Bounded-Memory Stream Processing

Özgür L. Özçep

Institute of Information Systems (IFIS)
University of Lübeck
Lübeck, Germany
oezcep@ifis.uni-luebeck.de

Abstract. Foundational work on stream processing is relevant for different areas of AI and it becomes even more relevant if the work concerns feasible and scalable stream processing. One facet of feasibility is treated under the term bounded memory. In this paper, streams are represented as finite or infinite words and stream processing is modelled with stream functions, i.e., functions mapping one or more input stream to an output stream. Bounded-memory stream functions can process input streams by using constant space only. The main result of this paper is a syntactical characterization of bounded-memory functions by a form of safe recursion.

Keywords: streams, bounded memory, infinite words, recursion

1 Introduction

Stream processing has been and is still a highly relevant research topic in computer science and especially in AI. The main aspects of stream processing that one has to consider are illustrated nicely by the titles of some research papers: the ubiquity of streams due to the temporality of most data ("It's a streaming world!", [12]), the potential infinity of streams ("Streams are forever", [13]), or the importance of the order in which data are streamed ("Order matters", [34]).

These aspects are relevant for all levels of stream processing that occur in AI research and AI applications, in particular for stream processing on the sensor-data level, e.g., for agent reasoning on percepts, or on the relational data level, e.g., within data stream management systems. Recent interest on high-level declarative stream processing [11,6,31,28,24] w.r.t. an ontology have lead to additional aspects becoming relevant: The enduser accesses all possibly heterogeneous data sources (static, temporal and streaming) via a declarative query language using the signature of the ontology. The EU funded project CASAM[1], demonstrated how such a uniform ontology interface could be used to realize (abductive) interpretation of multimedia streaming data [18]. The efforts in the EU project OPTIQUE [2] [17] resulted in an extended OBDA system with a flexible, visual interface and mapping management system for accessing static data

[1] http://cordis.europa.eu/project/rcn/85475_en.html
[2] http://optique-project.eu/

(wellbore data provided by the industrial partner STATOIL) as well as temporal and streaming data (turbine measurements and event data provided by the industrial partner SIEMENS). This kind of convenience and flexibility for endusers leads to challenges for the designers of the stream engine as they have to guarantee complete and correct transformations of endusers' queries to low-level queries over the backend.

The main challenging fact of stream processing is the potential infinity of the data: It means that one cannot apply a one-shot query-answering procedure, but has to register queries that are evaluated continuously on streams. Independent of the kind of streams (low-level sensor streams or high-level streams of semantically annotated data), the aim is to keep stream processing feasible, in particular by minimizing the space resources required to process the queries. The kind of data structures used to store the relevant bits of information in the so-called synopsis (or summary or sketch [8]) may differ from application to application but sometimes one can describe general connections between the required space and the expressivity of the language for the representation of the stream query.

Bounded-memory queries on streams are allowed to use only constant space to store the relevant bits of informations of the growing stream prefix. This notion depends on the underlying computation model and so bounded-memory computation can be approached from different angles. Bounded-memory stream processing has been in the focus of research in temporal databases [7] under the term "bounded history encoding" and in research on data stream management systems [2,19] but it has been also approached in the area of theoretical informatics in the context of finite-memory automata [23], string-transducers [14,1,15] and from a co-algebraic perspective [32].

In this paper, bounded-memory stream processing is approached in the infinite word perspective of [20]. Streams are represented as finite or infinite words and stream processing is modelled by stream functions/queries, i.e., functions mapping one or more stream to an output stream. The important class of abstract computable functions (AC) are those representable by repeated applications of a kernel, alias window function, on the growing prefix of the input. Various other classes of interesting stream functions, which can be characterized axiomatically (see e.g. [27]), result by considering restrictions on the underlying window functions. The focus of this paper are AC functions with windows computable in bounded memory. The underlying computation model is that of streaming abstract state machines [20].

Though the restriction of constant space for bounded memory functions limits the set of expressible functions, the resulting class of streams functions is still expressive enough to capture interesting information needs over streams. In fact, in this paper it is shown that bounded-memory functions can be constructed using principles of linear primitive recursion. The main idea is to use a form of safe recursion of window function applications. The result is a rule set for inductively building functions on the base of basic functions. In familiar programming speak the paper gives a characterization of stream functions that correspond to programs using linearly bounded for-loops (and not arbitrary while loops).

2 Preliminaries

The following simple definition of streams of words over a finite or infinite alphabet D is used throughout this paper. An alphabet D is also called *domain* here.

Definition 1. *The set of* finite streams *is the set of finite words D^* over the alphabet D. The set of* infinite streams *is the set of ω-words D^ω over D. The set of (all) streams is denoted $D^\infty = D^* \cup D^\omega$.*

The basic definition of streams above is general enough to capture all different forms of streams, in particular those that are considered in the approaches mentioned in Sect. 4 on related work.

$D^{\leq n}$ is the set of words of length maximally n. For any finite stream s the length of s is denoted by $|s|$. For infinite streams s let $|s| = \infty$ for some fixed object $\infty \notin \mathbb{N}$. For $n \in \mathbb{N}$ with $1 \leq n \leq |s|$ let $s^{=n}$ be the n-th element in the stream s. For $n = 0$ let $s^{=n} = \epsilon =$ the empty word. $s^{\leq n}$ denotes the n-prefix of s, $s^{\geq n}$ is the suffix of s s.t. $s^{\leq n-1} \circ s^{\geq n} = s$. For an interval $[j,k]$, with $1 \leq j \leq k$, $s^{[j,k]}$ is the stream of elements of s such that $s = s^{\leq j-1} \circ s^{[j,k]} \circ s^{\geq k+1}$. For a finite stream $w \in D^*$ and a set of streams X the term $w \circ X$ or shorter wX denotes the set of all w-extensions with words from X: $wX = \{s \in D^\infty \mid$ There is $s' \in X$ s.t. $s = w \circ s'\}$. The finite word s is a prefix of a word s', for short $s \sqsubseteq s'$, iff there is a word v such that $s' = s \circ v$. If $s \sqsubseteq s'$, then $s' -_\sqsubseteq s$ is the suffix of s' when deleting its prefix s. If all letters of s occur in s' in the ordering of s (but perhaps not directly next to each other) then s is called a *subsequence* of s'. If $s' = usv$ for $u \in D^*$ and $v \in D^\infty$, then s is called a subword of s'. Streams are going to be written in the word notation, sometimes mentioning the concatenation $\circ$ explicitly. For a function $Q : D_1 \longrightarrow D_2$ and $Y \subseteq D_2$ let $Q^{-1}[Y] = Q^{-1}(Y) = \{w \in D_1 \mid Q(w) \in Y\}$ be the preimage of Y under Q.

The very general notion of an *abstract computable* [20] stream function is that of a function which is incrementally computed by calculations of finite prefixes of the stream w.r.t. a function called *kernel*. More concretely, let $K : D^* \longrightarrow D^*$ be a function from finite words to finite words. Then define the *stream query* $Repeat(K) : D^\infty \longrightarrow D^\infty$ *induced by kernel* K as

$$Repeat(K) : s \mapsto \bigcirc_{j=0}^{|s|} K(s^{\leq j})$$

Definition 2. *A query Q is* abstract computable *(AC) iff there is a kernel such that $Q(s) = Repeat(K)(s)$.*

Using a more familiar speak from the stream processing community, the kernel operator is a *window operator*, more concretely, an unbounded window operator. The "window" terminology is the preferred one in this paper.

That abstract computability is an adequate concept for stream processing can be formally undermined by showing that exactly the AC functions fulfill two fundamental properties: AC functions are prefixed determined (FP^∞) and

they are data-driven in the sense that they map finite streams to finite streams (F2F).

(FP$^\infty$) For all $s \in D^\infty$ and all $u \in D^*$: If $Q(s) \in uD^\infty$, then there is a $w \in D^*$ s.t. $s \in wD^\infty \subseteq Q^{-1}[uD^\infty]$.

(F2F) For all $s \in D^*$ it holds that: $Q(s) \in D^*$.

The following theorem states the representation result:

Theorem 1 ([20]). *AC queries represent the class of stream queries fulfilling (F2F) and (FP$^\infty$).*

Multiple (input) streams can be handled in the framework of [20] by attaching to the domain elements tags with provenance information, in particular information on the stream source from which the element originates. This is the general strategy in the area of complex event processing (CEP), where there is exactly one (mega)-stream on which event patterns are evaluated. But this tag-approach appears in some situation to be too simple as it provides no control on how to interleave the stream inputs—as it is required, e.g., for state-of-the art stream query languages following a pipeline architecture. Actually, in this paper the framework of [20] is generalized to handle functions on multiple streams genuinely as functions of the form $Q : D^\infty \times \cdots \times D^\infty \longrightarrow D^\infty$—similar to the approach of [35].

3 Bounded-Memory Queries

The notion of abstract computability is very general, even so as to contain also queries that are not computable by a Turing machine according to the notion of TTE computability [35]. Hence, the authors of [20] consider the refined notion of *abstract computability modulo a class* $\mathcal{C}$ meaning that the window K inducing an abstract computable query has to be in $\mathcal{C}$. In most cases, $\mathcal{C}$ stands for a family of functions of some complexity class. In [20], the authors consider variants of $\mathcal{C}$ based on computations by a machine model called *stream abstract state machine (*sAsm*)*. In particular, they show that every AC query induced by a length-bounded window (in particular: each so-called synchronous AC query: window-length always 1) is computable by an sAsm [20, Corollary 23].

A particularly interesting class from the perspective of efficient computation are bounded-memory sAsms because these implement the idea of incrementally maintainable windows requiring only a constant amount of memory. (For a more general notion of incremental maintainable queries see [29].) Of course, the space restrictions of bounded-memory sAsms are strong constraints on the expressiveness of stream functions, e.g., it is not possible to compute the INTERSECT problem of checking whether prior to some given timepoint t there were identical elements in two given streams [20, Proposition 26] with a bounded-memory sAsm. A slightly more general version of bounded-memory sAMS are

o(n)-bitstring sAMS which store, on every stream and every step, only o(n) bitstrings. (But neither can these compute INTERSECT [20, Proposition 28].)

An sAsm operates on first-order sorted structures with a static part and a dynamic part. The static part contains all functions allowed over the domain of elements D of the streams. The dynamic part consists of functions which may change by transitions in an update process. A set of nullary functions *in* and *out* is pre-defined and are used to describe registers for the input, output data stream elements, resp. Updates are the basic transitions. Based on these, simple programs are defined as finite sequences of rules: The basic rules are updates $f(t_1, \ldots, t_n) := t_0$, meaning that in the running state terms $t_0, t_1, \ldots, t_n$ are evaluated and then used to redefine the (new) value of f. Then, inductively, one is allowed to apply to update rules a parallel execution constructor par that allows parallel firing of the rule; and also, inductively, if rules r_1, r_2 are constructed, then one can build the "if-then-else construct": if Q then r_1 else r_2.Here the if-condition is given by a quantifier free formula Q on the signature of the structure and where the post-conditions are r_1, r_2. For *bounded-memory* sAsm [20, Def. 24] one additionally requires that *out* registers do not occur as arguments to a function, that all dynamic functions are nullary and that non-nullary static functions can be applied only to rules of the form $out := t_0$.

3.1 Constant-Width Windows

In this subsection we are going to consider an even more restricted class of bounded-memory windows, namely those based on constant-width windows. For this, let us recapitulate the definitions (and some result) that were given in [27].

The general notion of an n-kernel which corresponds to the notion of a finite window of width n is defined as follows:

Definition 3. *A function $K : D^* \longrightarrow D^*$ that is determined by the n-suffixes ($n \in \mathbb{N}$), i.e., a function that fulfills for all words $w, u \in D^*$ with $|w| = n$ the condition $K(uw) = K(w)$ is called an* n-window. *If additionally $K(s) = \epsilon$, for all s with $|s| < n$, then K is called a* normal n-window. *The set of stream queries generated by an n-window for some $n \in \mathbb{N}$ are called* n-window abstract computable *stream queries, for short* n-WAC *operators. The union* WAC $= \bigcup_{n\in\mathbb{N}}$ *n-WAC is the set of* window abstract computable *stream queries.*

The class of WAC queries can be characterized by a generalization of a distribution property called (Factoring-n) that, for each $n \in \mathbb{N}$, captures exactly the n-window stream queries.

(Factoring-n) $\forall s \in D^*$: $Q(s) \in D^*$ and

1. if $|s| < n$, $Q(s) = \epsilon$ and
2. if $|s| = n$, for all $s' \in D^\infty$ with $|s'| \geq 1$: $Q(s \circ s') = Q(s) \circ Q((s \circ s')^{\geq 2})$.

Proposition 1. *[27] For any $n \in \mathbb{N}$ with $n \geq 1$, a stream query $Q : D^\infty \longrightarrow D^\infty$ fulfills* (Factoring-n) *iff it is induced by a normal n-window K.*

Intuitively, the class of WAC stream queries is a proper class of AC stream queries because the former consider only fixed-size finite portions of the input stream whereas for AC stream queries the whole past of an input stream is allowed to be used for the production of the output stream. A simple example for an AC query that is not a WAC query is the parity query PARITY : $\{0,1\}^\infty \longrightarrow \{0,1\}^\infty$ defined as $\mathit{Repeat}(K_{par})$. Here, K_{par} is the parity window function $K : \{0,1\}^* \longrightarrow \{0,1\}$ defined as $K_{par}(s) = 1$, if the number of 1s in s is odd and $K_{par}(s) = 0$ else. The window K_{par} is not very complex, indeed one can show that K_{par} is a bounded-memory function w.r.t. the sASM model or, simpler, w.r.t. the model of finite automata: It is easy to find a finite automaton with two states that accepts exactly those words with an odd number of 1s and rejects the others. In other words: parity is incrementally maintainable. But finite windows are "stateless", they cannot memorize the actual parity seen so far. Formally, it is easy to show that any constant-width window function is AC^0 computable, i.e., computable by a polynomial number of processors in constant time: For any word length m construct a circuit with m inputs where only the first n of them are actually used: One encodes all the 2^n values of the n-window K in a boolean circuit BC_m, the rest of the m word is ignored. All BC_m have the same size and depth and hence a finite window function is in AC^0. On the other hand it is well known by a classical result [16] that PARITY is not in AC^0.

3.2 A Recursive Characterization of Bounded-Memory Functions

Though the machine-oriented approach for the characterization of bounded-memory stream functions with sASMs is quite universal and fits into the general approach for characterizing computational classes, the following considerations add a simple, straight-forward characterization following the idea of primitive recursion over words [22,3]: Starting from basic functions on finite words, the user is allowed to built further functions by applying composition and simple forms of recursion. In order to guarantee bounded memory, all the construction rules are built with specific window operators, namely $\mathit{last}_n(\cdot)$, which output the n-suffix of the input word. This construction gives the user the ability to built (only) bounded-memory window functions K in a pipeline strategy. The main adaptation of the approach of [20] is adding recursion for n-window kernels. This leads to a more fine-grained approach for kernels K. In particular, now, it is possible to define the PARITY query with n-window Kernels whereas without recursion, as shown in the example before, it is not.

It should be noted that in agent theory usually the processing of streams is described by functions that take an evolvement of states into account: Depending on the current state and the current percept, the agent chooses the next action and the next state. In this paper, a different approach is described which is based on the principle of tail recursion where the accumulators play the role of states.

In order to enable a pipeline-based construction the approach of [20] is further extended by considering multiple streams explicitly as possible arguments for functions with an arbitrary number of arguments. Still, all functions will output a single finite or infinite word—though the approach sketched below can easily

be adapted to work for multi-output streams. All of the machinery of Gurevich's framework is easily translated to this multi-argument setting. So, for example the axiom (FP^∞) now reads as follows:

($\mathbf{FP}^\infty$) For all $s_1, \ldots s_n \in D^\infty$, and all $u \in D^*$: If $Q(s_1, \ldots, s_n) \in uD^\infty$, then there are $w_1, \ldots, w_n \in D^*$ such that $s_i \in w_i D^\infty$ for all $i \in [n]$ and $w_1 D^\infty \times \cdots \times w_n D^\infty \subseteq Q^{-1}(uD^\infty)$.

Monotonicity of a function $Q : (D^\infty)^n \longrightarrow D^\infty$ now reads as: For all $(s_1, \ldots, s_n)$ and $(s'_1, \ldots, s'_n)$ with $s_i \sqsubseteq s'_i$ for all $i \in [n]$: $Q(s_1, \ldots, s_n) \sqsubseteq Q(s'_1, \ldots, s'_n)$.

The temporal model behind the recursion used in Definition 4 is the following: At every time point one has exactly n elements to consume, exactly one for each of the n input streams. These are thought to appear at the same time. To model also the case where no element arrives in some input stream, a specific symbol $\perp$ can be added to the system. Giving the engine a finite word as input means that the engine gets noticed about the end of the word (when it has read the word). In a real system this can be handled, e.g., the idea of punctuation semantics [33]. Of course, then there is a difference between the finite word abc, where the system can stop listening for the input after 'c' was read in, and the infinite word $abc(\perp)^\omega$, where the system gets notified at every time point that there is no element at the current time.

A further extension of the framework in [20] is that we add to the set of rules a co-recursive/co-inductive rule [32], in order to describe directly bounded-memory queries $Q = \mathit{Repeat}(K)$—instead of only the underlying windows K. This class is denoted MonBmem in Definition 4.

Three types of classes are defined in parallel: classes $\textsc{Accu}^n$ which are intended to model accumulator functions $f : (D^*)^n \longrightarrow D^*$; classes $\textsc{Bmem}^{(n;m)}$ that model incrementally maintainable functions with bounded memory, i.e., window functions that are bounded-memory and have bounded output, and classes MonBmem(n;m) of incrementally maintainable, memory-bounded, and monotonic functions that lead to the definition of monotonic functions on infinite streams. The main idea, similar to that of [3], is to partition the argument functions in two classes, normal and safe arguments. In [3] the normal variables are the ones on which the recursion step happens and which have to be controlled, whereas the safe ones are those in which the growth of the term is not restricted. In the definitions, the growth (the length) of the words is controlled explicitly and the distinction between input and output arguments is used: The input arguments are those where the input may be either a finite or an infinite word. The output variables are the ones in which the accumulation happens. In a function term $f(x_1, \ldots, x_n; y_1, \ldots, y_m)$ the input arguments are the ones before the semicolon ";", here: $x_1, \ldots, x_n$, and the output arguments are the ones after the ";", here: $y_1, \ldots, y_n$.

Using the notation of [22] for my purposes, a function f with n input and m output arguments is denoted $f^{(n;m)}$. Classes $\textsc{Bmem}^{(n;m)}$ and $\textsc{MonBmem}^{(n;m)}$ consist of functions of the form $f^{(n;m)}$. The class MonBmem defined as the union $\bigcup_{n\in\mathbb{N}} \textsc{MonBmem}^{(n;)}$ contains all functions without output variables and

is the class of functions which describe the prefix restrictions $Q_{\restriction D^*}$ of stream queries $Q : D^\infty \longrightarrow D^\infty$ that are computable by a bounded-memory sAsm.

Definition 4. *Let $n, m \in \mathbb{N}$ be natural numbers (including zero). The set of* bounded n-ary accumulator word functions*, for short* $\textsc{Accu}^n$*, the set of $(n+m)$-ary* bounded-memory incremental functions *with n input and m output arguments, for short* $\textsc{Bmem}^{(n;m)}$*, and the set of* monotonic, bounded-memory incremental $(n+m)$-ary functions *with n input and m output arguments, for short* $\textsc{MonBmem}^{(n;m)}$*, are defined according to the following rules:*

1. $w \in \textsc{Accu}^0$ *for any word* $w \in D^*$ *("Constants")*
2. $last_k(\cdot) \in \textsc{Accu}^1$ *for any* $k \in \mathbb{N}$ *("Suffixes")*
3. $S_k^a(w) = last_k(w) \circ a \in \textsc{Accu}^1$ *for any* $a \in D$ *("Successors")*
4. $P_k(w) = last_{k-1}(w) \in \textsc{Accu}^1$ *("Predecessors")*
5. $cond_{k,l}(w, v, x) = \begin{cases} last_k(v) & \text{if } last_1(w) = 0 \\ last_l(x) & \text{else} \end{cases} \in \textsc{Accu}^3$ *("Conditional")*
6. $\Pi_k^j(w_1, \ldots, w_n) = last_k(w_j) \in \textsc{Accu}^n$ *for any* $k \in \mathbb{N}$ *and* $j \in [n]$*,* $n \neq 0$*.* *("Projections")*
7. $shl(\cdot)^{(1;0)} \in \textsc{MonBmem}$ *with* $shl(aw;) = w$ *and* $shl(\epsilon;) = \epsilon$*.* *("Left shift")*
8. *Conditions for Composition* *("Composition")*
   (a) *If* $f \in \textsc{Accu}^n$ *and, for all* $i \in [n]$*,* $g_i \in \textsc{Accu}^m$*, then also* $f(g_1, \ldots, g_n) \in \textsc{Accu}^m$*; and:*
   (b) *If* $g^{(m;n)} \in \textsc{MonBmem}^{(m;n)}$ *and, for all* $i \in [m]$*,* $g_i \in \textsc{Accu}^l$ *and* $h_j^{(k;l)} \in \textsc{MonBmem}^{(k;m)}$ *for* $j \in [n]$*, then* $f^{(k;l)} \in \textsc{MonBmem}^{(k;l)}$ *where using* $\boldsymbol{w} = w_1, \ldots, w_k$*,* $\boldsymbol{v} = v_1, \ldots, v_l$
   $$f^{(k;l)}(\boldsymbol{w}; \boldsymbol{v}) = g^{(m;n)}(h_1(\boldsymbol{w}; \boldsymbol{v}), \ldots, h_m(\boldsymbol{w}; \boldsymbol{v}); g_1(\boldsymbol{v}), \ldots, g_n(\boldsymbol{v}))$$
   (c) *If* $g^{(m;n)} \in \textsc{Bmem}^{(m;n)}$ *and, for all* $i \in [m]$*,* $g_i \in \textsc{Accu}^l$ *and* $h_j^{(k;l)} \in \textsc{MonBmem}^{(k;m)}$ *for* $j \in [n]$*, then* $f^{(k;l)} \in \textsc{Bmem}^{(k;l)}$ *where using* $\boldsymbol{w} = w_1, \ldots, w_k$*,* $\boldsymbol{v} = v_1, \ldots, v_l$
   $$f^{(k;l)}(\boldsymbol{w}; \boldsymbol{v}) = g^{(m;n)}(h_1(\boldsymbol{w}; \boldsymbol{v}), \ldots, h_m(\boldsymbol{w}; \boldsymbol{v}); g_1(\boldsymbol{v}), \ldots, g_n(\boldsymbol{v}))$$
9. *If* $g : (D^*)^n \longrightarrow D^* \in \textsc{Accu}$ *and* $h : (D^*)^{n+3} \longrightarrow D^* \in \textsc{Accu}$ *then also* $f : (D^*)^{n+1} \longrightarrow D^* \in \textsc{Accu}$*, where:*
   $$f(\epsilon, v_1, \ldots, v_n) = g(v_1, \ldots, v_n)$$
   $$f(wa, v_1, \ldots, v_n) = h(w, a, v_1, \ldots v_n, f(w, v_1, \ldots, v_n))$$
   ("Accu-Recursion")
10. *If* $g_i : (D^*)^{n+m} \longrightarrow D^* \in \textsc{Accu}$ *for* $i \in [m]$*,* $g_0 \in \textsc{Accu}$ *then* $k = k^{(n;m)} \in \textsc{Bmem}^{(n;m)}$*, where k is defined using the above abbreviations as follows:*
    $$k(\epsilon, \ldots, \epsilon; \boldsymbol{v}) = g_0(\boldsymbol{v})$$
    $$k(\boldsymbol{w}; \boldsymbol{v}) = k(shl(\boldsymbol{w}); g_1(\boldsymbol{v}, \boldsymbol{w}^{=1}), \ldots, g_m(\boldsymbol{v}, \boldsymbol{w}^{=1}))$$
    ("Window-Recursion")

11. *If* $g_i : (D^*)^{n+m} \longrightarrow D^* \in \textsc{Accu}$ *for* $i \in [m]$, $g_0 \in \textsc{Accu}$, *then* $f = f^{(n;m)} \in \textsc{MonBmem}^{(n;m)}$, *where* f *is defined using the above abbreviations as follows:*

$$f(\epsilon, \ldots, \epsilon; out, \boldsymbol{v}) = out$$
$$f(\boldsymbol{w}; out, \boldsymbol{v}) = f(shl(\boldsymbol{w}); out \circ g_1(\boldsymbol{v}, \boldsymbol{w}^{=1}), g_1(\boldsymbol{v}, \boldsymbol{w}^{=1}), \ldots, g_m(\boldsymbol{v}, \boldsymbol{w}^{=1}))$$

("Repeat-Recursion")

Let $\textsc{MonBmem} = \bigcup_{n\in\mathbb{N}} \textsc{MonBmem}^{(n;)}$.

Within the definition above, three types of recursions occur: the first is a primitive recursion over accumulators. The second, called window-recursion, is a specific form of *tail recursion* which means that the recursively defined function is the last application in the recursive call. As the name indicates, this recursion rule is intended to model the kernel/window functions. The last recursion rule (again in tail form) is intended to mimic the *Repeat* functional.

In the first recursion, the word is consumed from the end: This is possible, as the accumulators are built from left to right during the streaming process. Note, that the length of outputs produced by the accu-recursion rule and the window-recursion rule are length-bounded.

The window-recursion rule and the repeat-recursion rule implement a specific form of tail recursion consuming the input words from the beginning with the left-shift function *shl*(). This is required as the input streams are potentially infinite. Additionally, these two rules implement a form of simultaneous recursion, where all input words are consumed in parallel according to the temporal model mentioned above.

Repeat recursion is illustrated with the following simple example.

Example 1. Consider the window function K_{par} that, for a word w, outputs its parity. The monotonic function $Par(w) = Repeat(K_{par})(w) = \bigcirc_{j=0}^{|w|} K_{par}(w^{\leq j})$ can be modelled as follows. The auxiliary xor function $\oplus$ can be defined with cond because with cond one can define the functionally complete set of junctions $\{\neg, \wedge\}$ with $\neg x := cond_{1,1}(x, 1, 0)$ and $x \wedge y = cond_{1,1}(x, 0, y)$. Using repeat recursion (item 11 in Definition 4) gives the desired function.

$$f(\epsilon; out, v) = out$$
$$f(w; out, v) = f(shl(w); out \circ v \oplus w^{=1}, v \oplus w^{=1})$$
$$Par(w) = f(w; \epsilon, 0)$$

For example, the input word $w = 101$ is consumed as follows:

$$\begin{aligned} Par(101) &= f(101; \epsilon, 0) = f(shl(101); \epsilon \circ 0 \oplus 101^{=1}, 0 \oplus 101^{=1}) \\ &= f(01; \epsilon \circ 0 \oplus 1, 0 \oplus 1) = f(01; 1, 1) \\ &= f(1; 1 \circ 1 \oplus 0, 1 \oplus 0) = f(1; 1 \circ 1, 1) \\ &= f(\epsilon; 1 \circ 1 \oplus 1, 1 \oplus 1) = f(\epsilon; 1 \circ 1 \circ 0, 0) = 110 \end{aligned}$$

The output of the repeat-recursion grows linearly: The whole history is outputted with the help of the concatenation function. Note that the concatenation functions appears only in the repeat-recursion rule and also—in a restricted form—in the successor functions, but there is no concatenation function defined in one of the three classes (as it is not a bounded-memory function). The repeat-recursion function builds the output word by concatenating intermediate results in the out variable. Because of this, it follows that all functions in MonBmem are monotonic in their input arguments. This is stated in the following proposition:

Proposition 2. *All functions in* MonBmem *are monotonic.*

Proof (sketch). Let us introduce the notion of a function $f(\boldsymbol{x};\boldsymbol{y})$ being monotonic w.r.t. its arguments $\boldsymbol{x}$: This is the case if for every $\boldsymbol{y}$ the function $f_{\boldsymbol{y}}(\boldsymbol{x}) = f(\boldsymbol{x},\boldsymbol{y})$ is monotonic. The functions in MonBmem are either the left shift function (which is monotonic) or a function constructed with the application of composition, which preserves monotonicity, or by repeat-recursion, which, due to the concatenation in the output position, also guarantees monotonicity. □

The functions in MonBmem map (vectors of) finite words to finite words. Because of the monotonicity, it is possible to define for each $f \in$ MonBmem an extension $\tilde{f}$ which maps (vectors) of finite or infinite words to finite or infinite words. If $f^{(n;)} : (D^*)^n \longrightarrow D^*$, then $\tilde{f} : (D^\infty)^n \longrightarrow D^\infty$ is defined as follows: If all $s_i \in D^*$, then $\tilde{f}(s_1,\ldots,s_n) = f(s_1,\ldots,s_n)$. Otherwise, $\tilde{f}(s_1,\ldots,s_n) = sup_{i\in\mathbb{N}} f(s_1^{\leq i},\ldots,s_n^{\leq i})$ where $sup_{i\in\mathbb{N}} f(s_1^{\leq i},\ldots,s_n^{\leq i})$ is the unique stream $s \in D^\infty$ such that $f(s_1^{\leq i},\ldots,s_n^{\leq i}) \sqsubseteq s$ for all i. Let us denote by BmemStr those functions Q that can be presented as $Q = \tilde{f}$ for some $f \in$ MonBmem and call them *bounded-memory* stream queries.

Theorem 2. *A function Q with one argument belongs to* BmemStr *iff it is a stream query computable by a bounded-memory* sAsm.

Proof (sketch). Clearly, the range of each function f in Bmem is length-bounded, i.e., there is $m \in \mathbb{N}$ such that for all $w \in D^* : |f(w)| \leq m$. But then, according to [20, Proposition 22], f can be computed by a bounded-memory sAsm. As the *Repeat* functional does (nearly) nothing else than the repeat-recursion rule, one gets the desired representation.

The other direction is more advanced but can be mimicked as well: All basic rules, i.e. update rules, can be modelled by Accu functions (as one has to store only one symbol of the alphabet in each register; the update is implemented as accu-recursion). The parallel application is modelled by the parallel recursion principle in window-recursion. The if-construct can be simulated using cond. And the quantifier-free formula in the if construct can also be represented using cond as the latter is functionally complete. □

Note that in a similar way one can model $o(n)$ bitstring bounded sAsm: Instead of using constant size windows $last_k(c)$ in the definition of accumulator functions, one uses dynamic windows $last_{f(\cdot)}(\cdot)$, where, for a sublinear function $f \in o(n)$, $last_{f(|w|)}(w)$ denotes the $f(|w|)$ suffix of w.

4 Related Work

The work presented here is based on the foundation of stream processing according to [20] which considers streams as finite or infinite words. The research on streams from the word perspective is quite mature and the literature on infinite words, language characterizations, and associated machine models abounds. The focus in this paper is on bounded-memory functions and their representation by some form of recursion. For all other interesting topics and relevant research papers the reader is referred to [35] and [30].

The construction of bounded-memory queries given in this paper are based on the *Repeat* functional applied to a window function. An alternative representation by trees is given in [21]: An (infinite) input word is read as sequence of instructions to follow the tree, 0 for left and 1 for right. The leaves of the tree contain the elements to be outputted. The authors give a characterization for the interesting case where the range of the stream query is a set of infinite words: In this case they have to use non-well-founded trees. Note, that in this type of representation the construction principle becomes relevant. Instead of a simple instantiation with a parameter value, one has to apply an algorithm in order to build the structure (here: the function).

In [20] and in this paper, the underlying alphabet for streams is not necessarily finite. This is similar to the situation in research on data words [5], where the elements of the stream have next to an element from a finite alphabet also an element from an infinite alphabet.

Aspects of performant processing on streams are touched in this paper with the construction of a class of functions capturing exactly those queries computable by an sAsm. This characterization is in the tradition of implicit complexity as developed in the PhD thesis of Bellantoni [4] which is based on work of Leivant [25]. (See also the summary of the thesis in [3] where the main result is the characterization of polynomial time functions by some form of primitive recursion). The main idea of distinguishing between two sorts of variables in my approach comes from [4], the use of constant, $o(n)$ size windows to control the primitive recursion is similar to the approach of [26] used for the rule called "bounded recursion" therein.

The consideration of bounded memory in [2] is couched in the terminology of data-stream management systems. The authors of [2] consider first-order logic (FOL) or rather: (non-recursive) SQL as the language to represent windows. The main result is a syntactical criterion for deciding whether a given FOL formula represents a bounded-memory query. Similar results in the tradition of Büchis result on the equivalence of finite-automata recognizability with definability in second-order logic over the sequential calculus can be shown for streams in the word perspective [14,1].

An aspect related to bounded memory is that of incremental maintainability as discussed in the area called dynamic complexity [29,36]. Here the main concern is to break down a query on a static data set into a stream query using simple update operators with small space.

The function-oriented consideration of stream queries along the line of this paper and [20] lends itself to a pipeline-style functional programming language on streams. And indeed, there are some examples, such as [9], that show the practical realizability of such a programming language.

The type of recursion that was used in order to handle infinite streams, namely the rules of window-revision and repeat-revision, uses the consumption of words from the beginning. This is similar to the co-algebraic approach for defining streams and stream functions [32].

5 Conclusion

Based on the foundational stream framework of [20], this paper gives a recursive characterization of bounded-memory functions. Though the achieved results have a foundational character, they are useful for applications relying, say, on the agent paradigm where stream processing plays an important role. The recursive style that was used to define the set of bounded-memory functions can be understood as a formal foundation for a functional style programming language for bounded-memory functions.

The present paper is one step towards axiomatically characterizing practically relevant stream functions for agents [27]. The axiomatic characterizations considered in [27] are on a basic phenomenological level—phenomenological, because only observations regarding the input-output behavior are taken into account, and basic, because no further properties regarding the structure of the data stream elements are presupposed. The overall aim, which motivated the research started in [27] and continued in this paper, is to give a more elaborated characterization of rational agents where also the observable properties of various higher-order streams of states such beliefs or goals are taken into account.

For example, if considering the stream of epistemic states $\Phi_1, \Phi_2, \ldots$ of an agent, an associated observable property is the set of beliefs $Bel(\Phi_i)$ an agent is obliged to believe in its current state Φ_i. The beliefs can be expressed in some logic which comes with an entailment relation $\models$. Using the entailment relation, the idea of a rational change of beliefs of the agent under new information can be made precise. For example, the success axiom expresses an agent's "trust" in the information it receives: If it receives α, then the current state Φ_i is required to develop into state Φ_{i+1} such that $Bel(\Phi_{i+1}) \models \alpha$. The constraining effects that this axiom has on the belief-state change may appear simple but, at least when the new information is not consistent with the current beliefs, it is not clear how the change has to be carried out. Axioms such as the success axiom are one of the main objects of study in the field of belief revision. But what is still missing in current research is the combination of belief-revision axioms (in particular those for iterated belief revision [10]) with axioms expressing basic stream-properties.

References

1. Alur, R., Cerný, P.: Expressiveness of streaming string transducers. In: Lodaya, K., Mahajan, M. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India. vol. 8, pp. 1–12 (2010)
2. Arasu, A., Babcock, B., Babu, S., McAlister, J., Widom, J.: Characterizing memory requirements for queries over continuous data streams. ACM Trans. Database Syst. **29**(1), 162–194 (Mar 2004)
3. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity **2**(2), 97–110 (1992)
4. Bellantoni, S.J.: Predicative Recursion and Computation Complexity. Ph.D. thesis, Graduate Department of Computer Science, University of Toronto (1992)
5. Benedikt, M., Ley, C., Puppis, G.: Automata vs. logics on data words. In: Dawar, A., Veith, H. (eds.) Computer Science Logic, LNCS, vol. 6247, pp. 110—124. Springer Berlin Heidelberg (2010)
6. Calbimonte, J.P., Mora, J., Corcho, O.: Query rewriting in rdf stream processing. In: Proceedings of the 13th International Conference on The Semantic Web. Latest Advances and New Domains - Volume 9678. pp. 486–502. Springer (2016)
7. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. **20**(2), 149–186 (Jun 1995)
8. Cormode, G.: Sketch techniques for approximate query processing. In: Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases. NOW publishers (2011)
9. Cowley, A., Taylor, C.J.: Stream-oriented robotics programming: The design of roshask. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 1048–1054 (Sept 2011)
10. Darwiche, A., Pearl, J.: On the logic of iterated belief revision. Artificial intelligence **89**, 1–29 (1997)
11. Della Valle, E., Ceri, S., Barbieri, D., Braga, D., Campi, A.: A first step towards stream reasoning. In: Domingue, J., Fensel, D., Traverso, P. (eds.) Future Internet – FIS 2008, LNCS, vol. 5468, pp. 72–81. Springer (2009)
12. Della Valle, E., Ceri, S., van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. Intelligent Systems, IEEE **24**(6), 83–89 (nov-dec 2009)
13. Endrullis, J., Hendriks, D., Klop, J.W.: Streams are forever. Bulletin of the EATCS **109**, 70–106 (2013)
14. Engelfriet, J., Hoogeboom, H.J.: Mso definable string transductions and two-way finite-state transducers. ACM Trans. Comput. Logic **2**(2), 216–254 (Apr 2001)
15. Filiot, E.: Logic-automata connections for transformations. In: Banerjee, M., Krishna, S.N. (eds.) Logic and Its Applications: 6th Indian Conference, ICLA 2015, Mumbai, India, January 8-10, 2015. Proceedings. pp. 30–57. Springer (2015)
16. Furst, M., Saxe, J.B., Sipser, M.: Parity, circuits, and the polynomial-time hierarchy. Theory of Computing Systems **17**, 13–27 (1984)
17. Giese, M., Soylu, A., Vega-Gorgojo, G., Waaler, A., Haase, P., Jiménez-Ruiz, E., Lanti, D., Rezk, M., Xiao, G., Özçep, Ö.L., Rosati, R.: Optique: Zooming in on big data. IEEE Computer **48**(3), 60–67 (2015)
18. Gries, O., Möller, R., Nafissi, A., Rosenfeld, M., Sokolski, K., Wessel, M.: A probabilistic abduction engine for media interpretation based on ontologies. In: Alferes, J., Hitzler, P., Lukasiewicz, T. (eds.) Proc. International Conference on Web Reasoning and Rule Systems (RR-2010) (2010)

19. Grohe, M., Gurevich, Y., Leinders, D., Schweikardt, N., Tyszkiewicz, J., Van den Bussche, J.: Database query processing using finite cursor machines. Theory of Computing Systems **44**(4), 533–560 (2009)
20. Gurevich, Y., Leinders, D., Van Den Bussche, J.: A theory of stream queries. In: Proceedings of the 11th International Conference on Database Programming Languages. pp. 153–168. DBPL'07, Springer-Verlag, Berlin, Heidelberg (2007)
21. Hancock, P., Pattinson, D., Ghani, N.: Representations of Stream Processors Using Nested Fixed Points. Logical Methods in Computer Science **5**(3:9), 1–17 (2009)
22. Handley, W.G., Wainer, S.S.: Complexity of primitive recursion. In: Berger, U., Schwichtenberg, H. (eds.) Computational Logic: Proceedings of the NATO Advanced Study Institute on Computational Logic, held in Marktoberdorf, Germany, July 29–August 10, 1997. pp. 273–300. Springer (1999)
23. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (Nov 1994)
24. Kharlamov, E., Mailis, T., Mehdi, G., Neuenstadt, C., Özçep, O.L., Roshchin, M., Solomakhina, N., Soylu, A., Svingos, C., Brandt, S., Giese, M., Ioannidis, Y., Lamparter, S., Möller, R., Kotidis, Y., Waaler, A.: Semantic access to streaming and static data at siemens. Web Semantics pp. 54–74 (2017)
25. Leivant, D.: A foundational delineation of poly-time. Information and Computation **110**(2), 391–420 (1994)
26. Lind, J., Meyer, A.R.: A characterization of log-space computable functions. SIGACT News **5**(3), 26–29 (Jul 1973)
27. Özçep, Ö.L., Möller, R.: Towards foundations of agents reasoning on streams of percepts. Proceedings of the 31st International Florida Artificial Intelligence Research Society Conference (FLAIRS-18) (2018)
28. Özçep, Özgür.L., Möller, R., Neuenstadt, C.: A stream-temporal query language for ontology based data access. In: KI 2014. LNCS, vol. 8736, pp. 183–194. Springer International Publishing Switzerland (2014)
29. Patnaik, S., Immerman, N.: Dyn-fo: A parallel, dynamic complexity class. Journal of Computer and System Sciences **55**(2), 199–209 (1997)
30. Perrin, D., Pin, J.: Infinite Words: Automata, Semigroups, Logic and Games. Pure and Applied Mathematics, Elsevier Science (2004)
31. Phuoc, D.L., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) Proceedings of the 10th International Semantic Web Conference (ISWC 2011), October 23-27, 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer (2011)
32. Rutten, J.J.M.M.: A coinductive calculus of streams. Mathematical. Structures in Comp. Sci. **15**(1), 93–147 (Feb 2005)
33. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. on Knowl. and Data Eng. **15**(3), 555–568 (Mar 2003)
34. Valle, E.D., Schlobach, S., Krötzsch, M., Bozzon, A., Ceri, S., Horrocks, I.: Order matters! Harnessing a world of orderings for reasoning over massive data. Semantic Web **4**(2), 219–231 (2013)
35. Weihrauch, K.: Computable Analysis: An Introduction. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2000)
36. Zeume, T., Schwentick, T.: Dynamic conjunctive queries. In: Schweikardt, N., Christophides, V., Leroy, V. (eds.) Proc. 17th International Conference on Database Theory (ICDT), March 24-28, 2014. pp. 38–49. OpenProceedings.org (2014)