Towards Efficient Feedback Control in Streaming Computer Vision Pipelines

Mohamed A. Helala, Ken Q. Pu, Faisal Z. Qureshi

Faculty of Science, University of Ontario Institute of Technology
Oshawa, ON, Canada
{Mohamed.Helala, Ken.Pu, Faisal.Qureshi}@uoit.ca

Abstract. Stream processing is currently an active research direction in computer vision. This is due to the existence of many computer vision algorithms that can be expressed as a pipeline of operations, and the increasing demand for online systems that process image and video streams. Recently, a formal stream algebra has been proposed as an abstract framework that mathematically describes computer vision pipelines. The algebra defines a set of concurrent operators that can describe a pipeline of vision tasks, with image and video streams as operands. In this paper, we extend this algebra framework by developing a formal and abstract description of feedback control in computer vision pipelines. Feedback control allows vision pipelines to perform adaptive parameter selection, iterative optimization and performance tuning. We show how our extension can describe feedback control in the vision pipelines of two state-of-the-art techniques.

1 Introduction

Recently, there has been a rapid growth of applications capable of generating vast amounts of images and videos. Examples of such applications include online image and video sharing services (e.g., Flickr[1] and ImageNet[2]), video surveillance systems [1–4], and satellite imagery [5–7]. An emerging direction for understanding and harnessing such big visual data is *stream processing*. Stream processing represents a category of methods that process infinite sequences of data, also called data streams. In this paper, we are interested in image and video streams which we refer to as *Vision Streams*. In order to process vision streams, researchers use stream processing concepts such as pipelines to construct online computer vision algorithms [8–14]. A question that then arise is how can we formally and efficiently describe online computer vision pipelines? In database community, this question has been answered for text stream processing by developing stream algebra frameworks [15–18]. A stream algebra defines a set of abstract algebraic operators with well defined semantics that process streams as operands. These operators are used to build mathematical expressions that declaratively construct stream processing pipelines. For example, Demers *et al.* [18] proposed an algebra to express queries on event streams. Chkodrov *et al.* [15] described

[1] Flickr: https://www.flickr.com/ (*last accessed on 7 September 2014*).
[2] ImageNet: http://www.image-net.org/ (*last accessed on 7 September 2014*).

an implementation of a stream algebra that extends relational algebra for data streams. There are several advantages for stream algebras. For example, they can provide formal methods to resolve pipeline bottlenecks, implement dynamic reconfiguration, apply incremental evaluation, define common optimization methods, etc.

Database stream algebras are designed for structured textual streams. So, they are inapplicable for vision streams with unstructured visual content. Despite such challange, there exist some software frameworks such as OpenVL [19] and GStreamer [20], that address the efficient implementation of vision pipelines. However, these frameworks do not define a stream algebra, and lack a formal definition of vision pipelines. Recently, Helala *et al.* [21] presented a stream algebra for computer vision pipelines. This algebra revises the previous database stream algebras, and provides an abstraction for formally expressing computer vision pipelines. The algebra contains operators for both data processing and flow rate control. Two online vision algorithms have been expressed in this algebra by [21]. However, these algorithms are only for feedforward pipelines. This limits the applicability of the algebra to other online computer vision systems that use feedback control to perform tasks such as parameter tuning [22–25], and iterative optimization [14]. These tasks are widely used in online vision algorithms. For example, Sherrah [23] presented an algorithm for continuous real-time parameter tuning of a people tracking surveillance system. Supancic *et al.* [25] explored parameter tuning for long term tracking. Iterative optimization was also studied by [14] to iteratively align Flickr's photo streams.

In this paper, we are studying feedback control loops in computer vision pipelines. Specifically, we extend the stream algebra of [21] to provide an algebraic description of feedback control; here we focus on parameter tuning, and iterative optimization. Our description of feedback control is formal and abstract, which makes it reusable by several online computer vision pipelines [22–25, 14]. The paper is organized as follows: Section 2 briefly reviews the algebra of [21], and discusses feedback control. Section 3 describes the feedback control of two state-of-the-art online computer vision algorithms. Then, we provide discussions in Section 4, and conclude the paper in Section 5.

2 Stream Algebra

The stream algebra in [21] contains three main parts: a common notation, a set of operators, and the formal semantics used to write pipeline expressions. This section gives a brief review of the algebra, and states the operators used in this paper. Finally, we discuss our algebraic extensions that provide an abstract and formal definition of feedback control in vision pipelines.

2.1 Notation

The algebra in [21] defines a data stream as an infinite sequence of data chunks with two function $\lambda x : x \rightarrow s$, and $\leftarrow s$, to write to and read from a stream

s, respectively. The algebra indicates the set of all possible streams as $\mathbf{S}$. To indicate the type of the stream data, the notation $\mathbf{S}\langle T\rangle$ is used, where T signify the data type. A stream operator is defined as a mapping function $h : \mathbf{S}^m \to \mathbf{S}^n : S^1_{\text{in}}, \ldots, S^m_{\text{in}} \to S^1_{\text{out}}, \ldots, S^n_{\text{out}}$, that maps m input streams to n output streams. We can define an operator by only using the following constructs as defined by [21]:

• Shared states: **state** u -Indicates that u is a state for subsequent loops. • Concurrency: **loop** : *body of loop* -Iterates over the body forever. -Each loop runs in its own thread. -All loops of the *same* operator share the states. -If there are multiple concurrent loops, $\mathbf{loop}_j$ indicates the j-th concurrent session.	• Atomicity: `{` *statements* `}` -Executes the statements as an atomic operation. • Stream I/O: -$x \leftarrow s$ reads from a stream s, and saves the result in x. -$e \rightarrow s$ writes the expression e to stream s. • Attribute Access: -$x.y$ accesses attribute y defined as part of variable x.

2.2 Operators

An operator can have zero or more parameters. These parameters can be simple functions, or initial values. We start by discussing the data processing operators.

Map is an operator, that *synchronously* reads from k incoming streams, applies a user-defined function $f : X_1 \times X_2 \times \ldots X_k \to Y$ on the read values, and writes the computed values to an outgoing stream. The operator is parametrized by the user-defined function. $\textsc{map}(f) : \mathbf{S}\langle X_1\rangle \times \ldots \mathbf{S}\langle X_k\rangle \to \mathbf{S}\langle Y\rangle$

$$\mathbf{loop} : f(\leftarrow S^1_{\text{in}}, \ldots, \leftarrow S^k_{\text{in}}) \to S_{\text{out}}$$

Reduce is an operator that has an internal state $u : U$. The operator reads from an incoming stream and applies a user defined function $g : U \times X \to U \times Y$. This function takes the saved state and the read value. Then, it computes a new state, and an output value. The operator updates the internal state, and writes the computed value to the output stream. The operator is parametrized by the function g, and an initial state u_0. $\textsc{reduce}(g, u_0) : \mathbf{S}\langle X\rangle \to \mathbf{S}\langle Y\rangle$

$$\begin{aligned} \mathbf{state} \quad & u = u_0 \\ \mathbf{loop} : \; & u, y = g(u, \leftarrow S_{\text{in}}) \\ & y \rightarrow S_{\text{out}} \end{aligned}$$

Copy reads from an incoming stream, and synchronously duplicates the read value to all outgoing streams. **Copy** has no parameters. $\textsc{copy}() : \mathbf{S} \to \mathbf{S}^n$

$$\begin{aligned} \mathbf{loop} : \; & x \leftarrow S_{\text{in}} \\ & x \rightarrow S^i_{\text{out}} \quad \text{for all } i \leq n \end{aligned}$$

Filter has one incoming stream and two outgoing streams S^1_{out} and S^2_{out}. It applies a used-defined predicate $\theta : X \rightarrow$ boolean, on the incoming values. It then writes values with θ true to S^1_{out}, and values with θ false to S^2_{out}. FILTER(θ) : $\mathbf{S} \rightarrow \mathbf{S}^2$

$$\begin{aligned}\textbf{loop} : x \leftarrow S_{\text{in}} \\ \textbf{if } \theta(x) \textbf{ then } x \rightarrow S^1_{\text{out}} \textbf{else } x \rightarrow S^2_{\text{out}}\end{aligned}$$

Ground ends an incoming stream. GROUND : $\mathbf{S} \rightarrow \emptyset$

$$\textbf{loop} : \leftarrow S_{\text{in}}$$

Now, we will discuss the rate controlling operators:

Latch has one incoming stream S_{in} and two outgoing streams S^1_{out} and S^2_{out}. It reads from S_{in}, synchronously writes to S^2_{out}, and asynchronously writes to S^1_{out}. It performs the asynchronous write by saving the most-recent incoming reading, and writing it to S^1_{out}, whenever it is possible. So S_{in} and S^1_{out} have different data rates. LATCH() : $\mathbf{S} \rightarrow \mathbf{S}^2$.

$$\begin{aligned}&\textbf{state } u = \textbf{nil} \\ &\textbf{loop}_1 : x \leftarrow S_{\text{in}} \qquad\qquad \textbf{loop}_2 : \{u \rightarrow S^1_{\text{out}}\} \\ &\qquad \{u = x \,; x \rightarrow S^2_{\text{out}}\}\end{aligned}$$

Cut is similar to **Latch**, but it writes the incoming values only once to the asynchronous output stream. For the extra writes, in case S^1_{out} has a higher data rate, **nil** is used. CUT() : $\mathbf{S} \rightarrow \mathbf{S}^2$

$$\begin{aligned}&\textbf{state } u = \textbf{nil} \\ &\textbf{loop} : x \leftarrow S_{\text{in}} \qquad\qquad \textbf{loop} : \{y = u \,; u = \textbf{nil}\} \\ &\qquad \{u = x \,; x \rightarrow S^2_{\text{out}}\} \qquad\qquad y \rightarrow S^1_{\text{out}}\end{aligned}$$

Mult has k incoming streams S^k_{in}, and one output stream S_{out}. The operator reads one value at a time from each incoming stream, forms a vector $(x_1, ..., x_k)$, and synchronously writes this vector to the outgoing stream. MULT() : $\mathbf{S}^k \rightarrow \mathbf{S}$

$$\textbf{loop} : \begin{bmatrix} \leftarrow S^1_{\text{in}} \\ ... \\ \leftarrow S^k_{\text{in}} \end{bmatrix} \rightarrow S_{\text{out}}$$

Left-Mult is similar to **Mult**; however, it has only two incoming streams S^1_{in} and S^2_{in}. It latches on S^2_{in} to make the outgoing data rate depends only on the rate of S^1_{in}, and independent of S^2_{in}. LEFT-MULT : $\mathbf{S}^2 \rightarrow \mathbf{S}$

$$\begin{aligned}S^1,\ S^2 &= \text{LATCH}(S^2_{\text{in}})\ ;\ \text{GROUND}(S^2) \\ \textbf{loop} &: \begin{bmatrix} \leftarrow S^1_{\text{in}} \\ \leftarrow S^1 \end{bmatrix} \rightarrow S_{\text{out}}\end{aligned}$$

Add asynchronously merges together k incoming streams S_{in}^k into one outgoing stream. ADD : $\mathbf{S}^k \rightarrow \mathbf{S}$.

$$\begin{aligned} &\text{for } j \leq k \\ &\quad \mathbf{loop}_j : x \leftarrow S_{\text{in}}^j \\ &\qquad\qquad\ x \rightarrow S_{\text{out}} \end{aligned}$$

2.3 Feedback Control

Feedback control is an essential task in several online computer vision algorithms [14, 22–25] that perform parameter tuning, or iterative optimization. These algorithms evaluate the current output results to enhance the future outputs. In this section, we extend the stream algebra of [21] by providing a formal and abstract description of feedback control in computer vision pipelines.

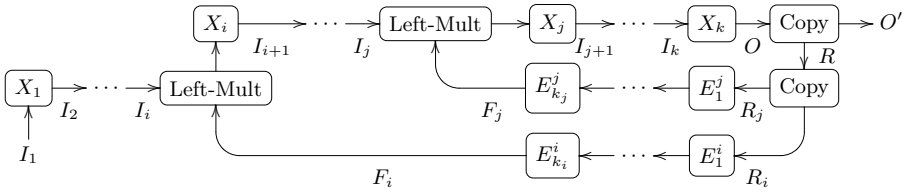

Fig. 1. An example of multi-point feedback control. Arrows show stream flow directions. Letters on arrows represent stream names. I_1, and O' indicate input and output streams, respectively.

We assume an input stream I_1, a vision pipeline with a sequence of operators $X_1, ..., X_k$, and an output stream O. Each operator X_j has an input stream I_j, and an output stream I_{j+1}. In order to define feedback control for operator X_j, we assume a return stream R_j, a sequence of evaluation operators $E_1^j, ..., E_{k_j}^j$, and a feedback stream F_j. Given these assumptions; we describe the feedback loop in a vision pipeline as in figure 1. We discuss two types of feedback control, single-point and multi-point. In single-point, we only control one operator X_j. So, the pipeline has one return stream R, and one feedback stream F. In order to obtain the return stream R, we can apply either the COPY, FILTER, or CUT operator on the output stream O. This is represented by the equation,

$$R, O' \triangleq \textsc{Copy}()(O). \tag{1}$$

O' now represents the output. We can next apply a sequence of evaluation operators $E_1, ..., E_n$ on R to compute the feedback stream F. If the feedback loop performs parameter tuning, then F represents a stream of new parameters. We can then apply a LEFT-MULT operator to attach the new parameters to the

input stream I_j, which updates the internal parameters of the pipeline operator X_j. This is represented by the equation,

$$I'_j \triangleq \textsc{Left-Mult}()(I_j, F). \tag{2}$$

I'_j in this case represents the input to operator X_j. Note that, the feedback loop that we just defined is synchronized with the original vision pipeline. If we replace Copy in equation 1 by Cut, then we have an asynchronous feedback loop that evaluates samples from the output stream O. Furthermore, in case of iterative optimization that reprocesses the output, we can replace Left-Mult in equation 2 by an Add operator. In this case, the stream I'_j will have interleaved elements from the streams I_j and F.

In multi-point feedback control, we can control more than one operator. This is performed by applying Copy on the return stream R several times to get m duplicate streams, which we use to define the feedback loops of m operators. Figure 1 shows multi-point feedback for two operators X_i and X_j. Each operator has a distinct set of evaluation operators to construct its feedback stream.

3 Feedback Control in Computer Vision Pipelines

There is a large interest in developing online computer vision algorithms that use feedback control to perform parameter tuning or iterative optimization tasks. These tasks allow an online algorithm to adapt itself continuously to different scene contexts, or iteratively improve output results over time. In this section, we discuss two state-of-the-art algorithms [24, 14] that process vision streams and apply feedback control to perform parameter tuning and iterative optimization. Without loss of generality, we will discuss how we can effectively express the feedback control of these algorithms using our algebraic extensions.

3.1 Online Adaptation of Tracking Parameters

Online tracking of moving objects is one of the fundamental problems in computer vision, and several trackers have been proposed. However, if the input video stream has an unknown scene, it becomes difficult to select a tracking algorithm. Chau *et al.* [24] proposed a method for online parameter tuning that can adapt a tracking algorithm to scene changes. This method works in two phases: an offline training phase, and an online control phase.

In the offline training phase, the algorithm takes as input, a set of training videos together with annotated moving objects, annotated trajectories, and a tracking algorithm with parameters. The method tracks people as the moving objects using the appearance based tracking algorithm of [26], which is controlled by six parameters. For each training video, the algorithm extracts context features from every frame. The context features are a vector of six elements that describes the density of moving objects, their occlusion level, and appearance characteristics. The algorithm then segments the video into consecutive clips, based on similarity of the context feature vectors. For each clip, the

algorithm performs parameter optimization to select the best parameter values of the given tracking algorithm. Then, a clustering step is performed to cluster contexts from all training videos. Afterwards, the best tracking parameters are defined for each cluster. Finally, the context clusters together with their best parameters are stored in a database D.

In the online stage, the algorithm takes as input, a video stream $V = \{V_i | i = 0, 1, 2...\}$. For every frame $V_i \in V$, people is detected using the HOG-based detection algorithm in [27]. This generates the objects stream $B = \{B_i | i = 0, 1, 2...\}$, where every B_i represents a list of detected objects (people) in frame V_i. The algorithm then records the detected objects from every frame in a temporal window of interval $\triangle t_1$. After that, the method attaches with every frame $V_i \in V$, the recent list of recorded objects up to frame V_i. This generates the stream $U = \{U_i | i = 0, 1, 2, ...\}$, where $U_i = (V_i, B_i)$. The method also defines the parameters feedback stream $F = \{F_i | i = 0, 1, 2, ...\}$, where every F_i is a vector of tracking parameters. We will see later how this stream is generated. Every F_i is attached to U_i to generate the tracker input stream $Q = \{Q_i | i = 0, 1, 2, ...\}$, where $Q_i = (V_i, B_i, F_i)$. The tracking algorithm takes the stream Q as input and updates the tracking parameters using $F_i \in Q_i$. It also generates a list of trajectories T_i for every frame. The tracker attaches the trajectories T_i to $(V_i, B_i) \subset Q_i$, and constructs the output trajectories stream $J = \{J_i | i = 0, 1, 2, ...\}$, where $J_i = (V_i, B_i, T_i)$. The algorithm in [24], then defines a feedback control loop that takes the output stream J and records the video frames $V_i \in J_i$ in a temporal window of interval $\triangle t_2$. This defines a clip stream $C = \{C_i | i = 0, 1, 2, ...\}$. This stream is used with J to define the loopback stream $H = \{H_i | i = 0, 1, 2, ...\}$, where $H_i = (V_i, B_i, T_i, C_i)$. Now, for every H_i, the algorithm uses $(V_i, B_i, T_i) \subset H_i$ to calculate two error scores: An object interaction score s_1 to calculate the overlap of objects, and a tracking error score s_2 to measure the tracking quality. Given two thresholds Th_1 and Th_2, if $s_1 > Th_1$ and $s_2 > Th_2$, the algorithm declares an error and retrieves the best tracking parameters suitable to the context defined by the clip $C_i \in H_i$, from the database D. Otherwise, we continue using the current parameters. This generates the feedback stream F that was used previously together with the stream U to define the tracker input stream Q. This illustrates the feedback control loop of [24].

Now, we will describe the online control stage of [24] in the algebra of [21], and use our algebraic extensions to define the feedback control loop. We start by defining the following data types,

$$\texttt{Frame} : \texttt{2DImage}; \quad \texttt{Video} : \mathbf{S}\langle\texttt{Frame}\rangle; \quad \texttt{Clip} : \textsc{List}\langle\texttt{Frame}\rangle$$
$$\texttt{Histogram} : \textsc{List}\langle\mathbb{R}\rangle; \quad \texttt{Object} : \mathbb{R}^8 \times \texttt{Histogram}; \quad \texttt{Params} : \mathbb{R}^6$$
$$\texttt{FrameInfo} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle; \quad \texttt{Trajectory} : \textsc{List}\langle\mathbb{R}^2\rangle;$$
$$\texttt{TrackInput} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle \times \texttt{Params}$$
$$\texttt{TrackInfo} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle \times \textsc{List}\langle\texttt{Trajectory}\rangle$$
$$\texttt{LoopBack} : \texttt{Frame} \times \textsc{List}\langle\texttt{Object}\rangle \times \textsc{List}\langle\texttt{Trajectory}\rangle \times \texttt{Clip}$$

where a `Frame` is a 2D image, a `Video` is a stream of frames, a `Clip` is a list of frames, and a `Histogram` is a vector of values. An `Object` is a pair (a, b), where b : `Histogram`, and $a : \mathbb{R}^8$ is a vector that represents the following object features: 2D shape ratio, 2D area, color covariance in RGB, and dominant color in RGB. `Params` is a vector that represents the 6 parameters of the tracking algorithm [26]. `FrameInfo` is a pair (v, w), where v : `Frame` and w : List $\langle$`Object`$\rangle$. A `TrackInput` is a 3 elements vector (v, w, p), where p : `Params`. A `Trajectory` is a list of 2D points. `TrackInfo` is a 3 elements vector (v, w, e), where e : List $\langle$`Trajectory`$\rangle$. Finally a `LoopBack` is a 4 elements vector (v, w, e, c), where c : `Clip`.

Given an input video stream $V \in$ `Video`, we copy V into two identical streams V_1, V_2 using a Copy operator,

$$V_1, V_2 \triangleq \textsc{Copy}()(V). \tag{3}$$

We will now process V_1 and return later to discuss the use of V_2. We define a function f_1 : `Frame` $\rightarrow$ List $\langle$`Object`$\rangle$ that detects objects from every frame $V_i \in V_1$ using the HOG-based detection algorithm of [27]. This function can be used with a Map operator to define the objects stream $B : \mathbf{S}\langle \textsc{List}\langle$`Object`$\rangle\rangle$,

$$B \triangleq \textsc{Map}(f_1)(V_1). \tag{4}$$

We then define the function,

$$g_1 : \textsc{List}\langle\texttt{Object}\rangle \times \textsc{List}\langle\texttt{Object}\rangle \rightarrow \textsc{List}\langle\texttt{Object}\rangle \times \textsc{List}\langle\texttt{Object}\rangle$$

$$\begin{aligned} g_1(u, x) = \{ & \text{ for all } \ z \in u \\ & \quad \text{if } (\text{now}() - \text{arrival-time}(z) \geq \triangle t_1) \text{ then} \\ & \quad\quad u = u \ominus z \quad //\text{remove } z \text{ from } u \\ & \quad u = u \oplus x \quad //\text{append } x \text{ to } u \\ & \quad \text{return}(u, u) \quad \} \end{aligned}$$

This function maintains a window u of the most recent objects, in a time interval $\triangle t_1$. The function starts by deleting old objects. Then, it adds the new objects x to u, and returns the updated window as output. We can use the g_1 function together with a Reduce operator to generate the objects summary stream $M : S\langle \textsc{List}\langle$`Object`$\rangle\rangle$,

$$M \triangleq \textsc{Reduce}(g_1, \text{Empty-List})(B). \tag{5}$$

Now, we will go back and use the stream V_2. Note that this stream is a copy of the input video stream V. We synchronize the stream M with the stream V_2 using a Mult operator to generate the stream $U : S\langle$`FrameInfo`$\rangle$,

$$U \triangleq \textsc{Mult}()(V_2, M). \tag{6}$$

At this step, we define the feedback stream $F : S\langle$`Params`$\rangle$. We will show later how we generate this stream when we discuss feedback control. We synchronize

the stream F with the stream U using a LEFT-MULT operator to generate the stream $Q : S\langle\texttt{TrackInput}\rangle$,

$$Q \triangleq \textsc{Left-Mult}()(U, F). \tag{7}$$

We then define the tracking function $f_2 : \texttt{TrackInput} \rightarrow \texttt{TrackInfo}$. This function takes the vector $(v, w, p) : \texttt{TrackInput}$, and applies the tracking algorithm on v and w using the given parameters p. The function then outputs the object trajectories which we attach to the pair (v, w), to generate the output $y : \texttt{TrackInfo}$. We can use this function together with a MAP operator to process the Q stream and generate the trajectories stream $J : S\langle\texttt{TrackInfo}\rangle$,

$$J \triangleq \textsc{Map}(f_2)(Q). \tag{8}$$

Now, we discuss how we use our algebraic description to express the feedback control of [24]. We start by evaluating the equation, $R, J' \triangleq \textsc{Copy}()(J)$, to copy J into the output stream J' and the return stream R. Then, we define the function,

$g_2 : \texttt{Clip} \times \texttt{TrackInfo} \rightarrow \texttt{Clip} \times \texttt{LoopBack}$

$$\begin{aligned} g_2(u, x) = \{ & \text{ for all } \ z \in u \\ & \quad \text{if } (\text{now}() - \text{arrival-time}(z) \geq \triangle t_2) \text{ then} \\ & \quad\quad u = u \ominus z \quad //\text{remove } z \text{ from } u \\ & u = u \oplus x.v \quad //\text{append frame } x.v \text{ to } u \\ & y = x \oplus u \quad //\text{append clip } u \text{ to } x \\ & \text{return}(u, y) \quad \} \end{aligned}$$

The function g_2 maintains a clip u that has the most recent frames in a time interval $\triangle t_2$. It is similar to the function g_1, however g_2 appends the recorded clip to the input x, to form the output $y : \texttt{LoopBack}$. We use this function with a REDUCE operator to process the return stream R, and generate the loopback stream $H : S\langle\texttt{LoopBack}\rangle$,

$$H \triangleq \textsc{Reduce}(g_2, \text{Empty-List})(R). \tag{9}$$

The algorithm in [24] defines two scoring functions to calculate the tracking errors. These functions are $f_3 : \texttt{FrameInfo} \rightarrow \mathbb{R}$ for object interaction score, and $f_4 : \texttt{TrackInfo} \rightarrow \mathbb{R}$ for tracking error score. We use these functions to define another function,

$g_3 : \texttt{Params} \times \texttt{LoopBack} \rightarrow \texttt{Params} \times \texttt{Params}$

$$\begin{aligned} g_3(u, x) = \{ & \ s_1 = f_3(x.v, x.w) \\ & \ s_2 = f_4(x.v, x.w, x.e) \\ & \ \text{if } (s_1 > Th_1 \text{ and } s_2 > Th_2) \text{ then} \\ & \quad\quad p = \text{search-db}(x.c, D) \\ & \quad\quad \text{return}(p, p) \\ & \ \text{return}(u, u) \quad \} \end{aligned}$$

Note that this function starts by calculating the error scores of the current frame. If the scores s_1 and s_2 are larger than the given thresholds, we search the database D for the best parameters that meet the context of the most recent clip $x.c$, and output the new parameters. Otherwise, we return the old parameters u. This function can be used together with a REDUCE operator to process the loopback stream H, and generate the feedback stream $F : S\,\langle\texttt{Params}\rangle$,

$$F \triangleq \textsc{Reduce}(g_3, \text{Initial-Params})(H). \tag{10}$$

Remember that we used the stream F together with the stream U as input to the LEFT-MULT operator in equation 7 to define the tracker input stream Q. In this way, we use our definition of single-point feedback control to continuously update the tracker parameters.

3.2 Iterative Optimization for Aligning Photo Streams

Recently, there has been a large interest in the analysis of Web photo streams. This interest is driven by the existence of several photo hosting websites that contain huge amounts of personal photo collections. Kim *et al.* [14] proposed a recent algorithm to build common storylines from Flickr's photo streams. Their algorithm takes as input, a set of n photo streams $I = \{I_k | k = 1, 2, 3..n\}$ from different Flickr users that share a common user activity. Each photo in a stream I_k stores its capture time, a spatial pyramid histogram as a visual descriptor, and a set of foreground regions that is initially empty. Every stream I_k is divided into a sequence of photo blocks $B_k = \{B_{\{k,i\}} | i = 0, 1, 2...\}$, where each block $B_{\{k,i\}}$ represents the photos taken by a user over a certain period of time $\triangle t_1$ (for example, a day). Each block also stores the earliest and latest capture times of its photos. The algorithm then iterates between two tasks, an alignment task, and a cosegmentation task.

In the alignment task, the algorithm starts by reading one block from each stream, and constructs the block-list stream $L = \{L_i | i = 0, 1, 2...\}$. Every $L_i \in L$ is a list of blocks. Then, the algorithm selects from every list L_i, a set of blocks E_i that overlap in a timeline by at least a period $\triangle t_2$ (for example, an hour). We also attach to E_i, an iteration number N, that is initially zero. This generates the overlapped blocks stream $Q = \{Q_i | i = 0, 1, 2...\}$, where $Q_i = (E_i, N_i)$. The algorithm defines a feedback stream F which is of the same type as Q. We will see later how this stream is constructed. We add the elements of the two streams Q and F together to construct the interleaved stream $P = \{P_j | j = 0, 1, 2...\}$, where $P_j = (E_j, N_j)$. Note that the stream P contains interleaved elements from the two streams Q and H. For each block b of $E_j \in P_j$, the algorithm calculates the similarity of b to other blocks in E_j. This is performed for two blocks $(b_1, b_2) \subset E_j$ by first finding for each photo $x \in b_1$, its best visually similar neighbor $y \in b_2$. This similarity is calculated using a distance function $f_4 : \text{Photo} \times \text{Photo} \to \mathbb{R}$. If both x and y have foreground regions defined, f_4 returns the distance between the histograms of these regions. Otherwise, f_4 returns the distance between the spatial pyramid histograms of photos. The algorithm also defines another

distance function $f_5 : \text{Time} \times \text{Time} \rightarrow \mathbb{R}$, that calculates the difference between the capture times of two photos. The algorithm then measures the similarity between two blocks b_1 and b_2 using an energy function $f_6 : \text{Block} \times \text{Block} \rightarrow \mathbb{R}$, that sums the similarity defined by f_4 and f_5 along the correspondent photos between b_1 and b_2. The method then maps every list of blocks $E_j \in P_j$ into a graph $G_j : \text{Graph}\,\langle \text{Block}, \text{Block} \times \text{Block} \rangle$. This graph has the blocks of E_j as vertices. An edge exists between two blocks, if they are best similar to each other, in other words, they have the minimal distance to each other according to f_6. This generates the blocks-graph stream $Z = \{Z_j | j = 0, 1, 2...\}$, where $Z_j = (G_j, N_j)$. This stream defines the output of the alignment step.

The cosegmentation step takes as input, the blocks-graph stream Z. This step maps every graph $G_j \in Z_j$ into a new graph $Y_j : \text{Graph}\,\langle \text{Photo}, \text{Photo} \times \text{Photo} \rangle$. This is performed by first collecting all photos from the blocks of G_j. These photos define the vertices of Y_j. Then, the algorithm adds an edge between two photos (x, y) if they are a correspondent pair of two different blocks. In addition, for every photo x in a block b, the method adds edges to its k nearest neighbors from the same block b. We append the graph Y_j to Z_j to construct the photos-graph stream $M = \{M_j | j = 0, 1, 2...\}$, where $M_j = (G_j, N_j, Y_j)$. We also increment N_j by 1. Afterwards, the algorithm in [28] defines for each photo in the vertices of $Y_j \in M_j$, a set of m foreground regions. This is performed by applying the cosegmentation technique of [28] and belief propagation between each photo and its neighbors in Y_j. If a photo already have foreground regions defined, then they are enhanced.

The algorithm of [14] iterates between the alignment task and the cosegmentation task. This is achieved by defining a feedback loop that first filters the photos-graph stream M based on the iteration number $N_j \in M_j$. If $N_j \geq N_{stop}$ then M_j is sent to the output, Otherwise, it is sent to the feedback loop. N_{stop} defines the maximum number of iterations. This defines two streams M' and R with similar type to stream M, where M' is the output stream, and R is the return stream. The stream R is then mapped to the feedback stream $F = \{F_l | l = 0, 1, 2...\}$, where $F_l = (E_l, N_l)$. E_l is the list of blocks in the vertices of $G_l \in R_l$ and N_l is the iteration number. Remember that the feedback stream F was added to the stream Q to define the interleaved stream P in the alignment step. Note also that the block photos in F have foreground regions defined. These regions enhance the matching of the feedback blocks in the alignment step. Consequently, This improves the output of the cosegmentation step, which closes the feedback loop defined by [14] for iterative optimization.

Now, we will express the iterative optimization of [24] using our algebraic extensions for feedback control. We define the following data types,

$$\texttt{Shape} : \textsc{List}\,\langle \mathbb{R}^2 \rangle ; \quad \texttt{Region} : \texttt{Shape} \times \texttt{Histogram}$$
$$\texttt{Photo} : \texttt{2DImage} \times \texttt{Time} \times \texttt{Histogram} \times \textsc{List}\,\langle \texttt{Region} \rangle \times \mathbb{R}$$
$$\texttt{Block} : \textsc{List}\,\langle \texttt{Photo} \rangle \times \texttt{Time}^2; \quad \texttt{BlocksInfo} : \textsc{List}\,\langle \texttt{Block} \rangle \times \mathbb{R};$$
$$\texttt{BlocksGraphInfo} : \textsc{Graph}\,\langle \texttt{Block}, \texttt{Block} \times \texttt{Block} \rangle \times \mathbb{R}$$

$\texttt{PhotosGraphInfo} : \texttt{BlocksGraphInfo} \times \textsc{Graph}\,\langle \texttt{Photo}, \texttt{Photo} \times \texttt{Photo} \rangle$

Where, a `Shape` is a list of 2D points. A `Region` is a 2D vector (s, h), where s : `Shape`, and h : `Histogram` is the region descriptor. We reuse the definition of `Histogram` from the previous section. A `Photo` is a vector of five variables (a, t, h, r, nn), where a : `2DImage`, $t : Time$ is the capture time, h : `Histogram` is a visual descriptor, $r : \textsc{List}\,\langle \texttt{Region} \rangle$ is a list of foreground regions (initially empty), and nn is the index of the nearest neighbor photo. A `Block` is a 3D vector (b, t_1, t_2), where $b : \textsc{List}\,\langle \texttt{Photo} \rangle$, $t_1 : Time$ is the earliest capture time of photos in b, and t_2 is the latest capture time. A `BlocksInfo` is a vector (w, itr), where $w : \textsc{List}\,\langle \texttt{Block} \rangle$, and $itr : \mathbb{R}$ indicates the iteration number. A `BlocksGraphInfo` is a vector (c_1, itr), where c_1 is a graph on a set of blocks. Finally, `PhotosGraphInfo` is a 2D vector (q, c_2), where q : `BlocksGraphInfo`, and c_2 is a graph on a set of photos.

For simplicity, we will consider that we have three input photo streams $I = \{I_k | k = 1..3\}$. We now define the following function,

$g_4 : \texttt{Block} \times \texttt{Photo} \rightarrow \texttt{Block} \times \texttt{Block}$

$$
\begin{aligned}
g_4(u, x) = \{\ & \text{if duration}(u) \geq \triangle t_1 \text{ then} \\
& \quad u' = \emptyset; y = u \\
& \text{else} \\
& \quad u' = u \oplus x \quad \text{//append } x \text{ to Block } u \\
& \quad y = \emptyset \\
& \text{return}(u', y) \quad \}
\end{aligned}
$$

This function keeps appending the incoming photos to a block u' and setting the output y to an empty block. When the duration of the block $\triangle t = u'.t_2 - u'.t_1$ is larger than a certain interval $\triangle t_1$ (A day in [14]), the function copies the block u' to the output y and resets u' back to an empty block. We can use this function together with a Reduce and Filter operators to process every stream $I_k \in I$ and generate a corresponding stream $B_k : S\,\langle \texttt{Block} \rangle$,

$$B_k, E_k \triangleq \textsc{Filter}(\lambda\, x :\ |x| \neq 0) \circ \textsc{Reduce}(g_4, \text{Empty-List})(I_k). \tag{11}$$

$$\textsc{Ground}()(E_k). \tag{12}$$

Where the $\circ$ operator is defined by [21] as a composition operator, that supplies the output stream from the right operand as an input stream to the left operand. The Filter operator removes the empty blocks from the output stream of the Reduce operator. These empty blocks define the stream E_k which is ignored by a Ground operator. We can now synchronize the three streams B_1, B_2 and B_3 using a Mult operator to construct the block-list stream $L : S\,\langle \textsc{List}\,\langle \texttt{Block} \rangle \rangle$,

$$L \triangleq \textsc{Mult}()(B_1, B_2, B_3). \tag{13}$$

We then define the function $f_7 : \textsc{List}\,\langle\texttt{Block}\rangle \rightarrow \texttt{BlocksInfo}$. This function selects from every list $L_i \in L$, blocks that overlap in a timeline by at least a period $\triangle t_2$ (An hour in [14]). The function also attaches to the list of selected blocks, an iteration number, which is initially zero. We can use f_7 together with a Map operator to define the stream $Q : S\,\langle\texttt{BlocksInfo}\rangle$,

$$Q \triangleq \textsc{Map}(f_7)(L). \tag{14}$$

At this step, we define the feedback stream $F : S\,\langle\texttt{BlocksInfo}\rangle$. We will show later how we generate this stream when we discuss the feedback loop. We add the two streams Q and F using an Add operator to generate the stream $P : S\,\langle\textsc{List}\,\langle\texttt{Block}\rangle\rangle$,

$$P \triangleq \textsc{Add}()(Q, F). \tag{15}$$

Now, we define two functions f_8 : $\texttt{BlocksInfo} \rightarrow \texttt{BlocksGraphInfo}$, and $f_9 : \texttt{BlocksGraphInfo} \rightarrow \texttt{PhotosGraphInfo}$. The function f_8 maps a blocks list in BlocksInfo into a graph of blocks, where an edge exists between two blocks if they are nearest neighbors according to the function f_6. Note that f_6 gives better similarity distance if block photos have good foreground regions. The function f_9, on the other hand, maps a graph of blocks into a graph of photos, and increments the iteration number by 1. Cosegmentation is applied between each photo and its neighbors in the photos graph to define foreground regions or enhance existing ones. We can use the two functions f_8 and f_9 together with two Map operators to define the photos-graph stream $M : S\,\langle\texttt{PhotosGraphInfo}\rangle$,

$$M \triangleq \textsc{Map}(f_9) \circ \textsc{Map}(f_8)(P). \tag{16}$$

Now, we will discuss the feedback control of [14] to perform iterative optimization using our algebraic extensions. We start by applying a Filter operator on the stream M to define the output stream M' and the return stream R,

$$M', R \triangleq \textsc{Filter}(\lambda\, x :\ x.q.itr \geq N_{stop})(M). \tag{17}$$

We access the attribute $q : \texttt{BlocksGraphInfo}$ from each element $M_j \in M$. Then, we test if the iteration number of q reached the maximum number of iterations. If the test is true M_j is sent to M', if not, it is sent to R. We then define a function $f_{10} : \texttt{PhotosGraphInfo} \rightarrow \texttt{BlocksInfo}$ which maps every element in R into an element of type BlocksInfo. This is performed by converting the vertices of the blocks-graph attached to PhotosGraphInfo into a list of blocks, and copying the current iteration number. The function f_{10} together with a Map operator define the feedback stream $F : S\,\langle\texttt{BlocksInfo}\rangle$,

$$F \triangleq \textsc{Map}(f_{10})(M_2). \tag{18}$$

Remember that we used the stream F together with the stream Q as input to the Add operator in equation 15 to define the stream P. This shows how our definition of single-point feedback control describes iterative optimization in [14].

4 Discussions

The examples discussed in the previous section, demonstrate that we can effectively express feedback control in online computer vision algorithms using our abstract algebraic description presented in section 2.3. In each discussed algorithm, we first express its vision pipeline by a set of algebraic equations based on the operators defined by [21]. Then, we use our abstract definition of single-point feedback control to describe both the parameter tuning and the iterative optimization tasks. For example, the algebraic description of the online tracking example [24] shows a flexible integration of our single-point feedback to tune tracking parameters. Additionally, We can easily extend this single-point feedback control to multi-point feedback control (Figure 1). We can use this, for example, to control both the parameters of tracker and the parameters of the HOG-based people detector algorithm [27] used by [24] to detect moving objects. So, our multi-point description can scale up feedback to control several operators of a vision pipeline. Furthermore, if the vision pipeline has several output streams, we may extend our multi-point feedback to process multiple return streams. In this case, we can use rate control operators such as Cut and Latch to replace Copy in figure 1, and define asynchronous feedback control that does not affect the flow rate of the feedforward pipeline. This also motivates that we may perform multi-point feedback between different computer vision pipelines, that work at separate data rates, without affecting one another.

The discussed iterative optimization example [14] suggests that our formal feedback description may be useful for other tasks such as adaptive learning and incremental evaluation. In addition, our feedback description may be used for implementing blocking resolution in vision pipelines. For example, we can make every pipeline operator attach its estimated runtime to the output stream. Then, these runtimes can be collectively monitored by feedback control to send back appropriate actions to the operators. So, our formal feedback description opens new directions for tasks such as real-time debugging, performance monitoring and bottleneck identification.

5 Conclusion

This paper develops abstract and formal methods for describing and implementing feedback control in computer vision pipelines—online vision algorithms that process images and videos (vision streams). These formal methods build upon an existing stream algebra to flexibly integrate feedback control to vision pipelines. We have demonstrated our formal methods for two state-of-the-art online computer vision algorithms that implement feedback control for parameter tuning and iterative optimization. We also discussed how our methods can scale up feedback to control different stream operators of the vision pipeline. Our work opens up new research directions to study real-time debugging, dynamic reconfiguration, bottleneck identification, adaptive learning, and performance tuning of vision pipelines.

References

1. Zhao, B., Fei-Fei, L., Xing, E.: Online detection of unusual events in videos via dynamic sparse coding. In: CVPR, Colorado Springs (2011) 3313–3320
2. Helala, M., Pu, K., Qureshi, F.: Road boundary detection in challenging scenarios. In: AVSS. (2012) 428–433
3. Meghdadi, A., Irani, P.: Interactive exploration of surveillance video through action shot summarization and trajectory visualization. IEEE Trans. on Visualization and Computer Graphics **19** (2013) 2119–2128
4. Yenikaya, S., Yenikaya, G., Düven, E.: Keeping the vehicle on the road: A survey on on-road lane detection systems. ACM Comput. Surv. **46** (2013) 2:1–2:43
5. Ozcanli, O., Dong, Y., Mundy, J., Webb, H., Hammoud, R., Victor, T.: Automatic geo-location correction of satellite imagery. In: IEEE CVPR Workshops. (2014)
6. Wischounig-Strucl, D., Quartisch, M., Rinner, B.: Prioritized data transmission in airborne camera networks for wide area surveillance and image mosaicking. In: IEEE CVPR Workshops. (2011) 17–24
7. Yuping, L., Medioni, G.: Map-enhanced uav image sequence registration and synchronization of multiple image sequences. In: IEEE CVPR, Minneapolis, Minnesota, USA (2007) 1–7
8. Ryoo, M.S.: Human activity prediction: Early recognition of ongoing activities from streaming videos. In: ICCV, Barcelona, Spain (2011) 1036–1043
9. Lu, C., Shi, J., Jia, J.: Online robust dictionary learning. In: IEEE CVPR. (2013) 415–422
10. Xuand, C., Xiong, C., Corso, J.: Streaming hierarchical video segmentation. In: ECCV. Volume VI. (2012) 626–639
11. Loy, C., Hospedales, T., Xiang, T., Gong, S.: Stream-based joint exploration-exploitation active learning. In: CVPR. (2012) 1560–1567
12. Harbi, N., Gotoh, Y.: Spatio-temporal human body segmentation from video stream. In: Computer Analysis of Images and Patterns. Volume 8047. Springer (2013) 78–85
13. Yang, J., Luo, J., Yu, J., Huang, T.: Photo stream alignment and summarization for collaborative photo collection and sharing. IEEE Trans. on Multimedia **14** (2012) 1642–1651
14. Kim, G., Xing, E.: Jointly aligning and segmenting multiple web photo streams for the inference of collective photo storylines. In: CVPR. (2013) 620–627
15. Chkodrov, G., Ringseth, P., Tarnavski, T., Shen, A., Barga, R., Goldstein, J.: Implementation of stream algebra over class instances, Google patents (2013)
16. Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science **258** (2001) 99 – 129
17. Carlson, J., Lisper, B.: An event detection algebra for reactive systems. In: Proceedings of the 4th ACM International Conference on Embedded Software. (2004) 147–154
18. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: A general algebra and implementation for monitoring event streams. Technical report, Cornell University (2005)
19. Shen, C., Little, J., Fels, S.: Towards OpenVL: Improving real-time performance of computer vision applications. In: Embedded Computer Vision. Advances in Pattern Recognition. Springer London (2009) 195–216
20. GStreamer. http://gstreamer.freedesktop.org (2014) Accessed: 2014-01-26.
21. Helala, M.A., Pu, K.Q., Qureshi, F.Z.: A stream algebra for computer vision pipelines. In: IEEE CVPR Workshops. (2014)
22. Kisilev, P., Freedman, D.: Parameter tuning by pairwise preferences. In: BMVC. (2010)
23. Sherrah, J.: Learning to adapt: A method for automatic tuning of algorithm parameters. In: ACIVS. (2010) 414–425

24. Chau, D., Badie, J., Bremond, F., Thonnat, M.: Online tracking parameter adaptation based on evaluation. In: IEEE International Conference on AVSS. (2013) 189–194
25. III, J.S., Ramanan, D.: Self-paced learning for long-term tracking. In: CVPR, Washington, DC, USA (2013) 2379–2386
26. Chau, D., Bremond, F., Thonnat, M.: A multi-feature tracking algorithm enabling adaptation to context variations. In: ICDP 2011. (2011) 1–6
27. Corvee, E., Bremond, F.: Body parts detection for people tracking using trees of histogram of oriented gradient descriptors. In: IEEE International Conference on AVSS. (2010) 469–475
28. Kim, G., Xing, E.: On multiple foreground cosegmentation. In: IEEE CVPR. (2012) 837–844