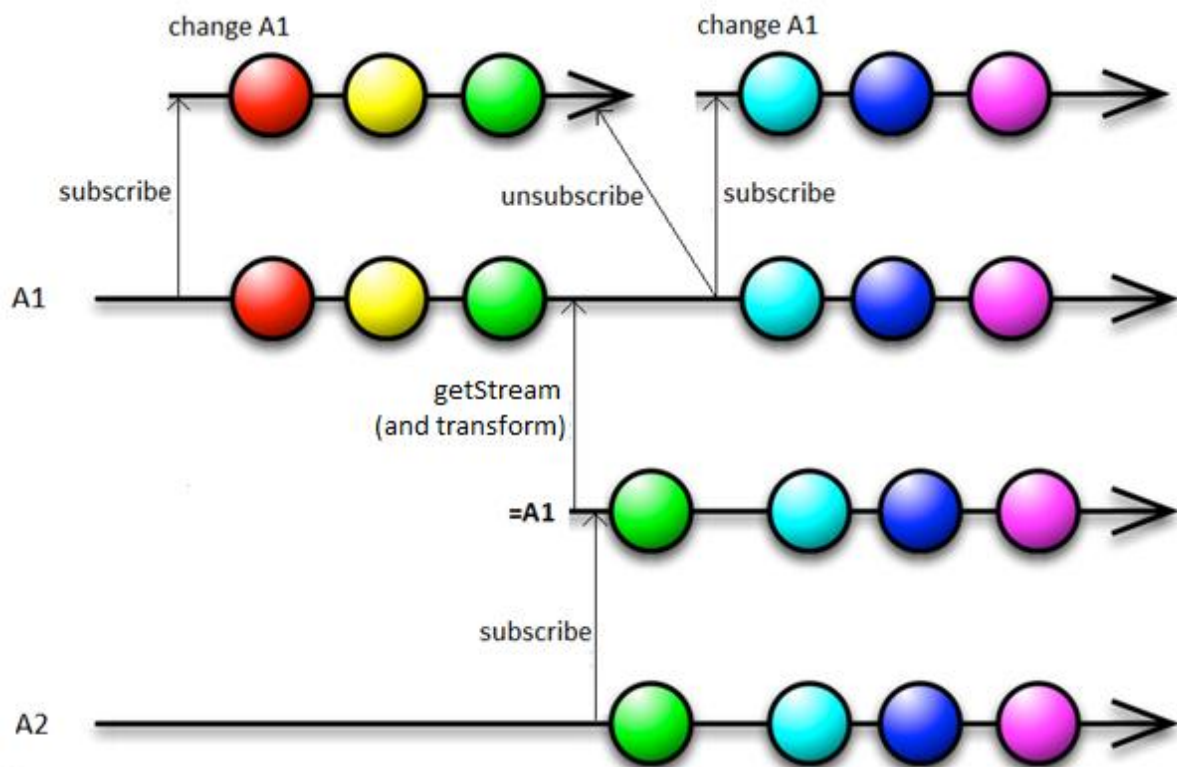


The downside of this design becomes clear if we have a cell that depends on the stream of A1. If cell A2 is a transformation of A1, an Rx operator is applied to the current observable of A1 and the resulting observable is stored for cell A2. However, the observable of A2 is not valid anymore if A1 changes, because A2 is a transformation of the old observable of A1. Hence, if the formula of a cell is changed, the dependent cells must be updated too. This would not be needed if A1 stored a value stream that contains all values over time in that cell as shown in the following figure:



In this situation, the dependent cells of A1 would not have to be re-evaluated when the formula of A1 changes, because the dependent cell is a transformation of the value stream of A1 instead of a transformation of the result of a specific formula. In this way, a formula change of a cell only affects the cell itself, but not its dependents. At the same time, this approach has the benefit that the history of a cell will not be lost, even if its formula changes.

A value stream of a cell, that contains all items in that cell over time, can be implemented with an Rx subject. It does not store the resulting observable of a formula, but the value stream of the cell subscribes to that observable. This process is shown in the following example.



1. The user changes the formula of cell A1, that results in an observable. The value stream of A1 subscribes to this observable and re-emits the items.
2. The user enters the formula **=A1** in cell A2. When the formula is evaluated, it retrieves the value stream of A1 as observable. No operations are applied to this stream, since the values should not be changed according to the formula. The value stream of A2 subscribes to the resulting observable.
3. The user changes the formula of cell A1, that results in an observable. The value stream of A1 unsubscribes from the previous observable and subscribes to the new observable.

This example shows that a formula change in cell A1 does not affect A2. Only A1 itself should unsubscribe from the previous observable and subscribe to the new one, but new values are automatically propagated to dependent cells.

The value stream of a cell is implemented as a subject. It acts as an observer when it subscribes to the result stream of a formula. Additionally, it is exposed as an observable, such that the value stream of a cell can be passed as argument to a spreadsheet function in another cell. A subject acts as a hot observable, which means that it starts emitting items from the moment that it is created. This is good, because we do not want to receive all historical values of a cell when we subscribe to its value stream. The most appropriate subject for the value stream is the `BehaviorSubject`, that starts with emitting the last published value. This conforms to the expected behavior of a spreadsheet. If a cell is set to copy the content of A1, it should not wait for A1 to be updated, but it should immediately start with the current value of A1, which is the last published value of that cell.

The value streams of all cells are potentially infinite. It is always possible to add a new value to a cell. The Observable Contract^[4] states that an observable (or subject) will call the `onNext` function zero or more times, optionally followed by a call to either the `onError` or `onCompleted` function. Therefore, when a value stream of a cell subscribes to an observable, it only re-emits the `onNext` events, but not the `onError` and `onCompleted` events, since that would indicate the end of the value stream.

Spreadsheet functions

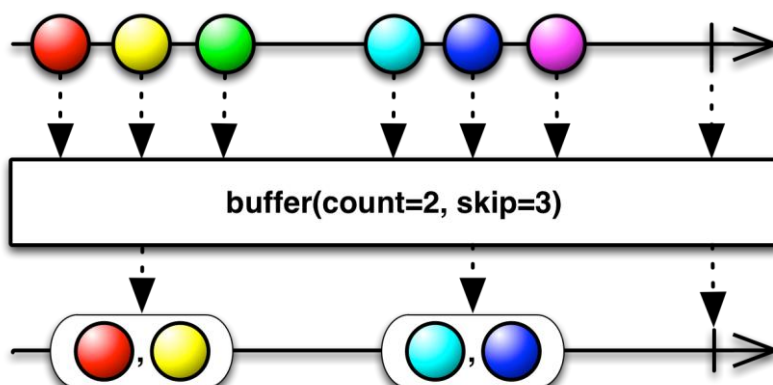
Spreadsheet functions take observables as arguments and return other observables as result. A function argument can be one of the following:

- A cell reference, referring to the value stream of another cell in the worksheet
- An application of a spreadsheet function
- A simple singular value, such as a text string, a number or a boolean

The first two options return observables of values and hence can be used directly as function arguments. The third option returns a single value, that must be first converted into an observable that emits that item with the *just* or *return* operator from Rx.

If a spreadsheet function takes more than one argument and it should combine the single elements emitted by the argument observables, it uses the *combineLatest* operator. This operator is illustrated in section 2.3. When one of the argument observables emits a new value, it is combined with the latest values from the other source observables and a new result is calculated. The way in which the result is calculated differs from function to function.

The history of a cell can be accessed by applying the Rx *buffer* operator to the value stream of the cell.



The buffer operator periodically collects items from a source observable and emits these collections as lists. The count argument decides how many elements should be collected in one bundle. The skip argument specifies after how many elements it should start a new bundle. When the skip argument is set to one, the buffer operator works as a sliding window, that continuously emits the last *count* items collected in a list. A spreadsheet should be able to store these lists in a cell and to perform some calculations on these lists. The main benefit of using the buffer operator to access the history of a cell, is that only those historical values of a cell are stored that are really used in the spreadsheet. At the moment that the spreadsheet user enters a formula that uses the buffer operator, the program starts collecting the lists. When more values become available, the observable emits a new list and the old list is no longer stored.

3.2 EXTERNAL DATA STREAMS

A user should not only be able to use data streams from other cells in the spreadsheet, but it should be possible to subscribe to live external data streams from the internet too, preferably with a simple spreadsheet formula that takes the URL of the location of the stream as argument. The server generates a stream of data and the spreadsheet application should receive a notification when new data is available.

For this kind of client-server communication, these are the four most used techniques:

1. Polling

With this technique the client sends an HTTP request to the server at regular time intervals (e.g. every second) to check if there is new data available. If there is no new data available, the server returns an empty response. This pull-based technique is in general very inefficient, because it makes many HTTP requests, also if the server does not generate any new data.

2. Long polling

Long polling is a variation of the polling technique. It is still the client that makes a request to the server. The difference is that the server does not respond immediately, but keeps the connection open until there is new information available. The server responds with the new information, and the client makes a new request to start the process again. Long polling is much more efficient than polling, because there are less HTTP requests. Furthermore the client receives the data real time, because there is no interval delay.

3. Server Sent Events

With Server Sent Events (SSE) the client opens a permanent HTTP connection to the server. Whenever there is new information available the server uses this connection to push the data to the client. The client does not have to ask the server if any updates are available. An SSE connection is one-directional. Only the server can send new data to the client, not the other way around. Server sent events are standardized in HTML5.

4. WebSockets

WebSockets are used for bi-directional communication. The client opens a persistent TCP connection with the server. Both the client and the server can start sending messages at any time.

From these four options, server sent events seem to be the best technique for the use case of streaming real time data to a spreadsheet client. In comparison to long polling, SSE is completely push based which corresponds to the reactive model. The server notifies the client when new data is available. WebSockets could be used as well, but in this case bi-directional communication is not needed, since the client only needs to receive updates from the server.

If the spreadsheet user subscribes to an external data stream, the client application must create an EventSource object that receives the server sent events. Events are sent over HTTP as strings, whereas a spreadsheet uses different data types. To know which data type is being sent, the server should not only send the data itself, but also the data type. This could be achieved by sending the data as JSON objects with the following two name/value pairs:

Name	Value
content	The data value
datatype	The type of the data. Can be text , numeric or bool

We would like to create a spreadsheet function that provides the functionality to subscribe to an external data stream. The URL of the web address that publishes the stream is passed as argument to this function. The function returns an observable, similar to the internal data stream functions, and should emit the items received over the SSE connection. The following pseudocode shows the process of creating an observable from an external SSE resource. It also ensures that the SSE connection will be closed at the moment that the observer unsubscribes from the observable, which happens when the user replaces the formula of the cell that is subscribed to the external data stream.

```
Observable[Value](observer => {
  //create the event source object that receives the sse messages
  val eventSource = new EventSource(url) {
    override def onEventReceived(sseEvent) {
      //convert the received json object into a spreadsheet value and
      //push the value to the observer if it is not yet unsubscribed
      if(!observer.isUnsubscribed()) {
        val item = Value.fromJSON(sseEvent)
        observer.onNext(item)
      }
    }
  }
}

//create a subscription with a lambda that closes the sse connection
val eventSourceSubscription = new Subscription(() => eventSource.close())

//add the sse subscription to the observer
//so that the connection will be closed
//when the observer unsubscribes from this observable
observer.add(eventSourceSubscription)
})
```

Publish internal data streams

In this stream processing model all cells in the spreadsheet application represent data streams. A useful feature would be to expose and publish these data streams, so that they can be used by other applications and shared with other users. This is most valuable if a stream is exported in a format, so that it can be imported directly as external data stream in another instance of the spreadsheet application. With the functionality to communicate among different instances of the application, the program can be used as a distributed spreadsheet.

To this end, the spreadsheet application must run a server in the background that is able to send data updates as server sent events. The server should listen to HTTP GET requests on a specific URL that is used to publish the data streams. A client that wants to subscribe to the value stream of a cell, submits an HTTP request to this URL, and adds an identifier for the worksheet and an identifier for the cell as query parameters to indicate which cell it is interested in.

When the server receives an HTTP request, it uses the values of the query parameters to retrieve the corresponding cell object. Then it subscribes to the value stream of that cell and opens an SSE connection with the client application. Every time the cell emits a new value, it converts the value into a JSON object, consisting of the data value and the data type, before sending it to the client. The client will receive these values as long as it keeps the SSE connection open.

4 EXCELSSTREAMS

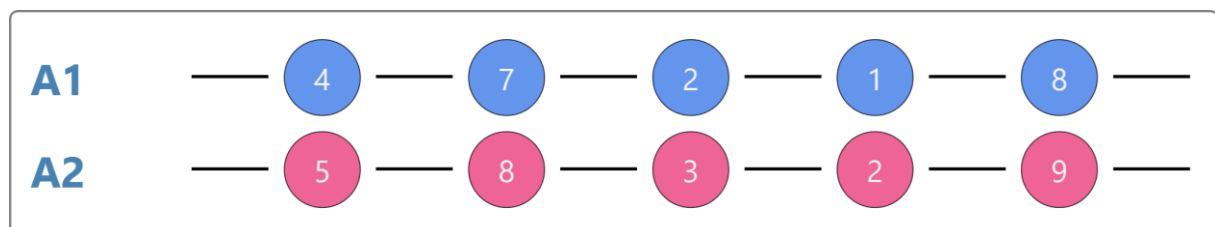
ExcelStreams is an add-in that provides the functionality to work with data streams in Microsoft Excel 2007 and later. It is written in C# and is integrated with Excel using Excel-DNA^[5], an open source project that integrates .NET into Excel and that creates Excel add-ins (.xll files) from compiled C# code and .NET libraries (.dll files). In contrast to Visual Studio Tools for Office (VSTO), Excel-DNA makes it possible to create User Defined Functions (UDFs) in C# that can be used directly in Excel. The data streams in the ExcelStreams add-in are implemented with Rx.NET, the Reactive Extensions library for .NET.

4.1 MODEL

Usually a cell in an Excel sheet contains a single value of a certain type (a number, a text string, a boolean) which means that the value in a cell is gone when it is replaced by another value. With the ExcelStreams add-in we want a cell to represent a stream of data instead of a single value. Therefore, for every cell that is in use we create a cell object that contains an Rx subject representing the value stream of that cell, as described in the spreadsheet model in chapter 3. When the user enters a formula in a cell, the value stream of that cell subscribes to the resulting observable. However, native functions in Excel cannot be changed and will result in a single value. That is why the Excel add-in distinguishes two kinds of functions: native excel functions that return single values and user defined functions that return observables.

4.1.1 NATIVE EXCEL FUNCTIONS

Assume the formula of cell A2 is equal to **=A1+1** and the numbers 4, 7, 2, 1 and 8 are entered into cell A1 sequentially. Then the value streams for these cells should contain the following data values.



Every time the value in a cell changes, it should be added to the value stream of that cell. The question is how these value changes can be observed. There is no value change event in Excel, and so the **AppEvents.SheetChange(Object sh, Range target)** event is the most convenient to use. This event will be fired when a cell is changed by the user or an external link, i.e. when the user enters a new formula or value into the cell. The first argument of the SheetChange event refers to the sheet where the change happened. The second argument is a reference to the range of cells that were changed. In the example above, the sheet change event is raised every time the user enters a new formula (or value) for cell A1. It is not raised for cell A2, because the user does not change its formula. The value of A2 changes only because the formula is recalculated and this will not fire the sheet change event.

The following steps describe the process of updating the value streams for cells A1 and A2. Note that the sheet is recalculated before the change event handler is called.

1. The user enters a new value in cell A1.
2. The sheet is recalculated and shows the updated value of A2.
3. The sheet change event is fired for A1.
 - a. Read the value of A1 and emit it in its value stream.
 - b. Update the value streams of the dependent cells of A1, in this case A2.

Updating dependents

This section gives the details of two alternatives to update the value streams of dependent cells. A first approach to emit the new values for all dependent cells is as follows:

For every cell, subscribe to its observable and do the following in the OnNext call:

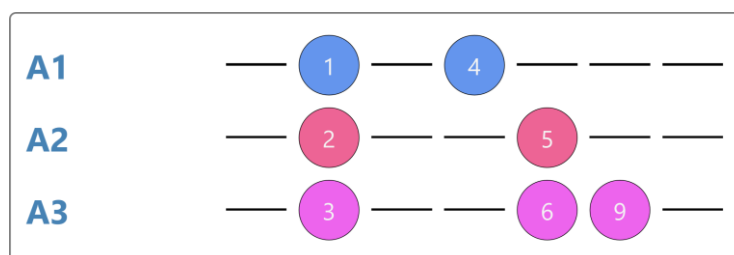
For each direct dependent D of this cell, read the value of D and emit this value for the corresponding value stream. When the Rx subject of D emits a new item, it will call the OnNext handler and starts updating its own dependents. In this way the changes will propagate to the indirect dependents as well.

When the user changes a formula or value that invokes the sheet change event handler for cell C, the value stream of cell C will emit the new value and starts the updating process by calling the OnNext method.

Most of the time this approach works fine, but in some cases it is not sufficient. Here is an example:

	A	B
1	1	
2	2	=A1+1
3	3	=A1+A2

If the value 4 is inserted into cell A1, one would expect the following data streams.



The change in A1 triggers an update for A2 and A3 at the same time, and then the change in A2 triggers an update for A3 again. So A2 would emit $4+1=5$ and A3 would emit $4+2=6$ and then A3 would emit $4+5=9$. However, Excel first recalculates the sheet until it is stable and only then fires the change event. Therefore, with this approach the second value for A3 reads a 9 instead of a 6 and the stream becomes 3 - 9 - 9, which is definitely not correct.

Another approach to update the streams of the dependent cells is as follows:

1. The user changes a formula or value that calls the sheet change event handler for cell C.
2. The new value is emitted in the value stream of cell C.
3. For each dependent D of cell C (both direct and indirect dependents) read the value of D and emit it in the corresponding value stream.

The set of dependents does not contain duplicates, so every cell emits only the final value after the sheet is completely recalculated. In the example above the result streams would be the same as in the figure, only the 6 is omitted. Hence, this is not a perfect solution either, but it is better than emitting the 9 twice. Since the sheet is completely recalculated before the sheet change event is fired, there is no way to retrieve the 6 from the sheet other than parsing and evaluating the formulas yourself, which is not feasible for all existing Excel formulas. The chapter about StreamSheets describes a spreadsheet program that is built from scratch with observables as basic data type. This program parses and evaluates the formulas itself, so this problem will not occur in that situation. The Excel add-in uses the second approach described in this section.

4.1.2 USER DEFINED FUNCTIONS

User defined functions (UDFs) are for the user the same as native Excel functions, but they work very differently in the background. They take observable streams as arguments and result in another observable stream, whereas native Excel functions take single value arguments and return a single value. The resulting observable will automatically update when (one of) the source observable emits a new item. The value stream of a cell subscribes to the result stream of a UDF. When the value stream of a cell emits a new value, it should update the value of the cell in the UI.

Updating the UI

Excel-DNA makes it possible to stream Rx observables as real time data to Excel using Excel's RTD functionality behind the scenes. Although at first sight it might seem a perfect fit for this spreadsheet model, it has a number of drawbacks which makes it unsuitable to use:

- Excel has an RTD throttle interval (default is usually 2 seconds). This means that Excel looks for changes in an RTD server every time the throttle interval has passed, so there is a delay of 2 seconds at maximum for a new value to appear in the cell. To reflect the changes in real time, the throttle interval can be changed to 0. However, a small interval value can cause Excel to keep recalculating and becomes unstable, in particular when a server updates too often.
- RTD callbacks are suspended when Excel is busy with other calculations. The main thread in Excel is used for calculations, but it is also the UI thread. Therefore:
 - Cells will not reflect updates when the user is editing a formula.
 - During a calculation a cell cannot be forced programmatically to update its value in the UI. This is often essential, because the value must be updated before reading the values of dependent cells. With the RTD functionality the value will only be updated after the sheet change event and all its calculations have finished.

A more straightforward approach to update the UI is as follows.

```
cell.GetStream().Subscribe(_ =>
{
    if (cell.rxFunction)
    {
        range.Dirty();
    }
});
```

For every cell, subscribe to its observable. If a new value comes in, set the range to dirty, i.e. force this range to recalculate. The user defined Rx function will be recalculated immediately. A UDF returns the latest value of the value stream of the cell it belongs to (which is illustrated in the next section), so the UI will be updated with the latest value without any delay.

Note that this process of updating the UI is only necessary for Rx functions. The result of an Rx function is an observable. The future items emitted by this observable should be pushed into the corresponding cell in the UI. On the other hand native functions are evaluated by Excel and result in single values. These values are read from the UI to emit them in the value streams of the corresponding cells.

Evaluating a formula

As an example, this section describes how the formula **=RX_COPY(A1)** is evaluated. It does exactly the same as the formula **=A1**, so it is not very useful in practice. However, it is the simplest Rx function, which makes it a good example to explain the process.

1. The user enters the formula **=RX_COPY(A1)** in cell A2.
2. The UDF in C# with the name RX_COPY is called and the argument A1 is passed as a reference. This reference is converted into a range and then gets the corresponding cell object to retrieve the value stream of cell A1.
3. The result stream of the UDF is the same stream as the value stream of cell A1, because it is just a copy. The more complex UDFs transform or combine the argument streams by applying some operations at this point.
4. The value stream of the caller cell (cell A2 that called the UDF) subscribes to the result stream, and re-emits the values that it receives. The subscription is stored in the cell object, which makes it possible to unsubscribe from the result stream when the formula of the cell is replaced.
5. The UDF cannot return a data stream, so it returns the latest value from the value stream of A2.

Although this is a good start, there are several problems with this evaluation process.

- During one sheet recalculation, a formula can be evaluated multiple times. This is not a problem for Excel without streams, because the final calculation is the only one that counts. However, if you work with streams, the historic values are important too. If a formula is evaluated twice, then the value will be duplicated in the stream.

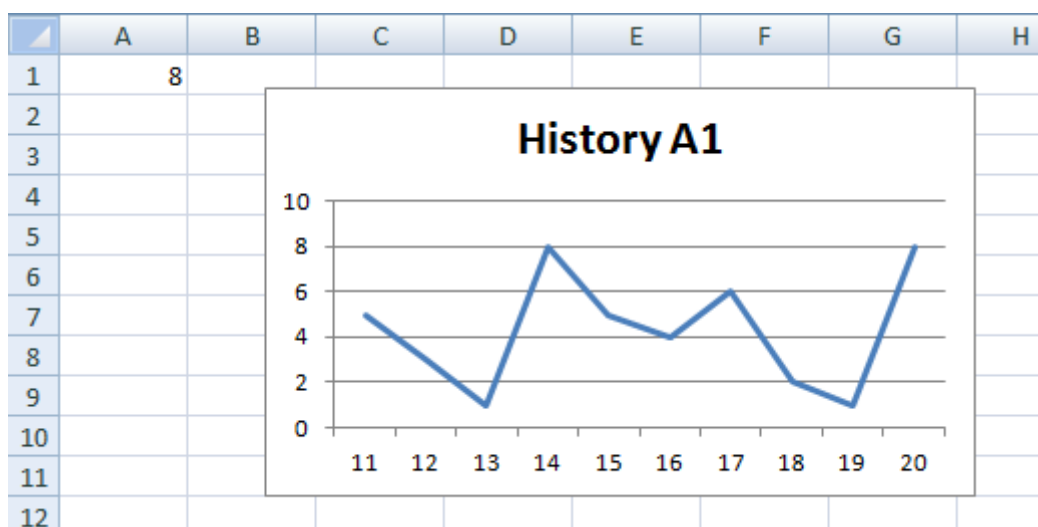
- Excel automatically recalculates dependent cells when the content of a cell changes, but this is not necessary for Rx UDFs. They are not based on a single value of another cell, but on its stream, and future values are already included in this stream. Hence, a UDF does not need to be recalculated if the value of a cell that it depends on changes.

A direction towards a solution could be setting Excel's calculation mode to manual and explicitly specifying in the code when a certain range needs to be recalculated from within the sheet change event. However, this makes the recalculating process overly complicated. All native Excel functions still must be recalculated before reading the new values from the UI and the Rx UDFs must be recalculated anyway to update the UI. Moreover this automatic recalculation of dependent cells and the propagation of changes is the key characteristic that makes a spreadsheet a reactive program, so all together, the automatic calculation mode is the preferred choice.

Instead, during the evaluation of a UDF the value stream of the caller cell will not immediately subscribe to the result stream, but the caller cell stores the result stream in a field named `nextStream`. When the formula is re-evaluated, the `nextStream` is replaced, so that it always contains the most recent calculated stream. The value stream of a cell only subscribes to the `nextStream` when the cell formula changes, i.e. when a sheet change event is fired for a cell with an Rx function. The other re-evaluations of a UDF do nothing more than updating the UI by returning the latest value of the cell's value stream.

Visualizing the history of a cell

Every cell in a spreadsheet that is in use stores an Rx subject with the value stream of that cell. The history of a cell can be visualized in a chart by right-clicking on the cell and selecting the option Show History from the context menu. It creates an Excel chart that subscribes to the value stream of the cell and updates the series when new data is available. The chart starts to record the data at the moment that the chart is created, because at that moment it subscribes to the value stream and it starts receiving updates. The maximum number of data points is set to 10 and values that are not numbers are interpreted as zero.



4.2 LIST OF RX FUNCTIONS

This is an alphabetical list of all Rx user defined functions available for Excel.

RX_BUFFER(source, count, skip)	
Description:	Periodically collects items from the source observable and emits these collections as lists. Read more about lists in the next section.
Arg1: source	A cell reference, refers to the value stream of that cell
Arg2: count	The number of items in one list
Arg3: skip	The number of items to skip before starting to collect a new list (optional)
Output:	A stream of lists with <i>count</i> elements per list

RX_COPY(source)	
Description:	Copies the source stream. Mainly used for debugging purposes.
Arg1: source	A cell reference, refers to the value stream of that cell
Output:	The same stream as the source stream

RX_MERGE(source1, source2)	
Description:	Merges two observables into one.
Arg1: source1	A cell reference, refers to the value stream of that cell
Arg2: source2	A cell reference, refers to the value stream of that cell
Output:	A stream with the merged values from both cells in the arguments

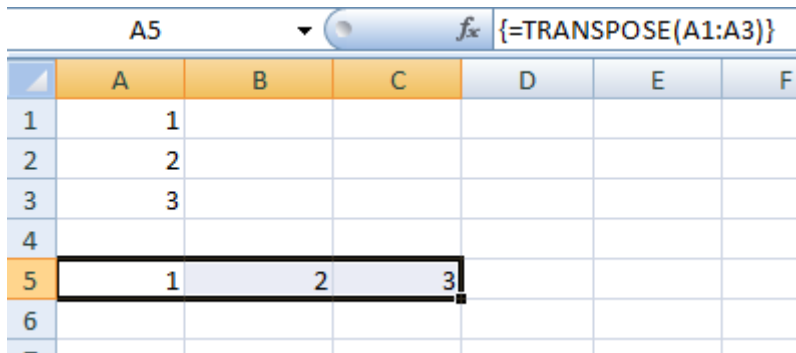
RX_PRE(source, n , start, tick)	
Description:	Retrieves one of the previous emitted items of the source stream.
Arg1: source	A cell reference, refers to the value stream of that cell
Arg2: n	The number of steps to navigate back in the source stream. The latest value is n=0, the previous value is n=1, the second previous n=2, and so on
Arg3: start	The value that should be emitted initially (optional)
Arg4: tick	Emit a new item only when the tick stream gets a new value (optional) See the use case example with a state machine for more information
Output:	A stream of previous values from the source stream

RX_STREAM(url)	
Description:	Subscribes to an external data stream. Read more about this function in the section about data streams and server sent events.
Arg1: url	The url to the external data stream
Output:	A stream that represents the sequence of items in the external data stream

4.3 LISTS IN EXCEL

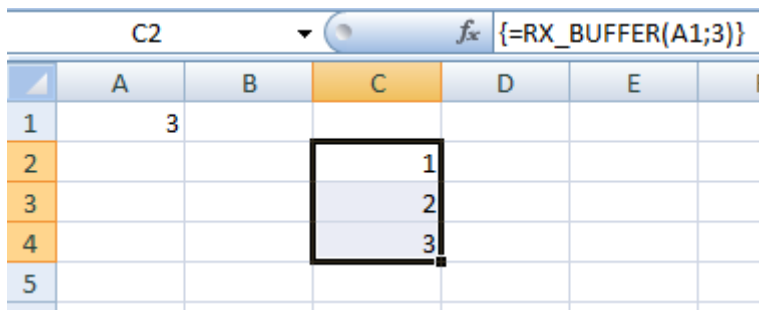
The function `RX_BUFFER` can be used to access the history of a cell. It periodically collects items from a source observable and emits these collections as lists. There is no list data type for cells in Excel, so the question is how these lists could be best represented in Excel.

There are some native Excel functions that return multiple values in adjacent cells. These functions are called array functions and are inserted by first selecting a range of cells and then inserting the formula with `CTRL + SHIFT + ENTER`. An example of an array function is `TRANSPOSE`, that transforms a horizontal cell range in a vertical range and vice versa.



	A	B	C	D	E	F
1	1					
2	2					
3	3					
4						
5		1	2	3		
6						
7						

With Excel-DNA it is also possible to create user defined array functions by returning a C# (multidimensional) array of objects. The function `RX_BUFFER` results in an observable of lists, and these lists could be transformed into an array that is returned by the UDF.



	A	B	C	D	E	F
1	3					
2			1			
3			2			
4			3			
5						

If A1 emits the items 1, 2 and 3, the formula in C2 returns the array {1, 2, 3}. This seems to work fine, but there is a problem if you want to use another Rx UDF somewhere in the sheet with the stream of C2 as argument. In this example you would expect C2 to be a stream of numbers, but in reality it is the stream of lists from the `RX_BUFFER` function. Furthermore, the value streams of C3 and C4 do not emit the 2 and 3, because the values come from the list in C2. Hence, this approach will not work properly if you continue calculating with the values of the range C2 to C4. Since C2 contains a list, it would be more natural to change the data type of C2 to list, so that this single cell contains the value [1, 2, 3]. This data type does not exist in Excel, but this value can be printed as a text string.

Internally the list is represented as an Excel range. The advantage of this range representation is that all native Excel functions that work on ranges can now also be applied to lists. It is not possible to create range objects programmatically without a reference to a range in a sheet. Therefore, every workbook has a hidden worksheet named MyLists that stores the lists that are used in the workbook as ranges. One row in this sheet represents one list. When the RX_BUFFER function returns a new list, it looks for the first empty row in the MyLists sheet and adds the list to that row, i.e. the first value of the list in column A, the second value in column B and so further.

If a value of a cell is a list, its value stream contains a reference to the corresponding range in the MyLists sheet and it is printed in a cell as a string in the format [a, b, ...]. Assume cell D1 contains the value [1, 2, 3] and you want to sum this list. You cannot directly use the formula =SUM(D1), because it will work on the string [1, 2, 3] and not on the range behind it. A (non-Rx) function L2R is added to the ExcelStreams add-in, that transforms a list string into its corresponding range by simply getting the latest value of the value stream of the referred cell. The formula to sum up the list will then be =SUM(L2R(D1)). Since the function L2R returns a range, it can also be inserted in multiple adjacent cells as an array as shown in the following example.

	A	B	C	D	E	F
1	3		=RX_BUFFER(A1;3)	[1, 2, 3]		
2						
3			=SUM(L2R(D1))	6		
4						
5			{=L2R(D1)}	1	2	3
6						
7			=L2A(D1)	MyLists!\$A\$2:\$C\$2		
8			=SUM(INDIRECT(D7))	6		

The reason that the range can now be inserted as an array, is that L2R is not an Rx function. It is processed as a native Excel function instead, which means that if the cell content changes, the value of the cell is read and emitted by the value stream observable. If it were an Rx function, the values of the observable would be pushed into the cell.

In addition to the L2R function, there is also a function L2A, that transforms the list into its address location. The output is similar to the output of the ADDRESS function, and can be used as argument in the INDIRECT function to get the range.

4.4 DATA STREAMS AND SERVER SENT EVENTS

A user cannot only use data streams from other cells in the spreadsheet, but it is also possible to subscribe to live external data streams from the internet. The UDF for this purpose is `RX_STREAM(url)`, in which `url` is the web address that publishes the stream. The server generates a stream of data and the Excel client receives a notification when new data is available. As explained in chapter 3, the server sends the data as server sent events to the client. The spreadsheet application must implement an `EventSource` to receive the server sent events. In addition, it should run an SSE server in the background to publish data streams and make them available for other users and applications.

4.4.1 SSE CLIENT

`EventSource4Net`^[6] is used to implement the Server Sent Event client in .NET. With this library one can create an `EventSource` object from a URL string. This `EventSource` object has an event handler `EventReceived` that is called when a new SSE message arrives.

The result stream from the user defined function `RX_STREAM` contains the Server Sent Events received on the specified URL. The UDF creates an observable and uses the `EventReceived` listener within the observable to emit new values as illustrated in section 3.2. The `EventReceived` listener from `EventSource4Net` receives the data as JSON objects. These objects are transformed into C# objects and emitted by the observable. The SSE connection will be closed when the observer unsubscribes from this observable.

4.4.2 SSE SERVER

Every cell in a spreadsheet is exposed as a data stream itself. By right-clicking on a cell, one can select the option `Copy URL` from the context menu. This copies the URL for that cell to the clipboard. The format of the URL is:

`http://localhost:8080/api/sse/83408896/A1`

The number 83408896 is a reference to the `sheetId` and `A1` is a reference to the cell address. This URL can be used as argument in the `RX_STREAM` function in another Excel instance with the `ExcelStreams` add-in.

When the Excel add-in is loaded, it asks for a port number and starts an HTTP localhost server. Note that Excel must be run as administrator to have the appropriate rights to start the server. The server is configured with the route template `api/sse/{sheetId}/{address}`, that maps HTTP requests that match the template to a method call in the `SSEController` class. This method is named `Get`, so that it will listen to HTTP GET requests. The `sheetId` and `address` in the route template are placeholder variables that are passed as arguments to the `Get` method.

`ServerSentEvent4Net`^[7] is used in the `Get` method to open an SSE connection to the client. The `sheetId` and `address` arguments are used to find the value stream of the requested cell and it subscribes to this observable. Every time the value stream emits a new item, the value is converted into its JSON representation. The JSON object is sent to the client over the SSE connection as a server sent event with `ServerSentEvent4Net`.

The following example shows how the RX_STREAM function is used to communicate between two different Excel instances.

	A	B	C	D	E	F	G
1	Running on http://localhost:8080/						
2							
3	1	=1					
4	2						
5							
6		=RX_STREAM("http://localhost:8081/api/sse/89110528/\$A\$4")					
7							
8							



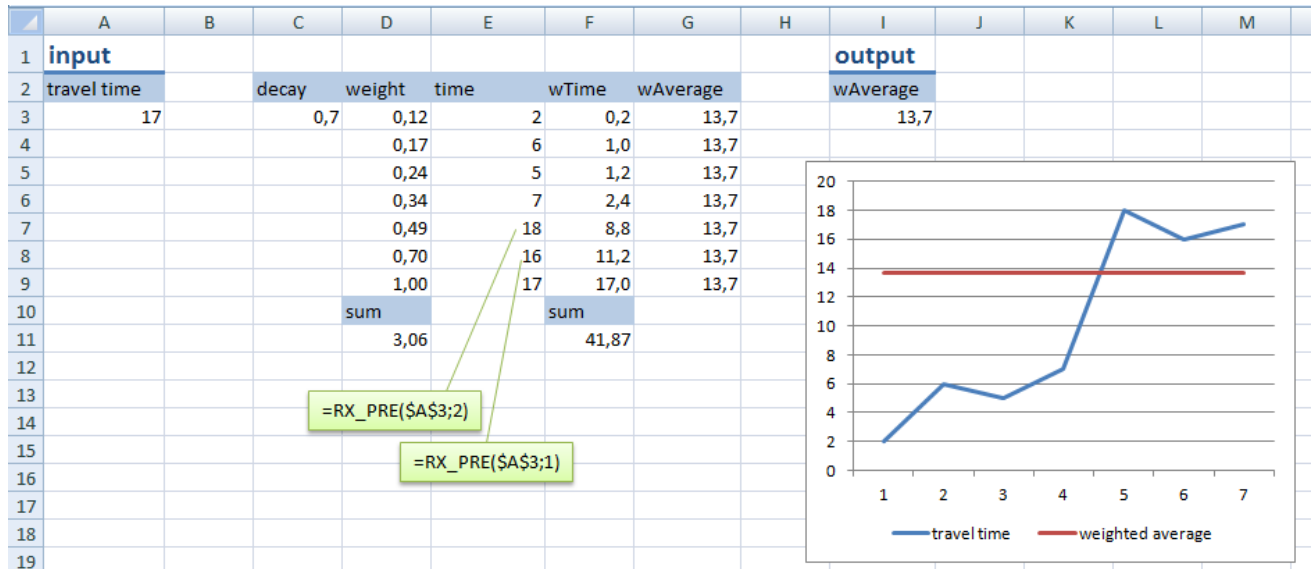
	A	B	C	D	E	F	G
1	Running on http://localhost:8081/						
2							
3	1	=RX_STREAM("http://localhost:8080/api/sse/74889216/\$A\$3")					
4	2						
5							
6		=A3+1					
7							

The first Excel instance runs on port 8080 and the second one on port 8081. The URL from cell A3 in instance 1 is copied from the context menu, and imported in cell A3 of instance 2 with the RX_STREAM function. Next, instance 2 applies a standard operation of incrementing the number in cell A4. The URL of this cell is copied and imported back again in cell A4 of instance 1. In this way Excel can be used as a distributed spreadsheet program, where different Excel instances can communicate and collaborate together.

4.5 EXAMPLES

This section provides three examples of stream processing in Excel with ExcelStreams. All examples are modifications of the case studies in the paper "Stream Processing with a Spreadsheet" by M. Vaziri et al.^[1]

4.5.1 RECENCY-WEIGHTED AVERAGE

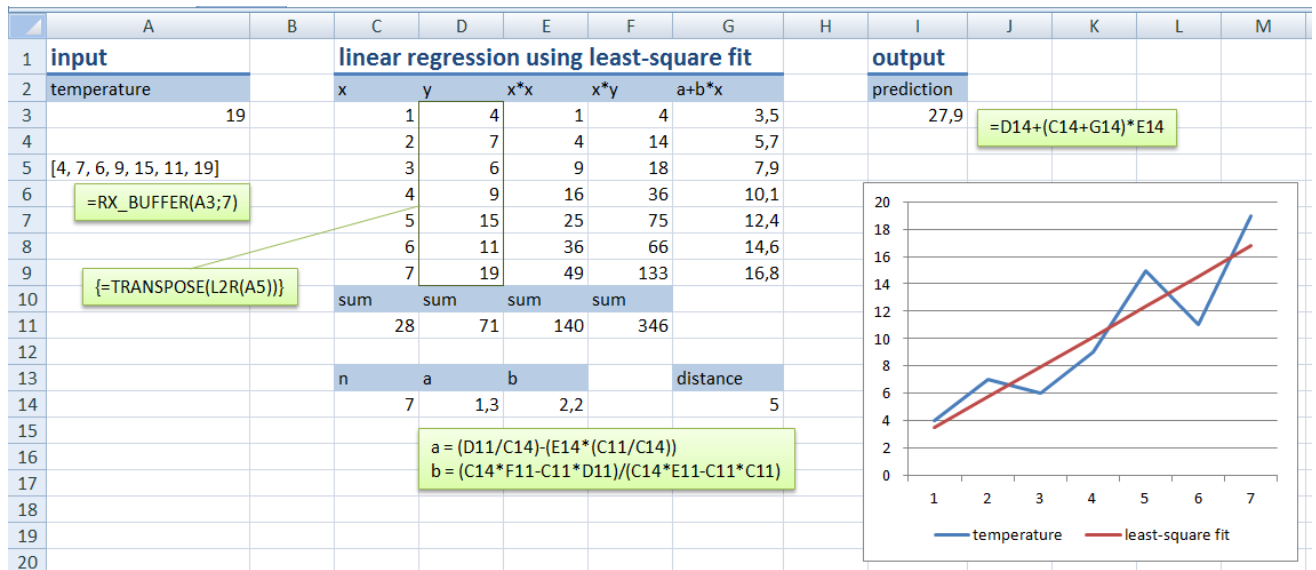


This example estimates the travel time between two geographical locations by analyzing a stream of registered travel times between the same two locations. To give an appropriate estimation, recent travel times should be considered more important due to changes in the current traffic situation like traffic jams or road blocks. Cell A3 represents the input stream of registered travel times. This stream could be imported with the RX_STREAM function from the internet where drivers submit their travel times.

The decay is set to 0.7 which means that the most recent value has weight 1, the next one has weight $1 \cdot 0.7$ and so on. The travel times in column E are the last seven values from the source stream in A3 and are calculated with the RX_PRE function. The first argument of this function is a reference to the source stream in A3. The second argument decides how many elements it should go back in the stream.

The weighted average is calculated by dividing the sum of weighted times by the total weight. This value is the estimation of the travel time at that moment and is calculated in cell I3. It is copied in column G to show the weighted average together with the actual travel times in the chart. It will update automatically when new travel times are registered. Cell I3 could be exported as output stream to the drivers to inform them about the current estimated travel time.

4.5.2 FORECASTING

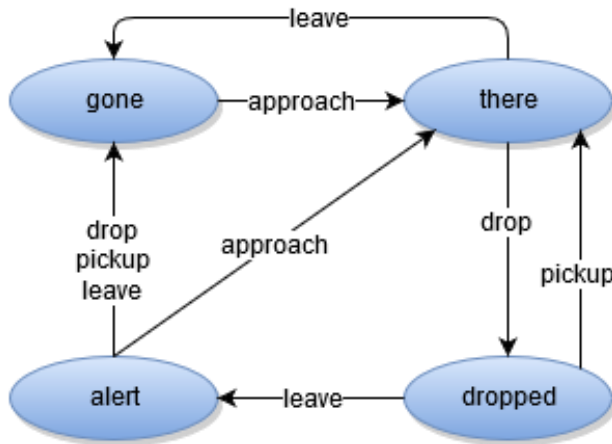


This example analyzes and visualizes the trends in a stream of temperature values and predicts the temperature at a point in the future. Cell A3 contains the input stream. The temperature readings can be inserted manually or it could be a reference to an external source with the RX_STREAM function. Cell A5 shows a list of the seven most recent values calculated with the RX_BUFFER function. This list is converted into a range with the L2R function and then transposed to fit in column D. This approach to set the seven most recent values in a range is more efficient than the one used in the weighted average example, because it only uses one buffer. The weighted average example uses the RX_PRE function six times and this function uses a buffer in the background. It takes the first element of the list and drops the rest of the list. In this example all elements in the range come from the same list in A5.

All other formulas in columns C to G are simple arithmetic operations to calculate the data points for the least-square fit line with formula $y = a + b * x$. This least-square fit line in column G is plotted in a chart together with the actual temperature values in column D. This chart shows the trend of the temperature stream. Lastly, the output stream in cell I3 is a stream of predicted temperature values in the future. The distance is set to five steps into the future. The predicted value is calculated by extrapolating the least-square fit line and reading the value at $x = n + distance$.

4.5.3 STATE MACHINE

With ExcelStreams it is also possible to implement state machines in Excel. This example implements a state machine regarding the identification of suspicious events. The idea is that there might be a security risk if someone approaches a location, drops an object and then leaves without taking the object back.



This state machine can easily be represented in a spreadsheet with a transition table. The left header of the table contains all possible states and the top header contains all possible activities. The intersection of a state and an activity is the result of applying the activity to that state. If the activity does not exist for the state, the result is the same state as it already was in.

	A	B	C	D	E	F	G	H	I	J	K	L
1	input		transition table						old state		output	
2	activity		state	approach	drop	pickup	leave		state		state	
3	leave		gone	there	gone	gone	gone		dropped		alert	
4			there	there	dropped	there	gone					
5			dropped	dropped	dropped	there	alert					
6			alert	there	gone	gone	gone					
7												
8												
9												
10												
11												
12												

=RX_PRE(K3;0;"gone";A3)

=IFERROR(
VLOOKUP(I3;
C3:G6;
MATCH(A3;D2:G2;0)+1;
FALSE);
I3)

Cell K3 only uses native Excel functions and is a two dimensional lookup in the transitional table of the old state (I3) and the value from the activity stream (A3). The value of I3 stores the old state. There is a circular reference between I3 and K3. The output state depends on the old state and the old state changes if the output state changes. That is why I3 uses the RX_PRE formula with four arguments. The first argument is the source stream, which is K3. The third argument is the start state. It starts in the state gone because there is initially no previous state. The fourth argument is the "tick stream", that is used to prevent the streams from updating continuously due to the circular reference. With A3 as the tick stream I3 emits a new item only at the moment that A3 gets a new value. It is implemented with the Rx operator WithLatestFrom that does not come with Rx.NET by default yet, but it will be added in the next release.

The second argument of the RX_PRE function is 0, which means that it should take the 0th previous item from the source observable, i.e. the current value of K3. The reason that this argument is 0 and not 1, is that the function in I3 is calculated before the function in K3. So I3 first takes the value from K3 and then updates the value in K3 itself.

Unfortunately the evaluation order of Excel formulas is not fixed. Excel tries to determine an optimized calculation sequence of cells. A second calculation will be faster, because it stores the calculation sequence after the first time and uses it in subsequent calculations. Nevertheless, the calculation sequence is dynamically determined and may change from calculation to calculation. My experience is that user defined functions are calculated before native excel functions, but it is not documented, so there is no guarantee that this assumption always holds. Therefore, one should carefully check this assumption before using it.

4.6 LIMITATIONS

Although the Excel add-in provides new functionality for stream processing in Excel, there are some limitations that should be considered.

Calculation order

The algorithms that Excel uses to update cells, does not only lead to user defined functions that may be calculated multiple times, but also to calculation sequence orders that may change from calculation to calculation. Hence, there is no guarantee that one cell will be updated before another one or vice versa. Usually this is no problem for spreadsheets in Excel, because the cells in a sheet are recalculated until nothing changes and the result is the only thing that counts. The number of evaluations and the calculation order do not affect the end state. However, when calculating with data streams it does affect the result. Multiple calculations of one cell can lead to duplicate emitted items in the observable. A solution to this problem was described in the section about user defined Rx functions.

The problem with different calculation orders is more complicated. The state machine example demonstrates how this could change the result of a formula. One assumption is that user defined functions are calculated before native excel functions, but this is not documented and it doesn't say anything about the order of two UDFs or two native functions. The following UDF does not work properly because of the calculation order.

RX_FILTER(source, filter)	
Description:	Emits the items from the source observable that satisfy the predicate
Arg1: source	A cell reference, refers to the value stream of that cell
Arg2: filter	A stream of booleans, that decide if a source item should be emitted
Output:	The same stream as the source observable without the items that don't satisfy the filter argument

This is an example of the RX_FILTER function.

	A	B
1	1	
2	TRUE	=A1 < 10
3	1	=RX_FILTER(A1;A2)
4		

If the source stream in A1 emits a new item, first the UDF function in A3 is calculated with the old value still in A2. Then A2 is updated because of the change in A1, which will propagate its change to A3 again. So if A1 emitted a 5, the observable of A3 would emit it twice. And if A1 emitted a 12, A3 would emit it once. The function would work properly if A2 would have been updated before A3. The RX_FILTER function cannot force its second argument to be evaluated first, because every time the RX_FILTER function is calculated it would calculate A2 which lead again to a calculation of RX_FILTER and so turns into an infinite calculation loop.

Combining UDFs and native Excel functions in one cell

Another limitation of the Excel add-in is that UDFs and native Excel functions cannot be combined in one formula. The value stream of a cell subscribes to the result observable from a user defined Rx function. This function then returns the latest value from the cell's value stream. This means that a UDF is coupled to a cell object and cannot be part of a larger formula belonging to one single cell. Here is an example:

	A	B
1	6	
2	7	=RX_COPY(A1)+1
3		

The visible result is exactly what you would expect, but the value emitted in the background for cell A2 is 6. The value stream of A2 subscribes to the result of RX_COPY(A1) and then returns the latest value of A2, which is the 6. Then Excel adds 1 and the result is 7. However, since this is an Rx function and not a native Excel function, the results are pushed into the cell and not read from the cell. The presented value doesn't correspond to the value emitted by the observable, and hence UDFs and native Excel functions cannot be combined in one formula. A solution is to split the formula in two cells.

	A	B
1	6	
2	6	=RX_COPY(A1)
3	7	=A2+1

In this example A2 has an Rx formula that corresponds to the value stream of the cell. A3 has a native Excel function, so the result is read from the cell and emitted for the value stream of A3.

Inserting and deleting rows and columns

There are no specific events in Excel for the insertion or deletion of entire rows and columns. Therefore, the worksheet change event must be used to detect these changes. That is the same event that is used for cell changes. A simple way to detect column insertion or deletion would be to check if the number of changed cells in the worksheet change event is equal to the total number of rows in Excel, although this is not very reliable. The maximum number of rows is not the same in different versions of Excel. Moreover, a column insertion has the same number of cell changes as a column deletion, so a distinction between the two cannot be made.

Another problem with column (or row) insertion and deletion in combination with the worksheet change event is that only the cells of the inserted or deleted column are in the changed range, but every column to the right of the changed column should shift left or right accordingly. For example, assume cell B1 has a cell object with a value stream and column A is removed. The data of B1 is shifted to A1, but there is no change event for B1, so the value stream is still tight to B1.

Finally, since there is no distinction between the insertion of an entire column and a regular sheet change event, it must be checked for every cell in the cell change event whether a corresponding cell object exists to emit an empty value. In Excel 2007 there are more than 1 million cells in a column, which results in bad performance.

To support column and row insertion and deletion, another workaround solution is needed to detect these kind of changes. Furthermore, these changes should be processed efficiently to not run into performance issues by iterating over all cells in the row or column. And the data integrity should be maintained by shifting subsequent rows or columns and their background cell objects accordingly. A solution may be found for future versions of the Excel add-in. The only way to disable row and column insertion and deletion is by protecting each worksheet. However, this disables other functionality like adding charts and notes too, so that is why it is still enabled.

Performance issues

The change of a large range of cells at the same time can lead to performance issues. Every cell that is in use has an object running in the background to store the value stream. For example, if a value 1 is inserted in all cells of column A at the same time, an object is created for every cell in column A, and each observable emits the value 1. Since one column in Excel has more than a million cells, this process is time-consuming and Excel becomes unresponsive.

Getting the dependents of a cell

When the user changes the content of a cell, a list of dependent cells must be retrieved to update the value streams of cells with native Excel functions. The method `Range.Dependents` is used for this purpose, but it comes with two limitations. First of all, this method returns only the dependents on the current sheet. One will not be notified of dependents on other sheets, which leads to inconsistency if there are cross references between different worksheets. Secondly, if a cell has no dependents it will throw an exception. There is no method `HasDependents` or something similar, so the only option is to catch the exception. Throwing many exceptions causes the program to become slow and unresponsive.

The best solution would be to parse the formulas and keep track of the dependencies yourself. Then it would be trivial to add an extra method that checks if there are any dependents and dependencies to other sheets could be included as well. `XLParser` (<https://github.com/spreadsheetlab/XLParser>) is a library for C# that parses Excel formulas with a 99.9% compatibility rate. This library can be used to get all the references in a formula. The problem with this solution is that in some cases too many new cell objects must be created at the same time to store the dependencies. For example, if B1 has formula `=SUM(A:A)`, then for every cell in column A, a cell object must be created to store a dependency to B1. This leads to a performance issue as described in the previous section. In the context of this thesis project, the `Range.Dependents` method is chosen over manual dependency management.

5 STREAMSHEETS

StreamSheets is a new spreadsheet application written in Scala with native support for stream processing. It is based on the spreadsheet model described in chapter 3. The data streams in this application are implemented with RxScala, the Reactive Extensions library for Scala.

5.1 ADVANTAGES AND DISADVANTAGES

StreamSheets has a number of advantages and disadvantages in comparison to the ExcelStreams add-in described in the previous chapter. The most significant advantage of StreamSheets is that it is primarily designed and built with the support for data streams in mind. Excel is built on the concept of a cell containing a single value, and that is why many workaround solutions were needed in the development of the Excel add-in, for instance because the formulas could be evaluated multiple times and the calculation order was not fixed. Additionally, a user defined function for Excel could not simply return a data stream, but it had to return the latest value of that data stream and then re-evaluate when the observable emitted a new value. With the support for data streams as a basic principle, no inelegant workaround solutions are needed in the StreamSheets application. This results in a spreadsheet model that is less complicated and much cleaner than the model for the Excel add-in. Furthermore, there are no Excel-specific limitations, such as certain events that do not exist or exceptions that are thrown while getting the dependents of a cell.

An obvious disadvantage of StreamSheets in comparison to ExcelStreams is that it lacks all the functionality that is already available in Excel, such as options for cell formatting, pivot tables, charts, and the long list of native Excel functions. A con in the development of StreamSheets is that all values and formulas must be parsed and evaluated. In Excel the parsing is done for you and one can simply create C# functions that are directly available in Excel as user defined functions. Values and native functions are evaluated by Excel, so that only the user defined functions need to be implemented.

With Scala's parser combinators, it is quite easy to implement a basic formula parser for a spreadsheet application. It becomes more complicated if it would be extended, for instance with cross sheet references and named ranges. An evaluator for the Scala application must be written to calculate the value of a formula. This offers more flexibility, because the evaluator can be defined to return an observable of values instead of a single value. In that case, it is not needed anymore to re-evaluate a formula of a cell, when the value stream of that cell emits a new item. A nice side-effect of using the Reactive Extensions framework is that manual dependency management is not needed, because it is implicitly done by RxScala. Rx operators are used to transform an observable into another one. When the source observable emits a new item, the change will automatically propagate to the dependent observables. Hence, if a source cell emits a new item, it is not needed to explicitly find and update its dependent cells.

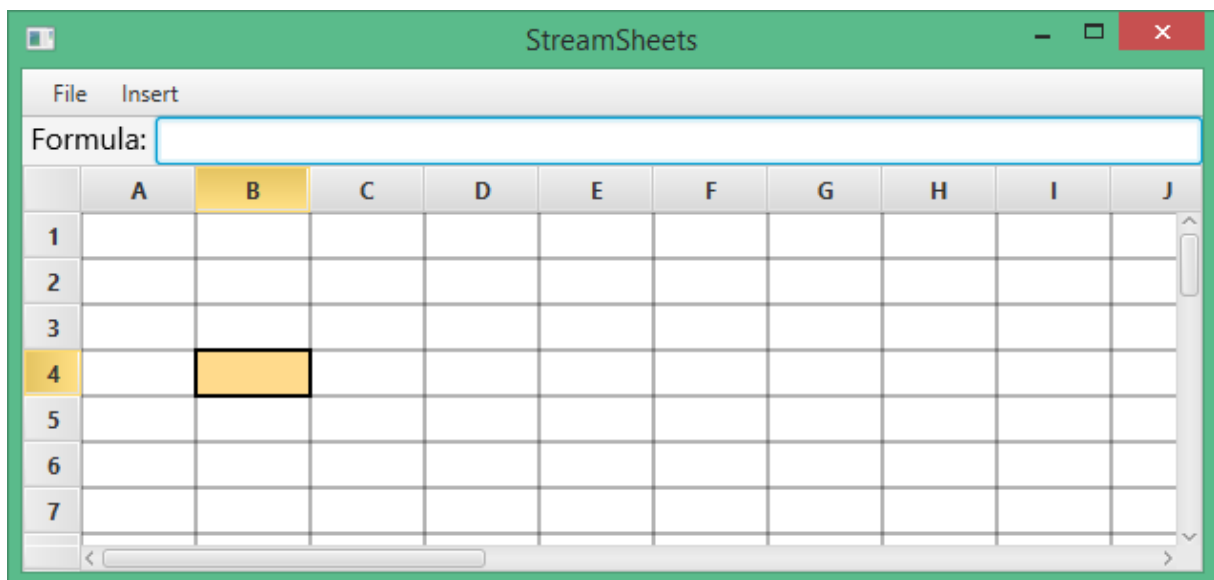
Another pro of StreamSheets is that the distinction between native functions and user defined functions does not exist. We have seen in the previous chapter that these two must be treated completely different in Excel, whereas all formulas are equal in the Scala application. Every function results in an observable and single values are wrapped in observables too. It is no problem to combine different functions in one formula and in one cell, because all functions and values are data streams that are processed equally.

5.2 MODEL

The three main components in the StreamSheets model are the user interface, the spreadsheet object and the cell object.

User interface

The user interface of the spreadsheet application is built with JavaFX and ControlsFX. ControlsFX is a library that provides UI controls built on top of JavaFX. One of these UI components is the SpreadsheetView, that is used as the spreadsheet grid. It is similar to the TableView of JavaFX, but offers some functionality specific for spreadsheet applications, such as resizable columns and rows that are indicated with letters and numbers respectively and the ability to fix a column or row to the screen so that it remains visible when the user scrolls right or down in the spreadsheet table. On top of the SpreadsheetView is a text field where the user can enter a formula for the currently highlighted cell in the SpreadsheetView.



Spreadsheet

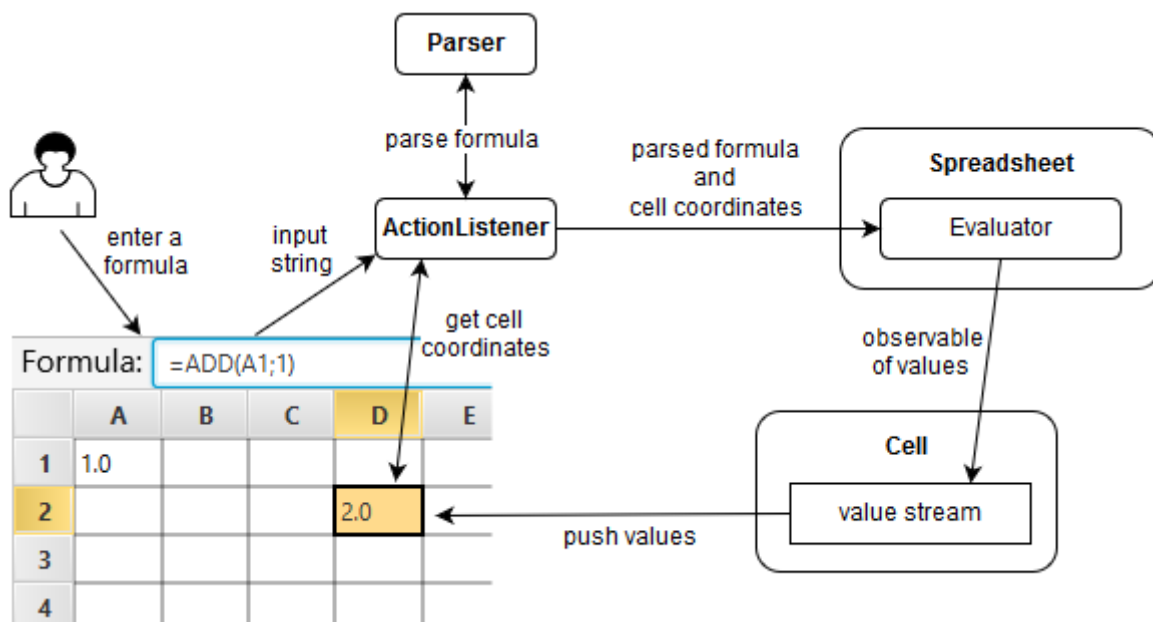
The spreadsheet table is represented by a spreadsheet object that is running in the background. This spreadsheet object stores a two-dimensional array of cells that correspond with the cells in the UI and defines a method that sets a new formula for a given cell. An action listener is attached to the formula input box in the UI. When the user enters a formula in the text field, the event handler asks for the coordinates of the selected cell in the SpreadsheetView and sets the new formula for the related cell object. The formula that is inserted in the input box is a textual string. The spreadsheet object converts this string into an observable of values with a parser and an evaluator. The parser and evaluator that are used in StreamSheets are extended and modified versions of the ones described in chapter 33 of Programming in Scala by M. Odersky et al. [8] and are explained in detail in section 5.3 and 5.4.

Cells

Every cell object has a value stream that is implemented as an Rx subject. The value stream of a cell acts as an observer when it subscribes to the result stream of a formula. It acts as an observable when another cell refers to the value stream of this cell. When a value stream of a cell subscribes to the result stream of a formula, the subscription is stored in a local field, so that the value stream can unsubscribe from the observable at the moment that the formula of the cell is being replaced.

Updating the UI

The value stream of a cell is exposed as an observable. When the UI is initialized, it iterates through the cells in the spreadsheet view and retrieves the corresponding cell objects. The cells in the UI subscribe to the value streams of the underlying cell objects, and update their cell values in the onNext call. Every time a value stream emits a new item, the value is pushed into the cell of the spreadsheet grid. As opposed to ExcelStreams, there is no distinction between Rx and non-Rx functions. All cells are reactive and push based, so the cell values are never read from the UI to emit them in the value stream of a cell object.



The spreadsheet model is summarized in the figure above. The user enters a formula in the text field in the user interface. When the user hits the enter button, the ActionListener receives the formula as a text string, and retrieves the coordinates of the cell that is currently selected (cell D2). The parser parses the formula and sends the result to the evaluator of the spreadsheet object together with the cell coordinates. Then the spreadsheet object evaluates the parsed formula. This results in an observable of values, that is sent to the cell object with the coordinates received from the ActionListener. The value stream of that cell subscribes to this observable. Changes in the value stream of a cell are pushed into the matching cell in the UI.

5.3 PARSING FORMULAS

The spreadsheet program has an internal language of formulas that a user writes to fill the spreadsheet with data. The formulas are entered in the formula box as simple text strings. A parser is needed to convert the strings into a data structure of formulas that are recognized by the program. A formula could be a simple value, such as a number, a textual string or a boolean, a coordinate that refers to another cell, or an application of a function to a list of arguments. This is a list of all formulas that are part of the spreadsheet language in StreamSheets.

```
trait Formula

case class Coord(row: Int, column: Int) extends Formula
case class Range(c1: Coord, c2: Coord) extends Formula
case class Number(value: Double) extends Formula
case class Textual(value: String) extends Formula
case class Bool(value: Boolean) extends Formula
case class Err(value: String) extends Formula
case class Application(function: String, arguments: List[Formula]) extends Formula
case class EmptyFormula() extends Formula
```

Although a range cannot be a formula on its own, it is included in the list, because a range can be an argument of a function application and the arguments are represented as a list of formulas. The parser will enforce that ranges can only be used as function arguments and not as top-level formulas.

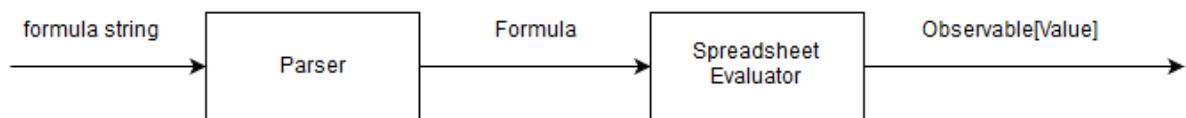
In Scala the easiest way to parse strings into objects is by using Scala's parser combinators. The idea of parser combinators is that you can combine many simple parsers into one bigger and more complex parser. A small parser consumes only a part of the input string and returns the output and the rest of the input string that then will be parsed by another parser. If one of the parsers fail, it will return a failure. The table below shows how the parser for the spreadsheet language is implemented. The bold and italic words in the second column are references to other parsers in the same table. The others are written with regular expressions.

Parser name	Matches	Example	Output object
cell	One letter followed by one or more digits	D12	Coord
range	A <i>cell</i> followed by a colon followed by a <i>cell</i>	A1:A5	Range
number	Any integer or decimal number	4.5	Number
str	Any textual string between double quotes	"hello"	Textual
textual	Any textual string of at least one character that doesn't start with an equals sign	hello	Textual
bool	Either TRUE or FALSE	TRUE	Bool
application	An identifier followed by a list of <i>args</i> between brackets that are separated by semicolons	SUM(A1:A5)	Application
expr	A <i>cell</i> , <i>number</i> , <i>str</i> , <i>bool</i> or <i>application</i>	A4	Formula
arg	A <i>range</i> or <i>expr</i>	A1:A5	Formula
formula	A <i>number</i> , <i>bool</i> , <i>textual</i> , empty string, or an equals sign followed by an <i>expr</i>	=SUM(A1:A5)	Formula

A spreadsheet formula is valid if the entire input string can be parsed by the formula parser. If that is the case, the output object of the formula parser is returned. Otherwise an Err object is returned with #PARSE_ERROR as error string.

5.4 EVALUATING FORMULAS

An evaluator is needed to calculate the value of a formula in the context of the spreadsheet that the formula is part of. Every spreadsheet object has its own evaluator, because the value of a formula depends on the values of the referenced cells in that spreadsheet. The evaluator takes an object of type `Formula` as input. This is the parsed representation of the formula that the user entered in the formula box. The output of the evaluation method is of type `Observable[Value]`. This is a key difference with the Excel add-in. In ExcelStreams every formula evaluates to a single value, whereas in the StreamSheets application every formula evaluates to a stream of values.



In a spreadsheet application, the output of a function can always be used as an argument for another function, i.e. spreadsheet functions can be nested. Therefore, the arguments of a function should always be of type `Observable[Value]` too. Single values are wrapped in observables that only emit one item.

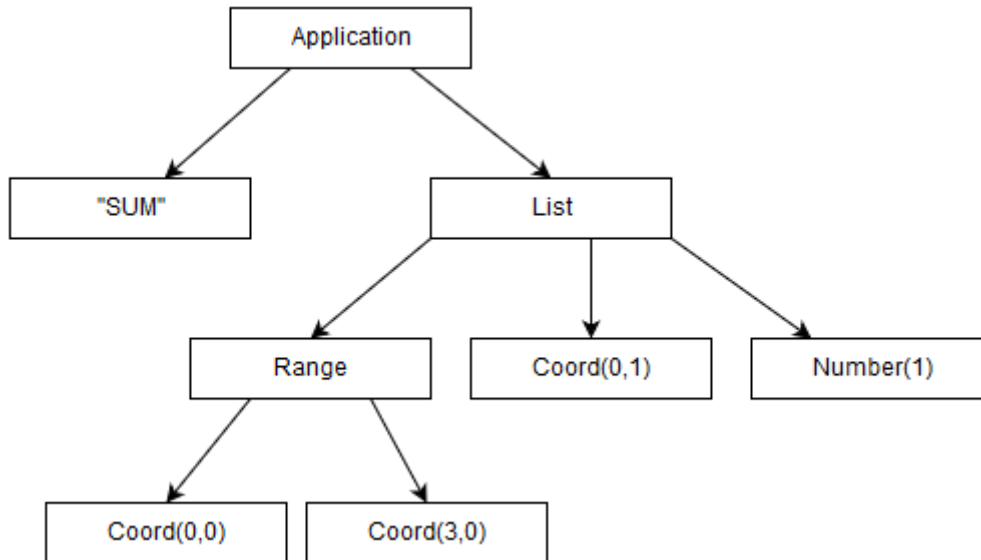
The main advantage of this evaluator is that a formula has to be evaluated only once, even if the content changes of the cells on which the formula depends. The evaluator combines the argument observables in a new observable, so that a change in one of the source observables will be propagated to the new observable automatically without reevaluating the formula. As a consequence, it is not needed to keep track of the dependencies between cells. This is implicitly done by RxScala.

Values in spreadsheet cells can be of different types. The types available in StreamSheets are:

- `EmptyValue()`
- `ErrorValue(v: String)`
- `BoolValue(v: Boolean)`
- `NumericValue(v: Double)`
- `TextValue(v: String)`
- `ListValue(v: List[Value])`

These types are all subtypes of `Value` and have string representations that are used to print the values to the cells in the UI. The evaluator transforms a formula into an observable of values. A number is converted into an observable that emits one `NumericValue`. Empty formulas, strings, booleans and error formulas are evaluated similarly. A coordinate formula retrieves the value stream of the referenced cell. Function application is a bit more complicated. First the list of arguments is evaluated to a list of observables of values. An argument could also be a range. A range is converted into a list of evaluated coordinates that are in that range, and the list is flattened with the other evaluated function arguments. Then the function is applied to the list of evaluated arguments. The argument observables are combined and result in a new observable.

The following example describes how a formula that is entered by the user results in an observable of values. Assume the user enters the formula **=SUM(A1:A3;B1;1)** in cell C1. The parser processes the string and returns a formula with the following structure:



The evaluator takes the root formula, which is a function application. The first thing that is done for function applications is evaluating the arguments one by one. The first argument is the range A1:A3. It iterates over the cells that are in that range, retrieves the value streams of all these cells and collect them in a list. The result type of the evaluated range is `List[Observable[Value]]`. The second argument is a reference to B1. The evaluator retrieves the value stream of that cell and wraps it in a list, so that its result type is `List[Observable[Value]]` too. The third argument is a simple number, that evaluates to an observable that emits one value. This observable is wrapped in a list as well. When all function arguments are evaluated, the resulting lists are flattened to a single list. The only thing left is applying the SUM function to the `List[Observable[Value]]` from the arguments.

The evaluator contains a map with a definition for all the available spreadsheet functions. The type of this map is: `Map[String, List[Observable[Value]] => Observable[Value]]`. The first string is the name of the function. We get the element from the map with the name SUM, and the result is a function from a list of observables of values to an observable of values. The input for this function is the list of evaluated arguments. The SUM operation folds over the list and combines the observables in the list with the `combineLatest` operator from RxScala by adding up the elements. If one of the source observables emits a new item, it is combined with the latest values from the other source observables and the sum of these values is recalculated. The result observable from the fold operation is the final result of the evaluation of this formula. The value stream of the cell that called this function will subscribe to this observable.

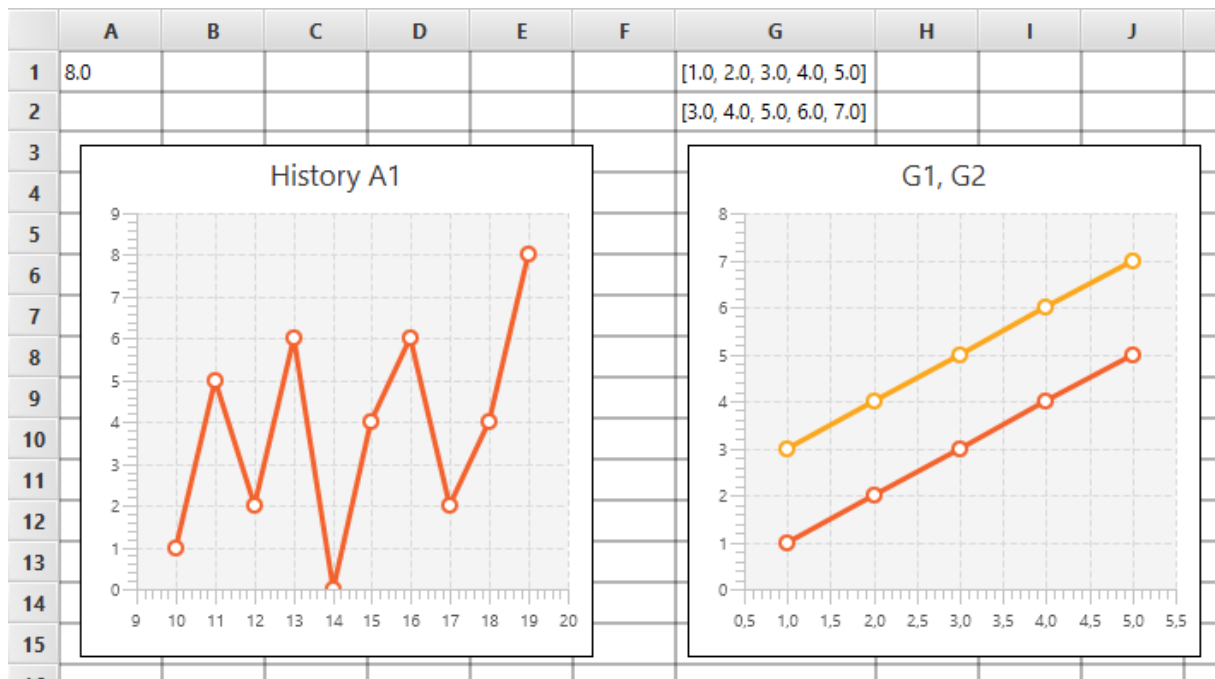
5.5 LIST OF FUNCTIONS

This is an alphabetical list of all functions that are available in StreamSheets.

Function	Description
ABS(number)	The absolute value of the number
ADD(number; number)	The sum of two numbers
AVERAGE(number; ...)	The average of the numbers in the argument list
BUFFER(stream; count; skip)	Periodically collects <i>count</i> items from the source stream and emits these collections as lists. The skip argument (optional) defines how many items should be skipped before starting to collect a new list
CONCAT(text; text)	Concatenates two strings into one
DIVIDE(number; number)	The first number divided by the second number
FILTER(stream; boolean)	Emits the items from the source stream if the second argument is true
GT(number; number)	Checks if the first number is greater than the second number
GTE(number; number)	Checks if the first number is greater than or equal to the second number
LT(number; number)	Checks if the first number is less than the second number
LTE(number; number)	Checks if the first number is less than or equal to the second number
MAX(number; ...)	The maximum of the numbers in the argument list
MERGE(stream; ...)	Merges all items emitted by the observables in the argument list
MIN(number; ...)	The minimum of the numbers in the argument list
MINUS(number; number)	The difference of two numbers
MOD(number; number)	The modulo of two numbers
MULTIPLY(number; number)	The multiplication of two numbers
POW(number; number)	The first number to the power of the second number
PRODUCT(number; ...)	The product of all numbers in the argument list
RANGE(from; to)	A list of numbers between <i>from</i> and <i>to</i> (inclusive)
STREAM(url)	Subscribes to an external data stream
SUM(number; ...)	The sum of all numbers in the argument list
SUM.RANGE(listOfNumbers)	The sum of a list of numbers

5.6 CHARTS AND LISTS

In StreamSheets it is also possible to visualize the history of a cell in a history chart. The chart can be inserted by right-clicking a cell and selecting the option show history from the context menu. At the moment that the chart is created, the data series of the chart subscribes to the value stream of that cell. From this moment new values of the cell will be added to the chart. The default maximum number of data points visible in the chart is set to 10. If a new value is added the chart shifts to the left and the most left data point disappears. Non numeric values are plotted as zero.

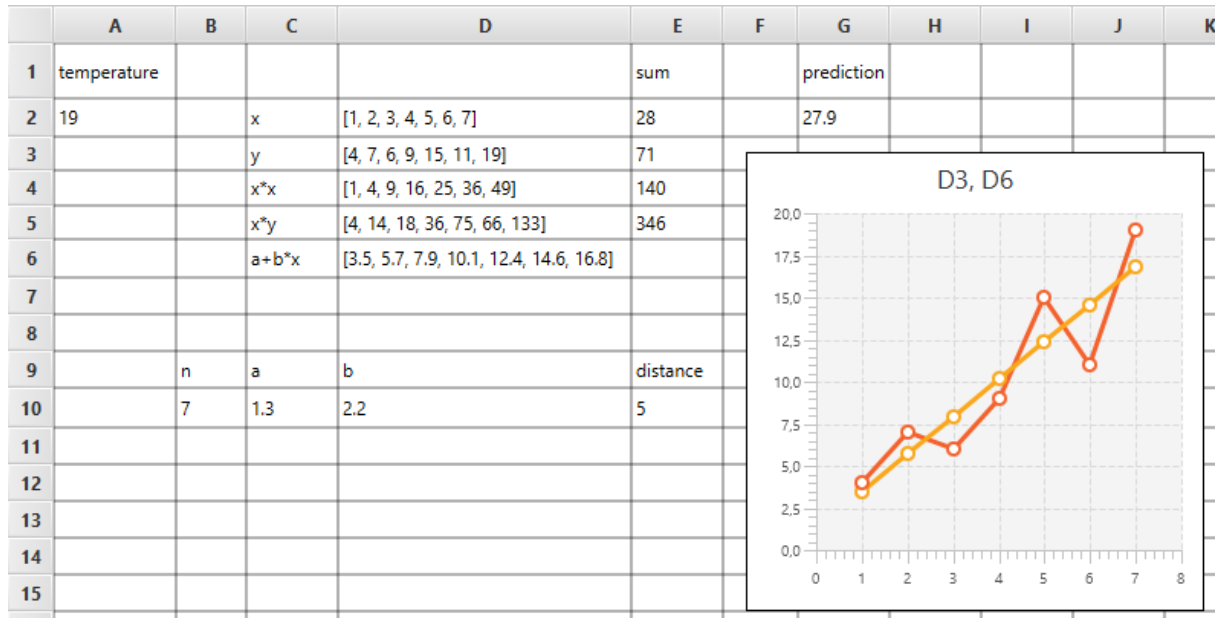


Lists are native data types in StreamSheets. The example in this figure shows the string representation of two lists in cells G1 and G2. The value stream of both of these cells emitted the list as a ListValue (see section 5.4). If another cell refers to one of these cells, the ListValue is used in the calculation. Hence, it is not needed to convert the string representation into the underlying list, as we had to do with the function L2R or L2A in Excel. On the other hand, the ListValue in StreamSheets is not the same as a range of cells and that is why functions must be defined to explicitly support lists if desired. Another option is to create specific functions that work on lists, which was done with the SUM.RANGE function.

Lists can be plotted in a line chart too. The chart gets the latest value from the referenced cell and transforms the list into a data series object. If the value is not a list but a single number, it is converted into a list with one item. Multiple lists can be plotted in one chart as well. This is illustrated in the figure above.

5.7 EXAMPLE

This section provides an example that demonstrates many of the features of StreamSheets that were described in the previous sections. The example is based on one of the case studies in the paper "Stream Processing with a Spreadsheet" by M. Vaziri et al.^[1] The spreadsheet analyzes and visualizes the trends in a stream of temperature values and predicts the temperature at a point in the future.



The details of this application are explained in section 4.5.2, where this example is implemented with ExcelStreams. This part focuses on the specific implementation in StreamSheets and the differences with ExcelStreams. Cell D3 contains a list with the last seven values from the temperature stream in A2 and is calculated with the BUFFER function. The biggest difference with Excel is that the resulting list from the BUFFER function does not need to be separated into a range of cells, because the arithmetic operations such as addition and multiplication, that must be applied to calculate the regression line, are defined for lists too.

The arithmetic binary operators are defined as follows:

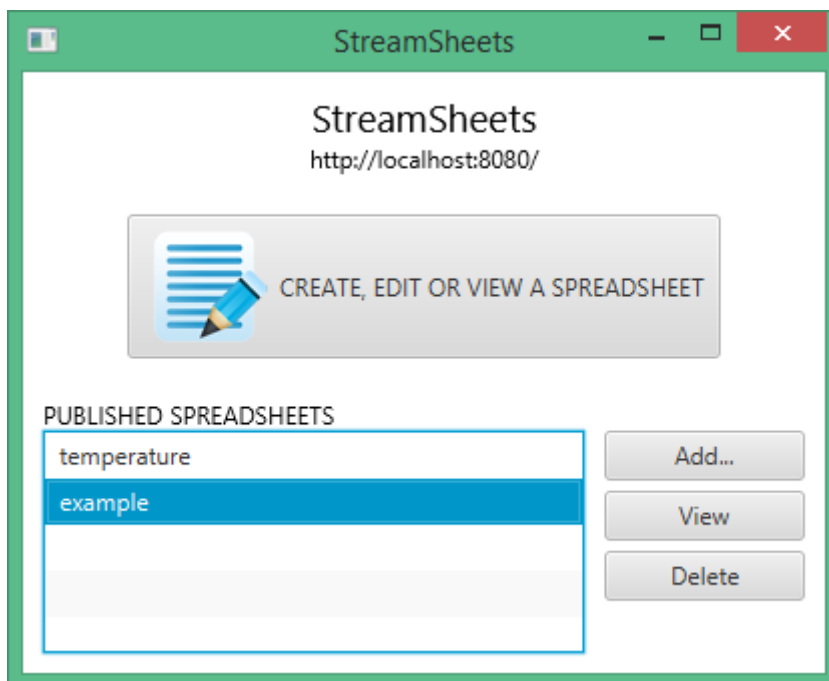
number <op> number	This is the standard situation. The binary operator is applied to the two numbers and results in a new number.
list <op> number or number <op> list	If one of the arguments is a list, it will map over that list and applies the operation to every element of the list with the number.
list <op> list	If both arguments are lists, the lists are zipped and the operation is applied pairwise. If one list is longer than another, the longest list is truncated.

Another difference with Excel is that lists are native data types in StreamSheets. These lists can be plotted in a chart directly, and do not have to be separated into a range of cells. The chart in this examples plots the list with the latest data points from the temperature stream collected in D3, together with the calculated least-square regression line from D6.

5.8 EXTERNAL DATA STREAMS AND PUBLISHED SHEETS

In the StreamSheets application it is possible to subscribe to an external data stream with the `STREAM` function, much similar to the `RX_STREAM` function in ExcelStreams. The only argument in the `STREAM` function is the URL that specifies the location of the source stream. The source stream sends JSON objects as server sent events to the client. The format of the JSON objects and the decision to use server sent events for this purpose are explained and motivated in section 3.2. With SSE the client opens a permanent HTTP connection with the server and the server pushes new messages to the client over this connection as they become available. The SSE client is implemented with Jersey^[9], an open source framework to develop RESTful web services in Java. The `STREAM` function sets up an `EventSource` on the specified URL that receives the SSE messages with the `onEvent` listener. The JSON messages are converted into a value object and are emitted by the resulting observable.

On startup of the spreadsheet application, a dialog pops up that asks for a port number. It starts a localhost Jetty server that can be used to expose the cells in a spreadsheet as external data streams. By right-clicking on a cell, one can copy the URL of a cell and use this URL in the `STREAM` function of another instance of the StreamSheets application to subscribe to the value stream of that cell. The data streams in a spreadsheet are not immediately visible for other applications. The main screen of the StreamSheets application contains two sections.



The button at the top opens the spreadsheet view, and allows the user to create new sheets or edit and view existing sheets. The second part contains a list of published spreadsheets that are accessible from other applications.

Spreadsheets can be saved as XML files. The XML file stores all non-empty cells from the spreadsheet together with its row and column coordinates and the formula of that cell. This is a part of the XML file from the example application from section 5.7.

```
<sheet>
  <cell col="0" row="0"><formula>temperature</formula></cell>
  <cell col="4" row="0"><formula>sum</formula></cell>
  <cell col="6" row="0"><formula>prediction</formula></cell>
  <cell col="2" row="1"><formula>x</formula></cell>
  <cell col="3" row="1"><formula>=RANGE(1;B10)</formula></cell>
  <cell col="4" row="1"><formula>=SUM.RANGE(D2)</formula></cell>
```

When the XML file is loaded in an empty spreadsheet, it iterates through the list of cells and reevaluates the formulas.

An XML file can be added to the list of published spreadsheets, to make it accessible for other applications. It creates a spreadsheet object from the XML file that runs in the background without a spreadsheet view. The value streams of the cells in the spreadsheet will still emit new values, but there is no view subscribed to receive those values and visualize the sheet. One can still view and edit a published spreadsheet if desired, but the advantage of running the spreadsheets without a view, is that many sheets can be published together in one instance of the StreamSheets application. This makes it possible for users to create a sheet locally, and then publish it to a central StreamSheets application that runs on a server.

The Jetty server runs a servlet, that specifies the URL that is used to publish the data streams and uses Jersey to send server sent events. The format of the URL is:

<http://localhost:8080/events?sheet=temperature&cell=A1>

In this example, temperature refers to the name of the published sheet, and A1 refers to the cell within that sheet. The event sender subscribes to the value stream of this cell. For every value it receives, it converts that value into its JSON representation and then sends the JSON object to the client over the SSE connection.

6 GOOGLE SHEETS

Google Sheets is a popular web-based spreadsheet application that is part of the Google Docs suite. This chapter explores whether it is possible to implement the spreadsheet model for stream processing in combination with Google Sheets. Two alternatives are explored. A first approach is to use the Google Sheets API to connect a client application to Google Sheets. The other option is to extend Google Sheets with an add-on that is written in Google Apps Script.

6.1 GOOGLE SHEETS API

The Google Sheets API can be used to connect a client application to a spreadsheet in Google Sheets. A GData client library is available for Java that makes it easier to communicate with the Google Sheets API. Most important features from the Sheets API are adding, deleting and modifying worksheets, and reading and writing data from and to the cells of a worksheet. The Google Drive API must be used to create or delete entire spreadsheets. A user has to authenticate with OAuth 2.0 to gain access to the non-public sheets of the user via the Google Sheets API.

One option to implement data streams in Google Sheets with the API is creating a client application in Java or Scala, where the user defines data streams as Rx observables and then attaches these observables to specific cells in a Google worksheet. Whenever an observable emits a new item, the API is used to send the new value to the worksheet and updates the value of that specific cell. However, this approach is very limited. Observables cannot be created within the context of a worksheet, because it doesn't know anything about the values of the other cells. Furthermore, the data streams are still represented as single values in the Google sheet. The formula of the cell that receives the updates from the observable only shows its latest value, but it does not indicate that it is subscribed to an external observable. Besides, Google Sheets has a function named IMPORTRANGE that provides the functionality to import or stream data from one Google sheet to another. All together, this approach does not offer much more than the IMPORTRANGE function and therefore is not very useful.

Another option is to create a Java or Scala spreadsheet application that is connected to Google Sheets at the back end. The user interacts only with the Java/Scala application, but the data is stored in a Google Sheet. The Google Sheets API is used to send values that have been changed from the front end application to the back end sheet. Formulas that are valid in Google Sheets could then be used in the client application too. The formula would be sent to and evaluated by Google Sheets and the calculated value would be returned to the client application. This way all functions that are available in Google Sheets are also accessible from the client application. The StreamSheets application described in chapter 5 could be extended with Google Sheets running at the back end. The gain would be a significant increase in the number of spreadsheet functions that can be used. However, there are some important drawbacks:

- Every time a value changes it should be sent to the Google Sheet, because the Google Sheet functions of the cells that depend on the changed cell should be re-evaluated. Since it is not possible in the API to only get the dependents of a cell, all Google sheet cells must be retrieved by the client application after a value changes in order to update its state. The problem is that this process of sending and retrieving information via the Sheet API is time-consuming due to network latency and this makes the application unresponsive.

- Another drawback is that a distinction must be made between native functions and Google Sheet functions, because Google Sheet functions must be evaluated by Google, whereas native functions must be evaluated by the client application. We have seen with the Excel add-in that this distinction leads to many complications. Additionally, there is the limitation that native functions and Google sheet functions cannot be combined in one formula.

In conclusion, the advantage of having more functions available does not compensate for the increased complexity and additional limitations. It would be better to extend the StreamSheets application itself, by implementing more native functions.

6.2 GOOGLE SHEETS ADD-ON

Google introduced add-ons for Sheets in 2014. While the Google Sheets API is made to connect a client application to a Google Spreadsheet, an add-on is an extension that runs inside Google Sheets. Add-ons in Google Sheets are equivalent to Excel add-ins. However, add-ons are built with Google Apps Script, a scripting language that is executed in the Google Cloud and that is based on JavaScript.

With Google Apps Script it is possible to create custom functions (user-defined functions) that can be called directly from a cell in a spreadsheet. An example of a custom function defined in Google Apps Script is:

```
function INCREMENT(n) {
  return n+1;
}
```

This is a simple function that takes one argument and increments that number by one. The function can be called from any cell in any worksheet as long as the add-on is loaded for that spreadsheet. Besides adding custom functions, one can use Google Apps Script to create custom menu items and embedded charts too. There are also some triggers or events available that are fired for instance when the user opens a sheet or changes the content of a cell. It is not possible to change existing features from Google Sheets.

There is one fundamental problem if we want to implement a model for stream processing as add-on in Google Apps Script. The script runs on the server side, and every time the spreadsheet requests the execution of a function in the script, it creates a new server-side context to run that function. It is not possible to preserve state or data or create variables that live as long as the spreadsheet is opened.

As an example assume we have the following script, that defines a global variable and a custom function that returns a value that depends on the value of the global variable:

```
var i = 0;

function GET_NEXT() {
  i++;
  return i;
}
```

The function `GET_NEXT` will always return a 1, regardless of how many times or from how many cells this custom function is called. That is because every function call runs in a new context, so in practice the global variable is reinitialized to 0 every time.

There are some solutions to store data more persistently, for example as key value pairs with the Properties Service. Nevertheless, it is too limited to implement reactive data streams in Google Apps Script. Every cell should have an observable, that represents the value stream of that cell. The observable should be running in the background continuously, so that it can push new values into the cells of the spreadsheet as they become available. For this reason, it is not possible to implement an add-on that is similar to the ExcelStreams add-in for Excel.

7 RELATED WORK

There are many initiatives that intend to make it easier for developers to create stream processing applications, such as Spark Streaming for Java, StreamInsight for C# and IBM's Streams Processing Language. However, these approaches require developers to learn new techniques or programming languages. Since there are many more spreadsheet users than developers with the skills to create these kinds of applications, some of the recent research efforts in this field focus on spreadsheets as a platform for developing streaming applications.

Gneiss is a system to create web applications by linking web UI elements to spreadsheet cells. Chang and Myers present an extended version of this system in the paper "A Spreadsheet Model for Handling Streaming Data".^[2] They describe a spreadsheet model with a technique to stream data from web services to a spreadsheet and it can be used to stream data from web input elements to a spreadsheet as well. In contrast to our spreadsheet model, the technique in this system to stream data from web services to the spreadsheet is pull based. It sends a web request with regular configurable time intervals and adds the results to the spreadsheet.

ActiveSheets^[1] is another programming platform for stream processing based on Microsoft Excel presented in the paper "Stream processing with a spreadsheet" by Vaziri et al. Live data can be imported into an Excel worksheet and standard Excel formulas and functionalities can be used to perform operations on the data streams and visualize the data with Excel charts. Computations can also be exported to run on a server, although only a subset of the Excel features are supported by the server.

Both in the Gneiss spreadsheet model and in ActiveSheets, a data stream in a sheet is represented as a range of adjacent cells in a column. A data stream in Gneiss is bound to a column in the worksheet and stacks the column with the latest values pulled from the web service. In ActiveSheets one first selects a window in the spreadsheet (a range of cells in a column) and then subscribes to a data stream from the server. New values are added to the window from bottom to top and it continues scrolling, so the size of the window decides how many historical values will be kept.

While Gneiss and ActiveSheets make it possible to import external data streams into a range of cells, data streams in our spreadsheet model are more ubiquitous. In our model every cell represents a data stream itself, which means that a tabular sheet has an additional time dimension. One can subscribe a cell to an external data stream and publish the data stream of a cell to make it available for other users. Streams can also be passed as arguments to spreadsheet functions and be returned by spreadsheet functions. A buffer operator can be applied on a data stream to obtain historical values and to simulate the sliding window feature of ActiveSheets. A major benefit of this approach is that the size of the window has not to be predetermined and that it stores only the historical data that is being used.

Popular spreadsheet applications like Microsoft Excel and Google Sheets provide some native functions to import data from external sources into a spreadsheet. Blockspring^[10] is a plugin that extends this functionality and makes it possible to access web services from a spreadsheet with a simple formula. These functions are useful to import data from external sources into a range of cells and update it with the latest values. However, the functions are pull based and historical values are not preserved, since the data within the spreadsheet is not represented as a stream.

8 CONCLUSION AND FUTURE WORK

The main contribution of this thesis is a novel spreadsheet model for real time stream processing with Rx. The spreadsheet model we developed and described answers the three sub-questions from the introduction.

1. **How can we introduce higher order streams in spreadsheets, i.e. how can we represent data streams as native data types, so that they can be used as arguments to spreadsheet functions and so that they can be returned by spreadsheet functions?**

The spreadsheet model consists of (user-defined) functions that take data streams as arguments and return data streams as well. Data streams are represented as observables from the Reactive Extensions framework. If a simple single value is passed as argument to a function, it is first converted into an observable that emits that single item. A function argument can also be a reference to another cell, since every cell represents a data stream that consists of the values of that cell over time. Moreover, spreadsheet functions can be nested in one formula, because a function returns a data stream that can be passed as argument to another function. A spreadsheet function combines the input observables and applies (a sequence of) Rx operators to calculate the result stream.

2. **How can we preserve historical data of a spreadsheet without storing large amounts of superfluous data or how can we add an extra time dimension to a two-dimensional tabular sheet?**

Every cell in a sheet stores a value stream implemented as a behavior subject, that emits the values from that cell. The Rx buffer operator can be applied to a value stream to obtain historical values of a cell. The buffer operator periodically collects items from a source observable and emits these collections as lists and it can be used to implement sliding windows. The historical data of a cell will be kept, only if one or more buffer operators are applied to its value stream and if the result stream of a buffer operator emits a new list, the previous list will be discarded. Therefore, only the historical data that is being used in the spreadsheet at that moment will be stored.

3. **How can we import external data streams live from the internet by only using spreadsheet formulas and without the need to continuously check for updates?**

The technology of server sent events is used to send data from a server to a spreadsheet client over an HTTP connection. With server sent events, the server pushes the data to the client as it becomes available. The client does not have to continuously check for updates, but opens a permanent connection to the server and gets notified when new data is available. The user subscribes to an external data stream with a spreadsheet formula and passes the URL of the data source as argument to the function. The spreadsheet formula creates and returns an observable that emits the server sent events that it receives. The SSE connection will be closed when the user unsubscribes from the observable.

In addition to the features described here, the spreadsheet model also defines how internal data streams can be published. The spreadsheet application runs a server in the background. When the server receives an HTTP request, an SSE connection will be opened to the client and when the requested data stream emits a new value, it is sent as a server sent event to the client. A published data stream can be imported directly as external data stream in another instance of

the spreadsheet application, so that the program can be used as a distributed spreadsheet application.

The spreadsheet model presented in this thesis report does not only add new functionality for stream processing. It also simplifies the traditional spreadsheet model, because manual dependency management is not needed with the spreadsheet model for data streams. Function arguments that are references to other cells, refer to the value streams of these cells instead of their current values. Future values are already included in these value streams. As a consequence, dependent cells do not have to be re-evaluated if the content of a cell changes. These dependencies are implicitly handled by the Rx framework.

As part of the thesis we developed two implementations of the spreadsheet model: ExcelStreams and StreamSheets. The first is an add-in for Microsoft Excel, the latter a new spreadsheet application built in Scala. Excel is among the most popular spreadsheet applications. The advantage of the implementation for Excel is the wide range of functionalities that are already available in Excel. Users are more likely to install an add-in for Excel than to switch to another spreadsheet application, in particular if it would provide less functionality. However, we have seen that the Excel implementation comes with some significant limitations. This is due to the fact that we have to combine the traditional spreadsheet model with the stream processing model and the distinction between native Excel functions and user defined functions. Moreover, functions in Excel cannot return data streams, so they must be recalculated every time a stream emits a new value. With this restriction, the Excel add-in does not fully benefit from the dependency management that is done by Rx. These limitations do not appear in the StreamSheets application. This application implements the stream processing model without complicated workarounds. In contrast to ExcelStreams, it contains an evaluator that returns a data stream instead of a single value and most importantly there is no distinction between native functions and user defined functions. Many features that are available in spreadsheet applications such as Excel are (still) missing, but the implementation is much cleaner, which is essential for the development of a software application in the long-term.

Both ExcelStreams and StreamSheets are far from perfect. For ExcelStreams it would be interesting to explore how the limitations listed in section 4.6 could be solved. One of the requirements of the spreadsheet model is that we do not want to store large amounts of historical data that is not used. In the Excel add-in this is only partially true. Lists that are emitted by a buffer operator are transformed into a range and added to a hidden sheet MyLists, but they are not removed. It should be further explored how previous emitted lists or ranges can be cleaned up after a buffer operator emits a new list. Further research on the StreamSheets application could focus on extending its functionality and investigating whether more common used spreadsheet functions could be implemented with Rx.

The spreadsheet model itself could be extended and formalized as well. An open question is how multiple values can be streamed together at the same tick in combination with the server sent events technology. One direction is to collect these values in a list and stream the entire list as a value. Another idea for future research is to test how the spreadsheet model performs on a larger scale and how we can optimize the model to increase performance, for example in combination with the addition or deletion of rows and columns.

REFERENCES

- [1] Vaziri, Mandana, et al. "Stream processing with a spreadsheet." *ECOOP 2014–Object-Oriented Programming*. Springer Berlin Heidelberg, 2014. 360-384.
- [2] Chang, Kerry Shih-Ping, and Brad A. Myers. "A Spreadsheet Model for Handling Streaming Data." *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015.
- [3] ReactiveX "An API for asynchronous programming with observable streams"
<http://reactivex.io/>
- [4] ReactiveX "The Observable Contract" <http://reactivex.io/documentation/contract.html>
retrieved on December 1, 2015
- [5] Excel-DNA "Free and easy .NET for Excel" <http://excel-dna.net/>
- [6] EventSource4Net "An eventsource (Server-Sent Events client) implementation for .Net"
<https://github.com/erizet/EventSource4Net>
- [7] ServerSentEvent4Net "Server-Sent Event(SSE) implementation for ASP.NET WebApi"
<https://github.com/erizet/ServerSentEvent4Net>
- [8] Odersky, Martin, Spoon, Lex, and Venners, Bill. "The SCells Spreadsheet" *Programming in Scala Second Edition* ISBN 9780981531649
- [9] Jersey RESTful Web Services in Java "Server-Sent Events (SSE) Support"
<https://jersey.java.net/documentation/latest/sse.html>
- [10] Blockspring "Access web services from spreadsheets" <https://www.blockspring.com/>
retrieved on December 4, 2015

