Stream Processing with a Spreadsheet

Mandana Vaziri, Olivier Tardieu, Rodric Rabbah,
Philippe Suter, and Martin Hirzel

IBM T.J. Watson Research Center, Yorktown Height, NY, USA
{mvaziri,tardieu,rabbah,psuter,hirzel}@us.ibm.com

Abstract. Continuous data streams are ubiquitous and represent such a high volume of data that they cannot be stored to disk, yet it is often crucial for them to be analyzed in real-time. Stream processing is a programming paradigm that processes these immediately, and enables continuous analytics. Our objective is to make it easier for analysts, with little programming experience, to develop continuous analytics applications directly. We propose enhancing a spreadsheet, a pervasive tool, to obtain a programming platform for stream processing. We present the design and implementation of an enhanced spreadsheet that enables visualizing live streams, live programming to compute new streams, and exporting computations to be run on a server where they can be shared with other users, and persisted beyond the life of the spreadsheet. We formalize our core language, and present case studies that cover a range of stream processing applications.

1 Introduction

Continuous data streams are ubiquitous: they arise in telecommunications, finance, health care, and transportation among other domains. They represent such a high volume of data that they cannot be stored to disk in raw form, and it is often crucial for the data to be analyzed right away. Stream processing is a programming paradigm that processes sequences of data immediately, and enables what is called continuous analytics.

In organizations that require stream processing, domain experts may have limited programming experience to directly implement their desired solutions. As a result, they rely on developers for the actual implementation. Our objective is to make it easier for these end-users to directly prototype and perform computations on live data. We believe this is an important facilitator for rapid turnaround and lower development costs that may otherwise hinder streaming data analysis.

This paper proposes the use of spreadsheets as a stream programming platform. The choice of spreadsheets stems from the fact that they are a pervasive tool used in many different domains, and are familiar to non-programmers[1].

[1] There are 9 million Java developers
(http://oracle.com.edgesuite.net/timeline/java/),
and an order of magnitude more Microsoft Excel users.
(http://blog.ventanaresearch.com/tag/microsoft-excel/)

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 360–384, 2014.

Spreadsheets offer a variety of visualization possibilities, and the ability to analyze, process, or augment source data by entering formulas in cells. They provide a unique interface where data is in the foreground and the code that produced it can be viewed in the same place. This is unlike common integrated development environments (IDEs) where code appears in a dedicated editor, and data visualization plays a subordinate and often orthogonal role.

Although spreadsheets are used for many different applications, they do not readily support online stream processing, which we believe requires the following essential features:

- *Live data in cells:* for online processing, one must have the ability to import live data into cells. Further, as the live data changes, the value of the cell must change contemporaneously.
- *Segmenting streams into windows:* some streaming operations are applied over aggregates of values (e.g., reductions). In spreadsheets, aggregates are groups of rows and columns called ranges. For online stream processing, an analogue between spreadsheet ranges and windows over streams is needed.
- *Stateful cells:* spreadsheets are functional by nature and do not readily support state or cyclic cell references. However, many stream processing applications need state to compute summaries or decisions via finite state machines.

This paper presents ActiveSheets, a programming platform for stream processing that is based on Microsoft Excel with enhancements to meet the challenges described above. It provides a language that an end-user can use to easily populate ranges of cells in a spreadsheet with the desired shape of data, a windowing mechanism that allows computations over windows of streaming data, and the ability to perform stateful computations by treating stateful and stateless cells uniformly. ActiveSheets retains and interoperates with familiar Excel features (e.g., built-in functions and macros, or visualizing live data) but also enhances Excel's native capabilities such that they operate correctly on live data. An example is the Excel *pivot* function which classically operates on a snapshot of cells (i.e., if the cells change, the filtered results do not). In ActiveSheets, it is possible to continuously pivot as the input cells change.

ActiveSheets is a client-server architecture in which the server publishes streams and the client, namely the spreadsheet, allows the user to subscribe to streams and operate on the *live* data; operations include visualization of streams and generation of new streams (Fig. 1). The client provides an export feature, making it possible to share the results with other users, as well as persisting the computation on the server, beyond the life of the spreadsheet.

We present formal semantics for the core language captured by our user interface, which we call the *spreadsheet calculus*. This is a reactive programming model that represents the spreadsheet computation as a combinatorial circuit derived from cell dependencies and formulas contained within the cells. As input cells change over time, any dependent cells are automatically recomputed and updated. Cells that must retain state can be viewed as circuits with latches. This model hides many common concerns from the programmer, because it offers a fixed control structure and manages cell updates automatically based on data

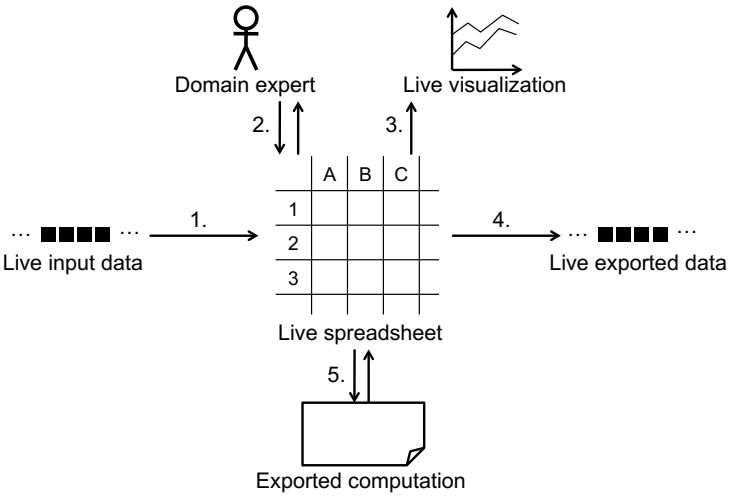


Fig. 1. ActiveSheets Overview: 1. The server publishes lives streams. 2. The domain expert subscribes to these streams and prototypes the computation in a spreadsheet. 3. Spreadsheet functionality is readily available, including visualization. 4. Data computed in the spreadsheet may be exported as its own stream. 5. The entire spreadsheet may be exported to the server, where the computation outlives client shutdown.

dependencies. As a result, the domain expert can focus on the data transformations they wish to compute.

A spreadsheet enables a live programming platform, meaning that code can be modified during the execution of the program. This is an essential feature, because streaming analytics applications cannot be stopped and restarted easily. The user has to be able to quickly modify computations without stopping data sources. This feature creates challenges, especially in the face of stateful computations, and we define its semantics formally in the spreadsheet calculus. Finally, our extensions to the spreadsheet must preserve its highly interactive nature, meaning that on every update to a cell, there can only be a bounded amount of computation and memory usage. We prove this property for our core language, and show that it is also deterministic, meaning that for any given set of inputs, the spreadsheet computation always yields the same result.

This paper makes the following contributions:

- A reactive programming model for stream processing based on spreadsheets and a uniform treatment of stateless and stateful cells.
- Formal semantics for our core language using a new spreadsheet calculus.
- Exporting spreadsheet computation to the server for sharing or persistence.
- A prototype implementation using Microsoft Excel, and case studies covering a range of stream processing applications.

2 Overview

This section presents an overview of how ActiveSheets works, using a streaming stock bargain calculator as a running example. The bargain calculator takes two input streams: `Trades` and `Quotes`. A stream is an infinite sequence of tuples, which are sequences of attribute/value pairs. A feed is the infinite sequence of values corresponding to a single attribute of a stream. Thus a stream is comprised of a collection of feeds whose values update synchronously.

The tuples of the `Trades` stream represent actual trades that have been made, using attributes `sym` (a stock symbol), `ts` (a timestamp), `price`, and `vol`. Each of these attributes defines a feed of values. The bargain calculator first computes the Volume Weighted Average Price (VWAP). Given a window of prices P_i and volumes V_i, the VWAP is defined as:

$$\text{VWAP} = \frac{\sum_i P_i \times V_i}{\sum_i V_i}$$

After computing the VWAP over the `Trades` stream, the bargain calculator determines whether or not each price in the `Quotes` stream is less than the VWAP. If yes, it outputs a bargain. Various streaming languages are well-suited to writing this program, such as CQL [4] or SPL [15]. However, end-users are typically unfamiliar with programming languages, let alone special-purpose languages such as CQL or SPL. Our objective is to bring stream programming to the end-user by enhancing the spreadsheet, a tool that is pervasive and familiar.

ActiveSheets is based on Microsoft Excel enhanced with controls for manipulating live streams as shown in Fig. 2.

Fig. 2. ActiveSheets Controls. Buttons from left to right: connect to and disconnect from the server, add a stream ('+' icon), pause a stream (pause symbol), disconnect from a stream ('−' icon), export data back to the server (flash symbol), stop data export (crossed out flash symbol), export computation (movie symbol), and lastly, debug mode (light bulb), used to debug the implementation of ActiveSheets.

Fig. 3 shows the bargain calculator program in ActiveSheets. We now explain how the user can obtain this program step by step.

Connecting to the server. To start using ActiveSheets, the user first clicks on the connect button. This prompts for the address to the server and connects to it. The server publishes several streams that the client may subscribe to, visualize, and work with. Depending on the server's installation, these streams could come from existing stream processing programs, live feeds, static data that is streamed, or exported streams from other ActiveSheets clients. In the case of this example, the server publishes the two input streams `Trades` and `Quotes`.

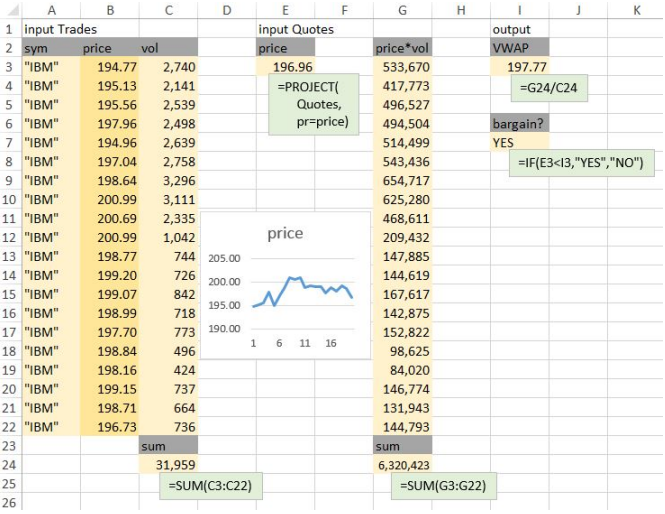
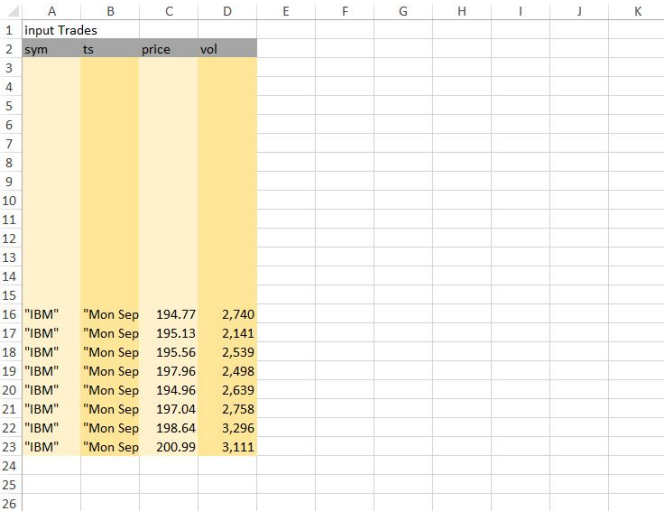


Fig. 3. Bargain Calculator in ActiveSheets

Subscribing to a stream. The next step is to subscribe to a stream. To do this, the user first chooses a window in the spreadsheet, then presses the subscribe button ('+'), and enters the stream name at the prompt. The selected stream is then displayed in the window that the user selected with one column per attribute (feed), and the values scroll from bottom to top. A visual indicator comes on if the user did not select a wide enough range of cells. At any given moment in time, the user sees a window of data that gets updated continuously. In the example, the user first subscribes to the `Trades` input stream. Fig. 4 shows the `Trades` input streaming into the spreadsheet in columns A through D over a window of size 20. The data fills the window from bottom to top and continues scrolling. The chosen window size not only specifies how much of the stream is shown at any given moment in time, it also determines the window of data over which the VWAP will be computed. The user may pause a stream by choosing a cell in it, and pressing the pause button. This causes all the feeds in that stream to stop until the user presses pause again to resume, which causes ActiveSheets to display the latest live data.

Adding new feeds. The user can create new data by entering formulas in cells directly, which creates new feeds. Fig. 5 shows how the user enters a standard Excel formula to compute the price times the volume in cell G3. Notice that, in this figure, the timestamp column has been deleted because it is not needed. The user then copies and pastes the formula in the rest of column G with familiar Excel gestures. Even though familiar controls are used to populate column G, the result is live in ActiveSheets: as the values of price and volume are updated, their product is recomputed. Fig. 5 further shows how the user can compute the sum for the volume and price-times-volume columns (cells C24 and G24),



	A	B	C	D	E	F	G	H	I	J	K
1	input Trades										
2	sym	ts	price	vol							
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16	"IBM"	"Mon Sep	194.77	2,740							
17	"IBM"	"Mon Sep	195.13	2,141							
18	"IBM"	"Mon Sep	195.56	2,539							
19	"IBM"	"Mon Sep	197.96	2,498							
20	"IBM"	"Mon Sep	194.96	2,639							
21	"IBM"	"Mon Sep	197.04	2,758							
22	"IBM"	"Mon Sep	198.64	3,296							
23	"IBM"	"Mon Sep	200.99	3,111							
24											
25											
26											

Fig. 4. The `Trades` input streaming in

and enter a formula for the VWAP (cell I3). Each feed in ActiveSheets gets updated at specific points in time, which we call its *tick*. For example, the sum of two cells gets updated whenever either of the cells are updated.

Adding new streams. In addition to entering formulas in cells one at a time, the user can also populate a range of cells with a stream (synchronous feeds) using ActiveSheets' query language. This language is relational in flavor, and includes operators for projection, selection, deduplication, sorting, pivoting, and aggregation. It also supports a simple mechanism for stateful computation. Queries are entered by selecting a window in the spreadsheet and pressing the '+' icon. The simplest query is giving the name of a stream to display all of its attributes. The user may use a selection to filter tuples in `Trades` with a price greater than a certain value

```
select(Trades, price > 200)
```

which would populate a range of cells with formulas to produce the desired result: a stream with all the attributes of `Trades` but with tuples having a price greater than `200`.

Bargain computation. Notice that the output of a query can still be a single feed: a projection, for example, can be used to view a single attribute of a stream. In cell E3 of Fig. 3, the user has added the `Quotes` input stream, using a query that only shows the price attribute:

```
project(Quotes, price = Quotes.price)
```

This query takes the `Quotes` stream and produces a new stream that has a single attribute named `price`. The new stream ticks synchronously with `Quotes`. Finally, the user enters an Excel conditional to determine whether or not the quoted price is a bargain (cell I7 in Fig. 3).

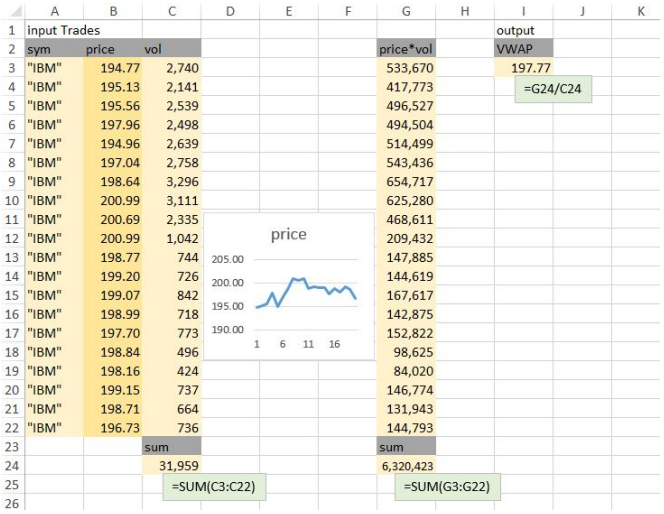


Fig. 5. Computing VWAPs

Exporting data. The user may want to export data back to the server. This can be accomplished by selecting the quoted price and whether or not it is a bargain (cells I3 and I6), and pressing the flash button. ActiveSheets will prompt for a name for this new stream (e.g., `Bargains`), and will start sending this data to the server. The tick of the new stream is the union of the ticks of the feeds that comprise it: i.e., whenever one of the feeds is updated, a new tuple with all the data is sent to the server. Other ActiveSheets users will then be able to subscribe to it. Since the data is computed in the spreadsheet, when the spreadsheet is closed, the stream will no longer be published to the server.

Exporting computation. When the user is ready to deploy the application, he or she can export the computation by pressing the movie button. This feature takes a snapshot of all formulas in the entire spreadsheet and sends it to the server. Each spreadsheet has a single output stream (visible to other users). During export, the user selects the cells that comprise attributes of the output stream. Multiple exports result in separate snapshots on the server. Once computation is exported, it runs at the server side, and exists even after the user closes the spreadsheet. There is a trade-off between data and computation export. In data export, the user may compute new data locally using custom macros and libraries, but the computation disappears when the spreadsheet is closed. In computation export, only a subset of Excel built-in features are supported (at the server), but the computation persists beyond the life of the spreadsheet.

Working with state. In this example, the user wants to keep count of the number of quotes that are bargains. Fig. 6 illustrates how this works. Cell I11 is set to 1 if there is a bargain, and 0 otherwise. Cell I14 is set to the old bargain count

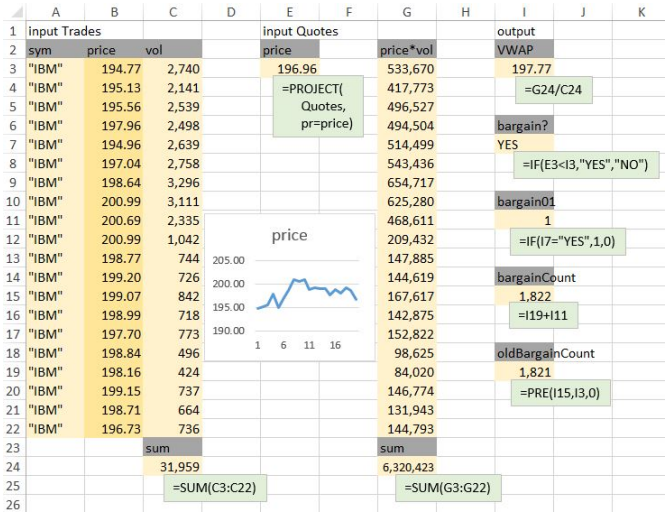


Fig. 6. Stateful computation using PRE

plus cell I11, so it increments iff there is a bargain. And Cell I19 obtains the old bargain count by using the PRE function. Function $\mathsf{PRE}(v, t, v_0)$ is formalized later in this paper; intuitively, it obtains the previous value of v, using the tick of t, and using value v_0 as the default when v is not yet defined. Note that the bargain count computation is cyclic (the new count depends on the old count and vice versa). As we shall see, this is well defined as long as every cycle contains a call to PRE.

Discussion. Fig. 7 shows the VWAP calculation in IBM's Streams Processing Language (SPL) [15]. The `Aggregate` operator invocation in Lines 6-10 consumes stream `Trades` and produces stream `PreVwaps`. Just like the ActiveSheets version, it uses a window of 20 tuples that slide at granularity 1. It sets attribute `priceVol` to $\sum_i P_i \times V_i$ and attribute `vol` to $\sum_i V_i$. The `Functor` operator invocation in Lines 11-13 consumes stream `PreVwaps` and produces stream `Vwaps`. It sets attribute `vwap` to `priceVol / vol`. Whereas ActiveSheets users always have concrete data to look at, developing code in a streaming language like SPL feels more decoupled from the data. Furthermore, writing code in a language like SPL requires familiarity with programming, which is arguably beyond the reach of an end-user.

Compared to the code in Fig. 7, the ActiveSheets experience makes computing with streams accessible to the end-user. It provides a reactive programming model with a fixed control structure: new tuples cause dependent cells to be recomputed and refreshed. The user is freed to focus on the data and its transformations without having to think about unfamiliar programming language syntax. The interface makes it easy to express computations on a window of data from the same stream, and allows computation export for deployment.

```
1.  type
2.    Trade   = tuple<rstring sym, timestamp ts, float64 price, float64 vol>;
3.    PreVwap = tuple<rstring sym, float64 priceVol, float64 vol>;
4.    Vwap    = tuple<rstring sym, float64 vwap>;
5.  graph
6.    stream<PreVwap> PreVwaps = Aggregate(Trades) {
7.      window Trades:   sliding, count(20), count(1);
8.      output PreVwaps: priceVol = Sum(price * vol),
9.                       vol      = Sum(vol);
10.   }
11.   stream<Vwap> Vwaps = Functor(PreVwaps) {
12.     output Vwaps:    vwap     = priceVol / vol;
13.   }
```

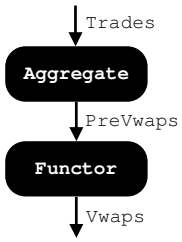


Fig. 7. VWAP in SPL

The spreadsheet also provides a variety of visualization possibilities. In the example, the user can create a line chart for the price as shown in Fig. 3, and the chart is live as well.

3 Spreadsheet Calculus

This section formalizes a core calculus to support our programming model. It first specifies the constructs and semantics of a minimal *client* spreadsheet—a collection of cells and formulas—connected to a *server* providing real-time data feeds. The constructs let us compute over recent feed histories and build stateful spreadsheets. The semantics define when and how cell values are computed. We prove that the resulting executions are well-defined, reactive, and deterministic provided the client spreadsheet is free from immediate cyclic dependencies (Section 3.1).

Clients compute over potentially infinite data feeds. Our programming model is intended to favor real-time analytics and prevent users from engaging into expensive querying of feed histories. A client for example can compute the average of a data feed over time (since the beginning of time), but it must do so incrementally as the live data flows through the client. We formally establish that executions in our model can be computed incrementally over time, using a bounded amount of computation per update (i.e., incoming data packet) and a bounded amount of memory to keep track of the execution state—the "past"—of size proportional to the client itself (Section 3.2).

The end-user can change formulas in the spreadsheet while real-time feeds are being processed. To support this form of live programming, we extend our semantics so that cells no longer contain static formulas, but feeds of formulas that change over time (Section 3.3). Our core calculus is not intended as an actual programming interface for the end-user. To bridge this gap, we specify a stream calculus by reduction to our core calculus. It supports richer notions of data streams—sequences of tuples with named attributes—and formulas (Section 3.4). Finally, we specify a query language that provides familiar relational operators on data streams such as projection and selection (Section 3.5).

3.1 Core Calculus

We start with the definitions, then establish key properties of our core calculus.

Ticks. Let a *tick* T be a possibly empty, at most countable, strictly increasing series of non-negative real numbers $\{t_0, t_1, t_2, \cdot\cdot\}$ representing a sequence of arrival times. We require that T is unbounded if infinite.

We write $T \triangleright t$ for the tick T up to time t that is formally the series $T \cap [0, t]$, which is always a finite tick. A non-empty finite tick T always admits a *maximal element* $\max(T)$. Given a finite tick T with at least two elements, we define the *second-to-max element* $\text{prev}(T)$ as $\max(T \setminus \max(T))$. We write $(t_0, t_1) \in T$ if t_0 and t_1 are two *consecutive arrival times* in T, that is, if $t_1 \in T$ and $t_0 = \text{prev}(T \triangleright t_1)$.

Feeds. Let a *feed* ϕ be a map from a tick to values. We write $\text{dom}(\phi)$ for the tick of ϕ. We say that ϕ *ticks* at time t iff $t \in \text{dom}(\phi)$. As a convenience, we overload the notation $\phi(t)$ as follows. If $t \in \text{dom}(\phi)$, then $\phi(t)$ is the usual function application. Otherwise, if $\text{dom}(\phi) \triangleright t \neq \emptyset$, then $\phi(t)$ is defined as $\phi(\max(\text{dom}(\phi) \triangleright t))$. Otherwise, $\phi(t)$ is undefined and we write $\phi(t) = \perp$ using $\perp$ to denote the absence of a value. In short, $\phi(t)$ is always the most recent value of ϕ at time t.

Servers. Let a *server* S be a finite collection of feeds. We define the *server tick* N of S as the tick $\bigcup_{\phi \in S} \text{dom}(\phi)$. Because of the required properties of ticks, it makes sense to think of N as $\mathbb{N}$ or a subset of $\mathbb{N}$ if it helps the reader. While ticks are intended to model real-time arrival times, our semantics really think of arrival times as logical instants. The order matters, but the time difference between two instants does not.

Clients, cells, and formulas. Let a *client* C be a finite collection of *cell*s. Each cell has a unique name c and contains a formula f. We write $c \equiv f$ iff c contains formula f. The syntax of formulas is defined as follows, where f denotes a formula, c a cell name, ϕ a server feed, and *op* a family of operators on values (such as division /, greater-than >, or Excel's IF function).

$$f ::= \phi \mid op(c_1, \cdot\cdot, c_n) \mid c_0 @ c_1 \mid \textbf{latch}(c_0, c_1)$$

For simplicity, our core calculus does not permit nesting constructs. The stream calculus of Section 3.4 lifts this restriction.

We do not explicitly model constant formulas as these can be obtained by means of constant server feeds. Observe that our semantics will distinguish constant feeds with the same value but distinct ticks.

Our core calculus is untyped. We assume all *op* operators are total functions. For simplicity, we do not consider "eager" operators capable of producing values even if not all operands are defined, but such operators could be added easily.

The calculus has two constructs to manipulate time: @ and **latch**. The @ construct makes it possible to sample a feed according to a Boolean condition (a feed with Boolean values): $c_0@c_1$ ticks when c_1 does and evaluates to *true*, returning the current value of c_0. The **latch** construct provides a general mechanism to delay a feed so that a feed value that is not the most recent can be accessed: $\mathbf{latch}(c_0, c_1)$ ticks when c_1 does returning the value of c_0 at the previous tick of c_1. We illustrate the two constructs below as we specify their semantics. In Section 3.4, we show how PRE can be defined using **latch**.

Well-formedness. We define the set of *immediate dependencies deps*(c) *of a cell* c as follows.

$$\text{deps}(c) = \begin{cases} \emptyset & \text{if } c \equiv \phi \\ \{c_1, \cdot\cdot, c_n\} & \text{if } c \equiv op(c_1, \cdot\cdot, c_n) \\ \{c_0, c_1\} & \text{if } c \equiv c_0@c_1 \\ \{c_1\} & \text{if } c \equiv \mathbf{latch}(c_0, c_1) \end{cases}$$

In essence, our semantics are such that if $c \equiv \mathbf{latch}(c_0, c_1)$ then c only depends on the past of c_0, hence c does not immediately depend on c_0. Reciprocally, if c immediately depends on c_0 then the semantics of c at time t will potentially be derived from the semantics of c_0 at time t. We therefore need immediate dependencies to be acyclic. We say that a client is *well-formed* iff the directed graph $\mathcal{G}$ of immediate dependencies is acyclic, where the vertices of $\mathcal{G}$ are the cell names and there exists an edge (c, c') in $\mathcal{G}$ iff $c' \in \text{deps}(c)$. If a client is not well-formed, we can identify a cycle and notify the user.

Semantics. We now specify the semantics of well-formed clients by recursion. Lemma 1 will establish that this recursion is well-founded.

We define by mutual recursion the tick $\mathcal{T}(c)$ of a cell c of a well-formed client C and the value $\mathcal{E}(c, t)$ of c at time $t \in [0, \infty)$ as follows, starting with $\mathcal{T}(c)$.

$$\mathcal{T}(c) = \begin{cases} \text{dom}(\phi) & \text{if } c \equiv \phi \\ \{t \in \bigcup_{i=1}^{n} \mathcal{T}(c_i) | \forall i \in \{1, \cdot\cdot, n\} : \mathcal{T}(c_i) \triangleright t \neq \emptyset\} & \text{if } c \equiv op(c_1, \cdot\cdot, c_n) \\ \{t \in \mathcal{T}(c_1) | \mathcal{E}(c_1, t) = true, \mathcal{T}(c_0) \triangleright t \neq \emptyset\} & \text{if } c \equiv c_0@c_1 \\ \mathcal{T}(c_1) & \text{if } c \equiv \mathbf{latch}(c_0, c_1) \end{cases}$$

In contrast with typical synchronous programming models [8], our core calculus does not require the operands of an operator *op* to be synchronous (share the same tick). Instead, an operator *op* ticks each time an operand does (once all operands are defined). Once c_0 is defined, $c_0@c_1$ ticks when c_1 does and evaluates to *true*. The tick of $\mathbf{latch}(c_0, c_1)$ is simply the tick of the second argument c_1.

We now consider the definition of $\mathcal{E}(c,t)$.

$$\mathcal{E}(c,t) = \begin{cases} \phi(t) & \text{if } c \equiv \phi \\ op(\mathcal{E}(c_1,t),\cdot\cdot,\mathcal{E}(c_n,t)) & \text{if } c \equiv op(c_1,\cdot\cdot,c_n) \text{ and } \forall i : \mathcal{E}(c_i,t) \neq \bot \\ \mathcal{E}(c_0, \max(\mathcal{T}(c) \triangleright t)) & \text{if } c \equiv c_0@c_1 \text{ and } \mathcal{T}(c) \triangleright t \neq \emptyset \\ \mathcal{E}(c_0, \text{prev}(\mathcal{T}(c_1) \triangleright t)) & \text{if } c \equiv \textbf{latch}(c_0,c_1) \text{ and } |\mathcal{T}(c_1) \triangleright t| \geq 2 \\ & \text{and } \mathcal{T}(c_0) \triangleright \text{prev}(\mathcal{T}(c_1) \triangleright t) \neq \emptyset \\ \mathcal{E}(c_1,t) & \text{if } c \equiv \textbf{latch}(c_0,c_1) \text{ and } |\mathcal{T}(c_1) \triangleright t| \geq 2 \\ & \text{and } \mathcal{T}(c_0) \triangleright \text{prev}(\mathcal{T}(c_1) \triangleright t) = \emptyset \\ \mathcal{E}(c_1,t) & \text{if } c \equiv \textbf{latch}(c_0,c_1) \text{ and } |\mathcal{T}(c_1) \triangleright t| = 1 \\ \bot & \text{otherwise} \end{cases}$$

The semantics of operators lifts an operator from values to feeds by simply invoking the operator on the most recent value of each feed.

The formula $c_0@c_1$ samples the value of c_0 when it ticks. For instance, if $a \equiv nat$ and $b \equiv isEven(a)$ and $c = a@b$ where *nat* is a server feed producing the natural integers and *isEven* a unary operator with the obvious semantics, then c only produces even integers. The arrival time of each integer in c is the same as the arrival time of the same integer in *nat*.

The formula $\textbf{latch}(c_0,c_1)$ provides for each tick of c_1 the value of c_0 recorded at the previous tick of c_1. But it defaults to the value of c_1 instead, if either this is the first tick of c_1 or c_0 was not yet defined when c_1 last ticked.

The **latch** construct serves a double purpose: it makes stateful clients possible and it enables clients to reason about windows of data. For an example of a stateful computation, suppose *zero* is the unary constant operator with value 0, *add* is the binary addition, and 1 is a sever feed with tick $\{0\}$ and value 1. The cell d in client $\{a \equiv feed, b \equiv zero(a), c \equiv 1, d \equiv add(c,e), e \equiv \textbf{latch}(d,b)\}$ counts the number of ticks in the server feed *feed*. Observe that b hence e and d tick exactly when *feed* does. Moreover, the initial value of d is 1 and each subsequent value of d is obtained by incrementing the previous value of d by one. For a window example, suppose *neq* is a binary inequality test operator. The cell c in client $\{a \equiv feed, b \equiv \textbf{latch}(a,a), c \equiv neq(a,b)\}$ ticks when the server feed *feed* does and evaluates to *true* iff the current value of *feed* is different from the previous value. In general, windows into feed histories can be obtained by chaining latches, e.g., $\{a \equiv feed, b \equiv \textbf{latch}(a,a), c \equiv \textbf{latch}(b,b), d \equiv \textbf{latch}(c,c)\}$. Cell a provides the current value of *feed*, b the previous value, c the value before that, etc.

Observe that in the stateful example, the **latch** is used to form a cycle of cells, whereas in the window examples, there is no such cycle. In the latter, the two arguments to **latch** can be the same. But well-formedness forbids cyclic uses of **latch** (via its second argument) as in the ill-formed client $\{a \equiv \textbf{latch}(a,a)\}$.

We now prove that the recursive definition of $\mathcal{T}$ and $\mathcal{E}$ is well-founded for well-formed clients. In the sequel, we require all clients to be well-formed.

Lemma 1 (Soundness). *For a cell c of a well-formed client C and a time t, the value $\mathcal{E}(c,t)$ of c at time t and the tick $\mathcal{T}(c) \triangleright t$ of c up to time t are defined via a well-founded recursion.*

Proof. Let depth(c) be the length of the longest path in $\mathcal{G}$ with source c. For a cell $c \in C$ and time $t \in [0, \infty)$ we define $\sigma(c,t) \in N \times \mathbb{N}$ as $(\max(N \triangleright t), \text{depth}(c))$. The lexicographic order $\leq$ of $N \times \mathbb{N}$ is well-founded, since C is well-formed.

We can rewrite the definition of $\mathcal{T}(c)$ as a definition of $\mathcal{T}(c) \triangleright t$ so that every tick instance of the right-hand side is only needed up to time t. In the definition of $\mathcal{E}(c_0@c_1, t)$, we can expand $\mathcal{T}(c) \triangleright t$ into its definition. We now establish that the recursive co-definition of $\mathcal{T}(c) \triangleright t$ and $\mathcal{E}(c,t)$ is well-founded using $(\sigma, \leq)$ to order the tuples $(c,t) \in C \times [0,\infty)$.

In all induction cases except for the definition of $\mathcal{E}(\mathbf{latch}(c_0, c_1), t)$, the terms of the right-hand side are only concerned with time up to t and cells of strictly lower depth. Moreover, the tick up to t and value at t of the cell c with formula $c \equiv \mathbf{latch}(c_0, c_1)$ are defined using $\mathcal{T}(c_1) \triangleright t$ and $\mathcal{E}(c_1, t)$ (same time, strictly lower depth) and possibly $\mathcal{T}(c_0) \triangleright t_0$ and $\mathcal{E}(c_0, t_0)$ with $t_0 = \text{prev}(\mathcal{T}(c_1) \triangleright t)$ such that $\max(N \triangleright t_0) < \max(N \triangleright t)$. □

Our calculus is therefore *deterministic*: the tick and values of a cell of a well-formed client are unambiguously defined at all times. Our calculus is also *reactive* in the sense that everything happens in *reaction* to the ticks of the server feeds.

Lemma 2 (Reactivity). *The tick of a cell c of a well-formed client C is a subset of the server tick N. The value of c at a time t is equal to the value of c at the most recent arrival time of c if any or undefined if none.*

Proof. The tick of $c_0@c_1$ is a subset of $\mathcal{T}(c_1)$. The tick of $op(c_1, \cdot\cdot, c_n)$ is a subset of $\bigcup_{i=1}^{n} \mathcal{T}(c_i)$. By induction over the depth of the cell. □

3.2 Boundedness

Because of **latch**, the values of the cell at time t are defined using past values of cells and feeds. But a careful look at the definitions shows that the dependency on past values is bounded. Concretely, $c \equiv \mathbf{latch}(c_0, c_1)$ only needs to retain one value of c_0 at a time (in addition to the current value of c). Formally, for all $c \in C$ and $t \in N$ we define:

$$\mathcal{H}(c,t) = \begin{cases} \mathcal{E}(c_0, \max(\mathcal{T}(c_1) \triangleright t)) & \text{if } c \equiv \mathbf{latch}(c_0, c_1) \text{ and } |\mathcal{T}(c_1) \triangleright t| > 0 \\ \bot & \text{otherwise} \end{cases}$$

Lemma 3 (Boundedness). *For all $(t_0, t) \in N$, the values of $\mathcal{H}$ and $\mathcal{E}$ at time t for each $c \in C$ can be computed as a function of $\mathcal{H}$ and $\mathcal{E}$ at time t_0 and the ticks and values of the server feeds at time t.*

Proof. We observe that we can rewrite the semantics of the core calculus as follows.

$$t \in \mathcal{T}(c) \Leftrightarrow \begin{cases} t \in \text{dom}(\phi) & \text{if } c \equiv \phi \\ (\exists i : t \in \mathcal{T}(c_i)) \wedge (\forall i : \mathcal{E}(c_i, t) \neq \bot) & \text{if } c \equiv op(c_1, \cdot\cdot, c_n) \\ t \in \mathcal{T}(c_1) \wedge \mathcal{E}(c_1, t) = \mathit{true} \wedge \mathcal{E}(c_0, t) \neq \bot & \text{if } c \equiv c_0@c_1 \\ t \in \mathcal{T}(c_1) & \text{if } c \equiv \mathbf{latch}(c_0, c_1) \end{cases}$$

If $t \notin \mathcal{T}(c)$ then $\mathcal{H}(c,t) = \mathcal{H}(c,t_0)$ and $\mathcal{E}(c,t) = \mathcal{E}(c,t_0)$ by Lemma 2. Otherwise, $\mathcal{H}(c,t) = \mathcal{E}(c_0,t)$ if $c \equiv \mathbf{latch}(c_0,c_1)$ or $\perp$ if not, and

$$\mathcal{E}(c,t) = \begin{cases} \phi(t) & \text{if } c \equiv \phi \\ op(\mathcal{E}(c_1,t),\cdot\cdot,\mathcal{E}(c_n,t)) & \text{if } c \equiv op(c_1,\cdot\cdot,c_n) \text{ and } \forall i : \mathcal{E}(c_i,t) \neq \perp \\ \mathcal{E}(c_0,t) & \text{if } c \equiv c_0@c_1 \\ \mathcal{H}(c,t_0) & \text{if } c \equiv \mathbf{latch}(c_0,c_1) \text{ and } \mathcal{H}(c,t_0) \neq \perp \\ \mathcal{E}(c_1,t) & \text{if } c \equiv \mathbf{latch}(c_0,c_1) \text{ and } \mathcal{H}(c,t_0) = \perp \\ \perp & \text{otherwise} \end{cases}$$

By induction using the well-foundedness argument of Lemma 1, the two semantics define the same tick and values for all cells at all times. □

In summary, storing one value for each occurrence of **latch** enables the incremental computation of these semantics over time. In particular, the memory required is bounded by the client size. Moreover, the amount of computation per tick is also bounded by the client size (assuming unit cost for the operators *op*).

3.3 Live Calculus

We now define the semantics of live clients where we permit formulas to evolve over time. We suppose that each cell $c \in C$ has a *feed of formulas* $\hat{c}$ with tick $\mathrm{dom}(\hat{c})$ and formula $\hat{c}(t)$ at time t. While we do not model cell creation or deletion explicitly, we permit cells to be initially empty. The formula feeds model external changes to formulas (e.g., user input). We do not consider "higher-order" spreadsheets where formulas could be computed by the spreadsheet itself.

We define the *immediate dependencies of cell c at time t* as follows.

$$\mathrm{deps}(c,t) = \begin{cases} \emptyset & \text{if } \hat{c}(t) = \phi \\ \{c_1,\cdot\cdot,c_n\} & \text{if } \hat{c}(t) = op(c_1,\cdot\cdot,c_n) \\ \{c_0,c_1\} & \text{if } \hat{c}(t) = c_0@c_1 \\ \{c_1\} & \text{if } \hat{c}(t) = \mathbf{latch}(c_0,c_1) \end{cases}$$

We say a client is well-formed iff the graph of immediate cell dependencies is acyclic at all times. We define the tick of cell c, $\mathcal{T}(c)$, by concatenating the ticks of its successive formulas over time. We first define the *tick of a cell c around time t* as follows.

$$\mathcal{T}[t](c) = \begin{cases} \mathrm{dom}(\phi) & \text{if } \hat{c}(t) = \phi \\ \{t \in \bigcup_{i=1}^{n} \mathcal{T}(c_i) | \forall i \in \{1,\cdot\cdot,n\} : \mathcal{T}(c_i) \triangleright t \neq \emptyset\} & \text{if } \hat{c}(t) = op(c_1,\cdot\cdot,c_n) \\ \{t \in \mathcal{T}(c_1) | \mathcal{E}(c_1,t) = \mathit{true}, \mathcal{T}(c_0) \triangleright t \neq \emptyset\} & \text{if } \hat{c}(t) = c_0@c_1 \\ \mathcal{T}(c_1) & \text{if } \hat{c}(t) = \mathbf{latch}(c_0,c_1) \end{cases}$$

$$\mathcal{T}(c) = \mathrm{dom}(\hat{c}) \cup \left(\bigcup_{(t_0,t_1) \in \mathrm{dom}(\hat{c})} \mathcal{T}[t_0](c) \cap [t_0,t_1) \right) \cup \left(\bigcup_{t = \max(\mathrm{dom}(\hat{c}))} \mathcal{T}[t](c) \cap [t,\infty) \right)$$

By convention, a cell also ticks when its formula feed does. The last term in this union handles the case of a finite formula feed.

We define the value of cell c at time t using the current formula $\hat{c}(t)$ as in the core calculus except for the **latch** construct. Let $t_0 = \max(\mathrm{dom}(\hat{c}) \triangleright t)$ be the most recent arrival of the formula feed $\hat{c}$ if defined. If $\hat{c}(t) = \mathbf{latch}(c_0, c_1)$ then

$$\mathcal{E}(c,t) = \begin{cases} \mathcal{E}(c_0, \mathrm{prev}(\mathcal{T}(c) \triangleright t)) & \text{if } \mathcal{T}(c) \triangleright t > t_0 \text{ and } \mathcal{T}(c_0) \triangleright \mathrm{prev}(\mathcal{T}(c) \triangleright t) \neq \emptyset \\ \mathcal{E}(c_1, t) & \text{if } \mathcal{T}(c) \triangleright t > t_0 \text{ and } \mathcal{T}(c_0) \triangleright \mathrm{prev}(\mathcal{T}(c) \triangleright t) = \emptyset \\ \mathcal{E}(c_1, t) & \text{if } \mathcal{T}(c) \triangleright t = t_0 \\ \bot & \text{otherwise} \end{cases}$$

Intuitively, a **latch** does not access values that predate the formula that contains the **latch**. This ensures that these semantics are still incrementally computable without the need for an "oracle" to predict future **latch** occurrences.

3.4 Stream Calculus

Our core calculus from Section 3.1 is not intended as an actual programming interface for the end-user. This section introduces a stream calculus that enriches the core calculus with higher-level notions of streams and formulas. A stream is a sequence of tuples with named attributes. The stream calculus permits nesting constructs in formulas, handles constant values, and formalizes PRE.

To simplify the presentation, we return to the fixed formulas of the core calculus, but the techniques for live editing in Section 3.3 remain applicable.

Streams. We say that two feeds are *synchronous* if they have the same tick. We define a *stream* s to be a non-empty, finite collection of synchronous feeds. The feeds in a stream are labeled with *attributes*. Given a stream s, we write $s.a$ to denote the feed of s labeled a. We write $\mathcal{A}(s)$ for the set of attributes of s.

Semantics. We now define a calculus over streams by reduction to the core calculus of Section 3.1. The syntax of formulas is as follows, where v stands for a constant value:

$$f \ ::= \ v \mid s.a \mid c \mid op(f_1, \cdot\cdot, f_n) \mid f_0 @ f_1 \mid \mathbf{latch}(f_0, f_1) \mid \mathsf{PRE}(f_0, f_1, v) \mid \mathsf{PRE}(f, v)$$

Constructs can be nested. Formulas v and $s.a$ are server feeds ϕ. Formula v denotes a feed with value v and tick $\{0\}$. Formula $\mathsf{PRE}(f_0, f_1, v)$ is a syntactic shortcut for $\mathbf{latch}(f_0, \mathit{first}(v, f_1))$ where the *first* operator maps (x, y) to x. Therefore, $\mathit{first}(v, f_1)$ produces a constant feed of values v with tick $\mathcal{T}(f_1)$. Formula $\mathsf{PRE}(f, v)$ is a shorthand for $\mathsf{PRE}(c, c, v)$ where c is a fresh cell with formula f. The binary form of PRE is the most intuitive one: $\mathsf{PRE}(f, v)$ ticks when f does, evaluates to v initially then to the previous value of f. This form cannot express cyclic computations such as accumulators. The tick of cell c in client $\{c = \mathit{add}(\mathsf{PRE}(c, 0), 1)\}$ cannot be defined by recursion.[2] The ternary form of

[2] Least-fixed-point approaches would not work either as our calculus supports substraction by means of the @ construct.

PRE therefore permits the independent specification of the formula f_0 to latch, the tick f_1 of the latch, and the initial value v of the latch. It is less expressive than the core **latch** construct—it restricts its second argument to a constant feed—but easier for the user to reason about.

Let C be a client in the stream calculus. We define the semantics of C by constructing a client C' in the core calculus. In particular, we specify that C is well-formed iff C' is. The semantics of a cell c in C is specified as the semantics of the cell c in C', that is, the cell with the same name in the reduced client.

Intuitively, the reduced client is simply defined by introducing helper cells for every subformula and replacing subformulas with references to these helper cells. Concretely, we specify by induction over the structure of formulas, a reduction $\mathcal{R}$ that maps a cell c with formula f in the stream calculus to a fragment of a client in the core calculus, that is, one or more cells with their respective formulas in the core calculus. All cells but c itself in each map are *fresh*, i.e., have a globally unique name.

The reduced client C' of C is then simply the union of these fragments for each cell c in C.

$$\mathcal{R}(c,f) = \begin{cases} \{c \equiv f\} & \text{if } f \in \{v, s.a, c_0\} \\ \{c \equiv op(c_1,\cdot\cdot,c_n)\} \cup \bigcup_{i=1}^{n} \mathcal{R}(c_i,f_i) & \text{if } f = op(f_1,\cdot\cdot,f_n) \\ \{c \equiv c_0@c_1\} \cup \mathcal{R}(c_0,f_0) \cup \mathcal{R}(c_1,f_1) & \text{if } f = f_0@f_1 \\ \{c \equiv \mathbf{latch}(c_0,c_1)\} \cup \mathcal{R}(c_0,f_0) \cup \mathcal{R}(c_1,f_1) & \text{if } f = \mathbf{latch}(f_0,f_1) \\ \{c \equiv c_0\} \cup \mathcal{R}(c_0,\mathbf{latch}(f_0,\mathit{first}(v,f_1))) & \text{if } f = \mathsf{PRE}(f_0,f_1,v) \\ \{c \equiv c_0\} \cup \mathcal{R}(c_0,\mathsf{PRE}(c_1,c_1,v)) \cup \mathcal{R}(c_1,f) & \text{if } f = \mathsf{PRE}(f,v) \end{cases}$$

3.5 Query Language

The stream calculus assumes a programming model where the user modifies one cell at a time, defining one value feed at a time. In contrast, ActiveSheets' query language allows the user to enter formulas in a range of cells at once by defining a stream with multiple attributes and a window over this stream history, all in a single step. Moreover, this query language provides higher-level mechanisms to process streams inspired from relational operators—emphasizing relations and deemphasizing arrival times.

In this section, we specify a basic query language over streams, and show how it reduces to the stream calculus. It consists of projection and selection operators. Our implemented query language supports other traditional relational operators such as sort, pivot, aggregate, and deduplicate. Excel has native features that support static version of some of these constructs (sort, pivot), and our query language complements these features with streaming ones.

The query language is tightly integrated with the UI. In particular, the number of rows in the target range of a query defines the length of the stream history to preserve. We do not model this coupling here.

Queries. The syntax of queries is defined as follows where q denotes a query, s a stream, a an attribute, and f a formula in the stream calculus.

$$\begin{aligned} q_s ::= {} & s' \\ & \mid \mathsf{PROJECT}(q_s, a_1 = f_1, \cdot\cdot, a_n = f_n) \\ & \mid \mathsf{SELECT}(q_s, f) \end{aligned}$$

A query q_s defines a new *client stream* named s. We require that the names of the streams (client and server) are pairwise distinct. The PROJECT construct defines a new stream with attributes a_1 through a_n, with formulas f_1 through f_n, respectively. In essence, the PROJECT construct allows the user to synchronize a collection of feeds to produce a stream: the values of f_1 through f_n are sampled according to the tick of the first parameter of PROJECT, and assigned to the attributes of the resulting stream. The SELECT construct defines a new stream with all the attributes of its first parameter, but with tuples that have been filtered according to the Boolean formula f.

Semantics. A client (C, Q) in the query language combines a client C in the stream calculus—a finite collection of cells and formulas—and a finite collection of queries Q. We denote by W (Y) the set of all the client streams (server streams, respectively).

The attributes $\mathcal{A}(s)$ of s in W are defined as follows:

$$\mathcal{A}(s) = \begin{cases} \mathcal{A}(s') & \text{if } q_s = s' \text{ or } q_s = \mathsf{SELECT}(q'_{s'}, f) \\ \{a_1, \cdot\cdot, a_n\} & \text{if } q_s = \mathsf{PROJECT}(q'_{s'}, a_1 = f_1, \cdot\cdot, a_n = f_n) \end{cases}$$

We map each attribute a of each client stream s to a fresh cell in C' denoted c^a_s. We define by induction on the structure of queries a reduction from a query q_s to a collection of cells $\mathcal{C}(s)$ in the stream calculus as follows.

$$\mathcal{C}(s) = \begin{cases} \bigcup_{a \in \mathcal{A}(s)} \{c^a_s \equiv s'.a\} & \text{if } q_s = s' \text{ and } s' \in Y \\ \bigcup_{a \in \mathcal{A}(s)} \{c^a_s \equiv c^a_{s'}\} & \text{if } q_s = s' \text{ and } s' \in W \\ \mathcal{C}(s') \cup \bigcup_{i=1}^{n} \{c^{a_i}_s \equiv nth(i, f_1, \cdot\cdot, f_n)@true(c^{a'}_{s'})\} & \\ \qquad \text{if } q_s = \mathsf{PROJECT}(q'_{s'}, a_1 = f_1, \cdot\cdot, a_n = f_n) \text{ and } a' \in \mathcal{A}(s') & \\ \mathcal{C}(s') \cup \bigcup_{a \in \mathcal{A}(s)} \{c^a_s \equiv c^a_{s'}@(f@true(c^a_{s'}))\} & \text{if } q_s = \mathsf{SELECT}(q'_{s'}, f) \end{cases}$$

For the PROJECT construct, we resample each f_i using the tick of $q'_{s'}$, which we can obtain by applying the constant unary *true* operator to one of its attributes a'. But we need to make sure that all f_i are defined before we emit a value for any attribute. We therefore combine all the f_i together using operator $nth : (i, a_1, \cdot\cdot, a_n) \mapsto a_i$. Like any operator lifted to feeds, it only starts ticking once all arguments are defined.

For the SELECT construct, we first sample the Boolean condition f according to the tick of the target stream s' using the rightmost @ construct, then apply the resulting filter to the stream s' using a second @ construct. This ensures that the output stream is synchronous with the input stream.

The reduced client C' is obtained as $C \cup \bigcup_{s \in W} \mathcal{C}(s)$. We specify that (C, Q) is well formed iff C' is.

4 Implementation

ActiveSheets is implemented as a client-server architecture. The client is a thin layer that implements minimal functionality by design so that it may be easily repurposed for integration with multiple spreadsheet front-ends. In the current implementation, the client is integrated with Microsoft Excel. It interacts with the server via a RESTful interface [12] that provides an API to discover available streams, subscribe to streams and create feeds, as well as to export data and computation. Fig. 8 sketches the overall system architecture.

4.1 Client Side

The client consists of two components. The first is the client proxy. It encapsulates front-end independent functionality including a session manager and a real-time data service that continuously updates the cells in the spreadsheet when ticks advance. The second is the front-end user interface and integration with the spreadsheet application (e.g., Microsoft Excel). The client UI and some of its features were described earlier in Section 2.

In our current implementation, the client proxy is written in C# and the UI front-end consists of a collection of Visual Basic macros. The client proxy implements the Real-Time Data Server interface (IRtdServer) to communicate with Excel. It makes it possible for the client proxy to notify Excel that new data is available and for Excel to asynchronously pull the data from the client proxy. The client proxy therefore acts as a buffer between the ActiveSheets server and Excel. The client proxy runs as a dynamic-link library (DLL) plugin inside Excel.

4.2 Server Side

The server side consists of the server proxy and a stream processing engine. The former implements the primary functionality while the latter is used to deploy generated stream processors when the client exports computation to the server. The server side proxy is comprised of a (1) name manager, (2) query processor, and (3) spreadsheet compiler.

The name manager maintains a directory of client connections and dispatches client requests to dedicated handlers. When a query is received that subscribes a client to a particular stream, the name manager allocates a dedicated handler to service the request. The handler persists as long as the client connection is maintained. The current implementation is written in Java and based on Akka, an actor-based system for highly concurrent and event-driven applications [1]. It is conceptually a message-driven runtime, where actors execute when messages are received, producing new messages that are consumed by subsequent actors or pushed to the client. Actors in ActiveSheets input tuples from existing streams, parse and reformat the tuples if necessary, and output the resulting tuples as new messages that are dispatched to registered listeners (e.g., clients). Data that is exported from the spreadsheet is handled by the name manager.

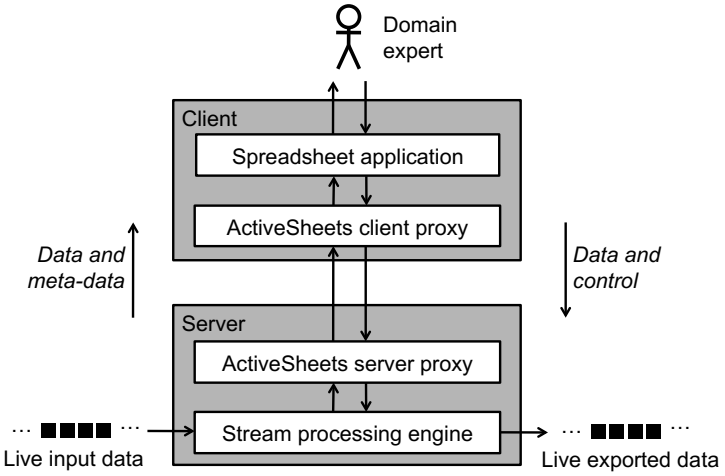


Fig. 8. ActiveSheets System Architectures

The query processor is an actor that applies a given set of transformations to a sequence of input tuples. The query is received from the client as a string, parsed on the server, and interpreted accordingly. All of the query operators described in Section 3.5 are supported. The operators are applied sequentially in the order implied by the programmer, although we believe the order of application is amenable to optimizations since some operators are commutative and may reduce the amount of computation applied to any given tuple.

The spreadsheet compiler is responsible for handling exported computation. It parses the spreadsheets and builds a dependence graph between the cells, which in turn is used to derive a computational circuit for the spreadsheet. Terminal cells which have no incoming edges or outgoing edges in the dependence graph are input and output signals, respectively. Internal cells contain formulas that correspond to gates in the circuit, with input wires flowing from and output wires flowing to other cells as in the dependence graph. One circuit is created for each exported spreadsheet, and it is encapsulated within a single actor that will update the output signals as new ticks arrive. Output signals are visible to other users as new streams. The computation on the server persists even if the spreadsheet is no longer running.

5 Case Studies

The goal of this section is to convey a feeling for what kind of streaming computations are natural to implement in a spreadsheet. The examples are drawn from a variety of domains (commerce, transportation, infrastructure, and security), and illustrate how the features of ActiveSheets play out in practice. Fig. 9 shows an Excel spreadsheet for the examples.

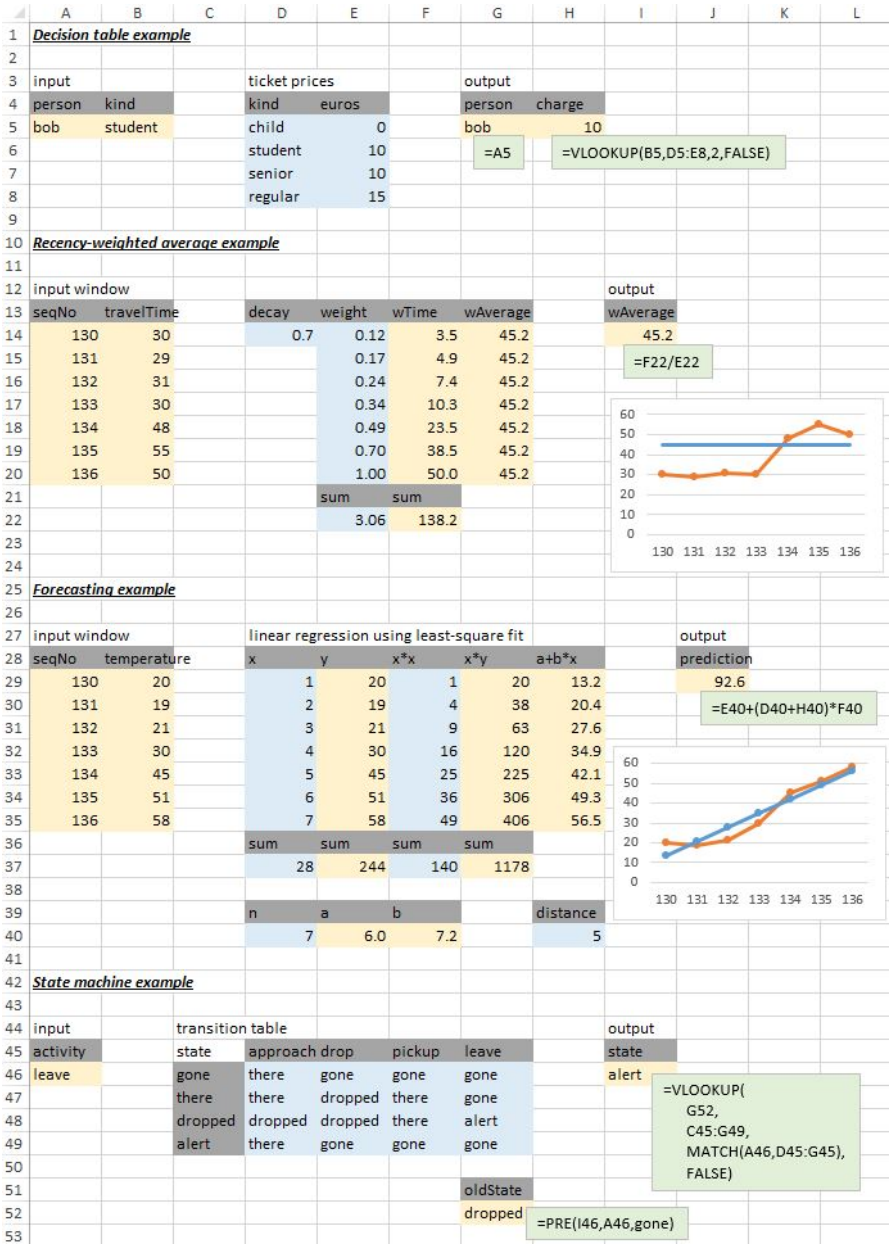


Fig. 9. Case studies. Yellow background indicates live streaming data, blue background indicates constants, and green rectangles show formulas.

Decision Table. Consider a commerce application where the input is a stream of persons (with name and age category), and the output is a stream of ticket prices. The ticket prices are obtained by looking them up in a table indexed by the age category (child, student, senior, or regular). Such tables are natural to express in spreadsheets, more so than in traditional text-based languages. The example in Fig. 9 looks up the ticket price for Bob, who is a student, and must thus pay 10 Euro. Excel offers VLOOKUP(key: ref, tab: rangeRef, valCol: int, range: bool) for table lookup. One requirement this use case illustrates is that besides single-cell references, we must also support range-references, which refer to a rectangular region comprising multiple rows and columns. The calculus models range references via n-ary functions. The VLOOKUP operator itself does not need to be baked into the calculus, since it is stateless and built into Excel. Variations on the decision-table case study could use relative lookup instead of absolute lookup, for instance, when the age is given as an integer instead of a category.

Recency-Weighted Average. Consider a transportation application where the input is a stream of travel times between two landmarks, and the output is a stream of travel time estimates between the same landmarks. To estimate travel time in current traffic, the most recent input samples should count the most in the estimation. This can be accomplished by weighting the window with a decay curve. In Fig. 9, the most recent travel time is in cell B20, and the cells above it use PRE to get earlier readings. Cell D14 specifies the decay factor with the constant 0.7. In many traditional streaming languages, such as CQL [4] or SPL [15], windows are high-level and opaque, supporting only a fixed set of built-in aggregations such as sum, min, max, or average. However, this use-case requires associative access on window contents. In ActiveSheets, this is natural to do, since the window contents are laid out in a range of cells, offering users full viewing and manipulation power. Variations of this use case could take additional information into account, such as the day of the week.

Forecasting. Consider an infrastructure application where the input is a stream of temperature readings in a data center, and the output is a stream of predictions for future temperature readings based on the current trend. A spreadsheet can implement this by calculating a least-square fit over the recent readings, then extend that curve into the future for forecasts. The example in Fig. 9 extends the temperature trend by a distance of 5 steps into the future, and predicts that it will reach a dangerous 92.6° Celsius. Such forecasting algorithms are not that easy to get right, and a spreadsheet can help with debugging, since the developer can visualize the curve and the prediction interactively. This use case does not pose any additional requirements on the calculus; it suffices to offer associative history access as is the case with recency-weighted average. As a variation, instead of predicting the temperature at a fixed distance in the future, the application could predict how long it would take to reach a fixed threshold value (say, 100° Celsius). This could be used for an evacuation count-down.

State Machine. Consider a security application where the input is a stream of activities at a business location, and the output is a stream of suspicious events that ought to be checked out by authorities. An example of a suspicious event would be when a person enters the business location, drops an object and then leaves the premises without taking back the object. This is easy to specify via a deterministic finite automaton (DFA). A spreadsheet can implement a DFA via a transition table indexed by the previous state and the current activity, to yield the next state and an output. Just like a decision table, a DFA transition table can be naturally represented by a block of cells in a spreadsheet. As cell I46 in Fig. 9 illustrates, the lookup in this case is two-dimensional, using VLOOKUP in combination with MATCH. As far as the calculus is concerned, this use case combines the requirement for a decision table with the requirement for history access. But in contrast to windows, which use PRE on input streams only, here, the old state in cell G52 comes from using PRE on the current state in cell I46, which is itself computed. Besides this security application, state machines are also useful in other stream processing domains, such as for detecting M-shape patterns in streams of stock quotes [14].

6 Related Work

This paper covers topics at the intersection of spreadsheet programming and stream processing.

Spreadsheets as a programming platform. The idea to use spreadsheets for coding is not new. In Haxcel, each cell can hold a Haskell definition [16]. Similarly, Wakeling also proposes a Haskell-based spreadsheet [22]. As in our approach, this is motivated by wanting to offer an interactive programming experience, where changes to code have immediate visible effects. Unlike our approach, these approaches assume that the programmer already knows Haskell, and these approaches do not attempt to tackle stream processing.

Woo et al. use spreadsheets as a tool for data analysis over sensor networks [23]. This work comes closer to streaming, since sensors continuously produce data. But the work is custom-tailored for the sensor domain, whereas we address stream processing more generally. Sestoft compiles spreadsheets to a functional implementation [20]. Like our work, this means exporting computation from a spreadsheet. Serafima augments spreadsheets to work with trees, motivated by processing XML data [19]. Neither Sestoft nor Serafima tackle using spreadsheets for stream processing.

McGarry augments spreadsheets with streaming data import and windows, but offers no feature to export data or code [17]. The StreamBase platform offers adapters that import and export data to Excel spreadsheets [21]. Like our work, this addresses programming with spreadsheets for stream processing. Unlike our work, the StreamBase Excel adapters export no code from the spreadsheet. Cloudscale uses Excel spreadsheets to configure streaming analytics [10]. Unlike our work, the user does not describe the analytics directly using the built-in computation features of Excel.

Programming models for streaming. Diverse programming models have been proposed to make it easier to write streaming applications. The programming languages community has developed several dedicated streaming languages, including LUSTRE for programming real-time controllers [8], StreamIt for programming many-cores [13], Lime for programming FPGAs [5], and SPL for programming distributed clusters [15]. These language-centric approaches enable advanced compiler optimizations, but require programmers to learn a new language.

Instead of requiring programmers to learn a new language, another approach is to build a library in an existing language. Spark Streaming, which is based on Scala, is an example for this [25]. However, Scala requires more sophistication, and has a smaller user base, than spreadsheets.

A popular approach for making programming of streaming applications more high-level is to offer not a full-fledged language, but simple patterns. Examples for this include SASE [24], Cayuga [11], and MatchRegex [14]. The patterns match over sequences of events to detect situations worthy of reporting. But while these might be easier to learn than a full language, they still come with a learning curve hindering wide-spread adoption.

The databases community tackles programming models for streaming by observing that many users are already familiar with SQL. Hence, approaches like CQL [4] or the language for Microsoft StreamInsight [2] use SQL as a starting point, and then add extensions such as windowing constructs for streaming. But for non-programmer end-users, spreadsheets are still more familiar than SQL.

At the far end of the spectrum, Mario requires no programming at all [7]. Instead, the user merely enters tags as they might in a web search engine. The system then guesses what might be the right stream program based on these tags. Like spreadsheets, this is immediately usable by end-users. Unlike spreadsheets, it offers far less control over what streaming application comes out in the end.

The formalization of our core calculus—choice of constructs, semantics, and properties—has a lot in common with synchronous programming languages [6]. It adopts the *synchrony hypothesis*: outputs are produced instantly so that inputs and outputs are formally synchronous. It has ticks but not clocks: arrival times are not required to be periodic or regular. It is asynchronous in that its constructs can compose arbitrary feeds irrespective of their relative arrival times. Feeds are implicitly sampled (i.e., re-clocked) when not in sync. As a consequence, we have no need for a clock calculus to ensure proper pairing and boundedness. While the calculus permits cyclic definitions, it guarantees causality. We choose to ensure causality by preventing timing cycles and making sure every value cycle includes a delay (a latch). While some synchronous programming languages such as ESTEREL favor more sophisticated causality analyses [18], we do not think these would be sensible extensions to the execution strategy of a typical spreadsheet. Because filtering is such an essential feature of our system, we choose to break timing cycles by explicitly clocking latches—separating the input tick from the input value—rather than introducing a delay in the @ construct, akin to delaying the reaction to absence in reactive programming models [3].

Our live calculus is not as expressive as higher-order synchronous models [9] but preserves the guarantees (bounded time and memory usage) of the core calculus in the presence of dynamically changing formulas.

7 Conclusion

This paper presents ActiveSheets, a system for visualizing and programming live streams in a spreadsheet. Stream processing has gained importance as many businesses have continuous data feeds, and analyzing these on-the-fly helps find opportunities and avoid risks. Using a spreadsheet makes streaming accessible to the end-user. Furthermore, a spreadsheet offers a very hands-on experience, since the data is manipulated directly where the user can see it, and interactive code changes have immediate visible effects. We formalize the semantics of ActiveSheets, and describe an implementation of ActiveSheets that uses Microsoft Excel as the client front-end. When the user programs a streaming application using ActiveSheets, he or she can elect to export either data or computation. Exported *data* can be further processed by the server, or can be used to initiate actions, such as alerts or sales. Exported *computation* can run directly on the server, and live on even when the client is closed. Since exported computation runs on the server, it saves the cost of communicating with the client; furthermore, it can be optimized and compiled to machine code. Overall, ActiveSheets enables end-users to author powerful and efficient streaming applications using familiar spreadsheet features.

Acknowledgements. We thank James Giles, Louis Mandel, and anonymous reviewers for their feedback and suggestions.

References

1. The Akka project, `http://akka.io` (retrieved November 2013)
2. Ali, M., Chandramouli, B., Goldstein, J., Schindlauer, R.: The extensibility framework in Microsoft StreamInsight. In: International Conference on Data Engineering (ICDE), pp. 1242–1253 (2011)
3. Amadio, R.M., Boudol, G., Castellani, I., Boussinot, F.: Reactive concurrent programming revisited. CoRR abs/cs/0512058 (2005)
4. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. Journal on Very Large Data Bases (VLDB J.) 15(2), 121–142 (2006)
5. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R.: Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 89–108 (2010)
6. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)

7. Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A., Riabov, A.: A tag-based approach for the design and composition of information processing applications. In: Onward! Track of Object-Oriented Programming, Systems, Languages, and Applications (Onward!), pp. 585–602 (2008)
8. Caspi, P., Pilaud, D., Halbwachs, N., Raymond, P.: Lustre: a declarative language for real-time programming. In: Symposium on Principles of Programming Languages (POPL), pp. 178–188 (1987)
9. Caspi, P., Pouzet, M.: Synchronous Kahn networks. In: International Conference on Functional Programming (ICFP), pp. 226–238 (1996)
10. Cloudscale big data analytics, `http://www.hashdoc.com/document/8626/big-data-analytics` (retrieved November 2013)
11. Demers, A., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.: Cayuga: A general purpose event monitoring system. In: Conference on Innovative Data Systems Research (CIDR), pp. 412–422 (2007)
12. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (2002)
13. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 151–162 (2006)
14. Hirzel, M.: Partition and compose: Parallel complex event processing. In: Conference on Distributed Event-Based Systems (DEBS), pp. 191–200 (2012)
15. Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M., Nasgaard, H., Schneider, S., Soulé, R., Wu, K.L.: IBM Streams Processing Language: Analyzing big data in motion. IBM Journal of Research & Development 57(3/4), 7:1–7:11 (2013)
16. Lisper, B., Malström, J.: Haxcel: A spreadsheet interface to Haskell. In: Workshop on the Implementation of Functional Languages (IFL), pp. 206–222 (2002)
17. McGarry, J.: Processing continuous data streams in electronic spreadsheets. Patent No. US 6,490,600 B1 (2002)
18. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel, 1st edn. Springer Publishing Company, Incorporated (2007)
19. Serafimova, I.: Spreadsheet-based template language prototype for tree data structure description and interpretation. In: International Conference on Computer Systems and Technologies (CompSysTech), pp. 148–154 (2012)
20. Sestoft, P.: Implementing function spreadsheets. In: Workshop on End-User Software Engineering (WEUSE), pp. 91–94 (2008)
21. StreamBase Microsoft Excel adapter, `http://docs.streambase.com/sb66/index.jsp?topic=/com.streambase.sb.ide.help/data/html/samplesinfo/Excel_sample.html` (retrieved November 2013)
22. Wakeling, D.: Spreadsheet functional programming. Journal of Functional Programming (JFP) 17(1), 131–143 (2007)
23. Woo, A., Seth, S., Olson, T., Liu, J., Zhao, F.: A spreadsheet approach to programming and managing sensor networks. In: Conference on Information Processing in Sensor Networks (IPSN), pp. 424–431 (2006)
24. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: International Conference on Management of Data (SIGMOD), pp. 407–418 (2006)
25. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Symposium on Operating Systems Principles (SOSP), pp. 423–438 (2013)