EPiC Series in Computing
Volume 46, 2017, Pages 1–13
LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Programming by Composing Filters

Jeffrey M. Fischer[1] and Rupak Majumdar[2]

[1] Data-Ken Research
jeff@data-ken.org
[2] Max Planck Institute for Software Systems (MPI-SWS)
Kaiserslautern and Saarbrücken, Germany
rupak@mpi-sws.org

Abstract

We present a formal model for event-processing pipelines. Event-processing pipelines appear in a large number of domains, from control of cyber-physical systems (CPS), to large scale data analysis, to Internet-of-things applications. These applications are characterized by stateful transformations of event streams, for example, for the purposes of sensing, computation, and actuation of inner control loops in CPS applications, and for data cleaning, analysis, training, and querying in data analytics applications.

Our formal model provides two abstractions: *streams* of data, and *stateful, probabilistic, filters*, which transform input streams to output streams probabilistically. Programs are compositions of filters. The filters are scheduled and run by an explicit, asynchronous, scheduler.

We provide a transition system semantics for such programs based on infinite-state Markov decision processes. We characterize when a program is *scheduler-independent*, that is, provides the same observable behavior under every scheduler, based on local commutativity.

1 Introduction

Event-based processing is at the core of many applications from many different domains, from sensing and control of cyber-physical systems to low-level high-throughput packet processing to large-scale data analysis. Consequently, many domain-specific languages and systems have been developed for writing event processors. In the domain of synchronous hardware systems, for example, synchronous languages such as Lustre provide an elegant declarative language that structures event processing as transformations on streams [6, 4, 13, 25]. In the domain of continuous control systems, languages such as Simulink and Stateflow provide a visual notation to structure the flow of (continuous) sensor data, control computations, and actuation signals. In packet processing, systems such as Click [19, 1] provide a modular way of composing individual packet transformers into asynchronous packet processing systems. In data analytics, systems such as Spark [2] or Linq [22] provide a functional API that can be composed to form complex data processing pipelines. Streaming databases provide languages to query and process streams of uncertain data [7, 9, 8, 18]. Superficially, these languages or language abstractions look quite different. But at their core, all these formalisms consist of a representation for *streams* and (often asynchronous) *transformations* of input streams into output streams in a *stateful* way. In addition, for many applications, it is important to explicitly model *uncertainties*, e.g., probabilistic nature of the underlying data or randomization in the processing.

In this paper, we present *filter language* (FL), a formal model for stateful data processing pipelines which captures the essence of stateful and asynchronous processing of data streams with explicit support for uncertainties. The core abstractions in FL are *streams* of (possibly continuous-valued) events, and *filters*, a trasducer with (potentially infinite) internal state mapping input streams of events to output streams. Filters can be connected through their input and output ports. Filters can be *probabilistic*, that is, the transition function can depend on the outcome of a probabilistic sample. A FL program is a set of filters interconnected into a graph. The execution of an FL program is asynchronous: a co-operative scheduler picks a filter to be executed atomically in each step.

We provide a transition system semantics for FL programs, based on infinite-state Markov decision processes. Our semantics captures the operational intuition that programmers have about processing sensor streams. In contrast to probabilistic languages such as PDL [20], our semantics involves actions chosen by the asynchronous scheduler: under a given scheduling policy (and an initial distribution), we recover the stochastic process semantics in these languages.

FL programs are non-deterministic: depending on the choice of schedulers, the outputs of the program can be different. A particular question of interest is when a program is *scheduler-independent*, that is, when the behavior of the program is independent of the choice of schedulers. We characterize a *confluence* theorem for FL programs: if each filter is (locally) commutative, the entire program is guaranteed to be confluent, that is, the distribution over the states on termination of the program is identical no matter which scheduler was used. Our result generalizes similar folklore results on stream processing (e.g., for mapreduce pipelines) to the probabilistic setting.

Our study of programming with filters is motivated by Internet-of-Things (IoT) applications, which combine sensing-computation-actuation of traditional control loops with data processing and distributed analytics. We have implemented FL as a Python API called ThingFlow.[1] Our design of ThingFlow ensures that there is a core abstraction that is simple enough to run on resource-constrained sensors while providing interfaces to many different systems (such as databases, storage, and machine learning pipelines). We have implemented a number of applications on top of ThingFlow, including home automation and control applications involving sensing, computation, learning, and control. Our experience with the FL abstraction indicates that it provides a simple abstraction and semantics for stream processing systems which is close to the intuition of the domain expert.

2 Informal Overview and Examples

We informally introduce FL through examples.

Filters. A basic FL component is called a *filter*. Filters implement state transducers: they accept (multiple) streams of inputs and produce (multiple) streams of outputs, and maintain internal state. State transitions can be probabilistic. Each filter has a set of input ports and a set of output ports, which determine the flow of events in the system, and transfer functions that determine how to process events coming into the input ports and what events to send along the output ports. Filters can maintain (potentially unbounded) internal state. For example, in

[1] Our implementation is available at `https://github.com/mpi-sws-rse/thingflow-python`.

control-related applications, the state can be real numbers derived from sensor measurements.[2] The transition function implemented by a filter can be probabilistic. The intuition is that each filter is intended to implement a focused task; complex behaviors are obtained by composing filters.

Programs and Scheduling. A FL program is a directed graph whose vertices are filters and whose edges, called *connections*, connect the output ports of filters to input ports of other filters. In general, the input or output port of a filter can be connected to output or input ports of multiple other filters. If an output port o of a filter A is connected to the input port i of a filter B, we say that B *subscribes* to A through the port mapping $o \mapsto i$.

Informally, a FL program processes sequences of events *asynchronously*. A FL program is scheduled by an application-level scheduler. At each step, the schedule picks a filter and an input port of the filter, and the filter processes the first event queued at its input port by changing its internal state and producing (potentially 0, 1, or multiple) events along its output ports. The events on the output ports are enqueued in the input ports of all downstream filters connected to those output ports. While events along each connection is FIFO-ordered, the sequence of execution is non-deterministic and depends on the order picked by the asynchronous scheduler. For example, if two events appear at two input ports of a filter, the order of execution of the two events is not determined a priori. "Free" input ports (not connected to an upstream filter) represent external inputs to the system (e.g., sensor streams) and "free" output ports represent external outputs.

We now describe some concrete example programs.

Example: Functional pipelines Data analysis applications such as Microsoft's Linq or Apache Spark provide a functional API to transform sequences. For example, these APIs provide functions to transform a sequence of values by applying a function on each element (`map`), to select a subsequence based on a predicate (`select`), and to perform an accummulation (`reduce`). As a concrete example, one can transform a stream of sensor values by extracting the value, filtering all readings above a threshold, and counting:

```
sensor.map(lambda x: x.val).select(lambda x: x > 20).reduce(lambda c, e: c+1, 0)
```

Here, we use `lambda x:` f`(x)` as concrete syntax for a lambda-abstraction.

One can view such pipelines as FL programs: `map`, `select`, and `reduce` are filters with exactly one input and one output ports. The first two are stateless. The last (`reduce`) maintains state through the accummulated value `acc`.

Example: Synchronous dataflow languages Synchronous dataflow languages such as Lustre [6] provide an elegant declarative programming model close to FL. A program in Lustre consists of variables representing streams of values and a set of equations defining filters. Programs have a synchronous interpretation: solving the equations synchronously gives the new values of the streams based on old values. For example, the example above could be expressed as the following equations:

```
X1 = sensor.val
X2 = X1 when X1 > 200
```

[2] Our use of the term *filter* is inspired by applications of FL programs in control and signal processing, where probabilistic transducers solve the "filtering problem" of state estimation from noisy measurements. This also avoids the common connotation that Mealy machines and state transducers are discrete-state, deterministic machines.

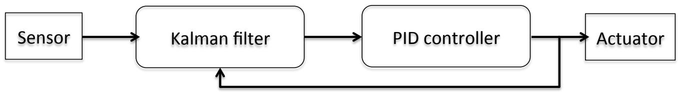

Figure 1: A PID control pipeline

```
Count = 0 -> pre(Count) + 1 when X2 > 200
```

Here, the variables `sensor`, `X1`, `X2`, `Count` represent streams of values, `when` is a sampling operation on streams, `pre(X)` is the stream `X` delayed by one time unit, and

```
X = 0 -> E
```

denotes the stream whose first element is `0` and thereafter the same as `E`.

There is no explicit notion of state in Lustre, because state can be maintained as just another stream of values (and use of the `pre` operation). To a first approximation, Lustre programs can be seen as an FL program with one filter (to ensure synchronous semantics). Code generation for Lustre programs using finite automata internally build this filter. The semantics of FL can be seen as a probabilistic extension to Lustre's semantics.

Example: PID control A simple implementation of control loops for continuous dynamical systems consists of a sequence of filters that read multiple sensor streams, transform the sensor streams into state estimates, computes a control action, and outputs the action to an actuator. Figure 1 shows an example FL pipeline. Both the Kalman filter and the PID controller maintains explicit internal state. The Kalman filtering algorithm maintains a "current belief," which is a probability distribution on the state, and updates the belief based on new sensor readings. The PID controller maintains the controller gains (and may switch between different modes based on the state). Specifically, the Kalman filter keeps track of the mean and covariance matrix of a Gaussian representing the distribution of states as a vector and matrix over reals. Note that the Kalman filter has two input ports: one from the sensor and one from the output of the PID controller. It receives a real-valued sensor measurement and a real-valued control output from the controller. It outputs the state estimate (the current mean) on its output port. Languages like Simulink and Stateflow are used to implement controllers and provide a visual framework to connect filters.

So far, the individual filters have been deterministic; while the theory behind Kalman filtering uses probabilities, the implementation of the filter itself is deterministic and manipulates symbolic representations of underlying probability distributions. However, more complex filtering algorithms, such as *particle filtering* [24], make explicit probabilistic decisions. A particle filter, like a Kalman filter, maintains as internal state a finite representation of the distribution of the system's state. However, unlike a Kalman filter, the distribution is not maintained symbolically but as a set of samples drawn from the distribution. When a new sensor input is read, the filter executes a randomized procedure to generate a new sample from the updated distribution.

Since probabilities are fundamental in many applications in data processing and control in an uncertain environment, in FL, we include probabilities in the model and its semantics. Thus, each filter defines a stochastic process over a potentially infinite state space. Accordingly, the semantics of FL programs are given by infinite-state Markov decision processes.

3 Transition System Semantics

We now give a semantics for FL as infinite-state Markov decision processes. For readability, we do it in two steps. First, we give a semantics for deterministic filters, whose transition functions are non-probabilistic. This semantics is a (non-probabilistic, infinite) labeled transition system. Then, using the intuition developed in the deterministic case, we define the probabilistic semantics. The probabilistic semantics is a Markov decision process over general infinite state spaces; to keep the semantics self-contained, we also state the necessary measure-theoretic preliminaries (see, e.g., [5]).

3.1 Deterministic Filters and FL Programs

Fix an alphabet Σ (not necessarily finite) of *events*. We write Σ^* for the set of finite strings over Σ. For strings $u, v \in \Sigma^*$, we write $u \cdot v$ for their concatenation. A *stream* is a (finite or infinite) sequence over Σ.

A *filter* $\mathcal{F} = (I, O, Q, \delta, \mathcal{O}, q_0)$ consists of a finite set of input ports I, a finite set of output ports O, a (not necessarily finite) set of internal states Q, a transition function $\delta : Q \times (I \times \Sigma) \to Q$, an output function $\mathcal{O} : Q \times (I \times \Sigma) \times O \to \Sigma^*$, and an initial state $q_0 \in Q$. A *run* of $\mathcal{F}$ on a sequence $(i_0, \sigma_0)(i_1, \sigma_1) \ldots \in (I \times \Sigma)^*$ is a sequence of states q_0, q_1, ... such that q_0 is the initial state of $\mathcal{F}$, and for each $j \geq 0$, we have $\delta(q_j, (i_j, \sigma_j)) = q_{j+1}$; moreover, this run produces a sequence of outputs $\mathcal{O}(q_0, (i_0, \sigma_0), o) \cdot \mathcal{O}(q_1, (i_1, \sigma_1), o) \ldots$ for each $o \in O$.

We extend δ to a sequence of inputs in the natural way: $\delta(q, \varepsilon) = q$ and $\delta(q, (i, \sigma) \cdot w) = \delta(\delta(q, (i, \sigma)), w)$, for $(i, \sigma) \in I \times \Sigma$ and $w \in (I \times \Sigma)^*$. Similarly, we extend $\mathcal{O}$ to a sequence of inputs: $\mathcal{O}(q, \varepsilon, o) = \varepsilon$ and $\mathcal{O}(q, (i, \sigma) \cdot w, o) = \mathcal{O}(q, (i, \sigma), o) \cdot \mathcal{O}(\delta(q, (i, \sigma)), w, o)$.

A (deterministic) FL program $\mathcal{P} = (V, E)$ is a directed graph where V is a set of filters and E is a set of connections between filters in V. Each connection in E is of the form (v, v', o, i), where $v, v' \in V$, o is an output port of v and i is an input port of v'.

For a filter $v \in V$, we write I_v, O_v, Q_v, δ_v, $\mathcal{O}_v$, and q_{0v}, respectively, to denote its components.

Example 1 (Kalman Filter)**.** *We model the Kalman filter example from Section 2 as a deterministic filter with one input port y (the measurement) taking values in* $\mathbb{R}^d$*, one output port x (the mean and covariance matrix representing the estimate), taking values in* $(\mathbb{R}^n, \mathbb{R}^{n \times n})$*. The internal states maintain the prior mean and covariance matrix; the set of internal states is thus* $\mathbb{R}^n \times \mathbb{R}^{n \times n}$*. The (deterministic) transition function applies the Kalman filter algorithm to derive the new mean and covariance from the old mean and covariance and the measured input. The output function outputs the state.* ∎

3.2 Deterministic Semantics

Let $\mathcal{P} = (V, E)$ be a FL program. A *configuration* of the program $\mathcal{P}$ is a tuple $(\mathbf{q}, \mathbf{e})$, where $\mathbf{q}$ maps each filter $v \in V$ to an internal state in Q_v and $\mathbf{e}$ maps each connection $(v, v', o, i) \in E$ to a sequence in Σ^*. Intuitively, $\mathbf{q}$ gives the internal states of each filter and $\mathbf{e}$ gives the queue contents for each connection. Let $\mathcal{C}$ be the set of all configurations.

The semantics of $\mathcal{P}$ is given as a labeled transition system over configurations in $\mathcal{C}$. We introduce some notation first. For a map $\mathbf{q}$ from filters to their states, a filter v, and a state $q \in Q_v$, we write $\mathbf{q}[v \mapsto q]$ for the function that maps v to q and every other filter v' to $\mathbf{q}(v')$. Similarly, we write $\mathbf{e}[e \mapsto w]$ for a function that maps the connection e to $w \in \Sigma^*$ and every other connection e' to $\mathbf{e}(e')$.

There is a *transition* $(\mathbf{q}_1, \mathbf{e}_1) \xrightarrow{e} (\mathbf{q}_2, \mathbf{e}_2)$ between configurations if the following holds:

1. There is a connection $e = (v, v', o, i) \in E$, and $\mathbf{e}_1(v, v', o, i) = \sigma \cdot w$ for some $\sigma \in \Sigma$ and $w \in \Sigma^*$.
2. $\mathbf{q}_2 = \mathbf{q}_1[v' \mapsto \delta_{v'}(\mathbf{q}_1(v'), (i, \sigma))]$.
3. Let $\mathbf{e}'_1 = \mathbf{e}_1[(v, v', o, i) \mapsto w]$. Define

$$\mathbf{e}_2(e) = \begin{cases} \mathbf{e}'_1(e) \cdot \mathcal{O}(\mathbf{q}_1(v'), (i, \sigma), o') & \text{if } e \equiv (v', \cdot, o', \cdot) \\ \mathbf{e}'_1(e) & \text{otherwise} \end{cases}$$

Informally, the first event in some input port of a filter v is processed by v, the internal state of v is updated according to its transition function δ_v, and new events are added to the output ports of v according to its output function.

Notice that the transition relation is non-deterministic, and depends on the ordering in which events are consumed by the filters. We formalize this non-determinism by introducing a *scheduler*. A scheduler is a partial function that takes as input a configuration $(\mathbf{q}, \mathbf{e})$ and returns a connection $(v, v', o, i) \in E$. A scheduler is *maximal* if it returns a connection whenever there exists some connection $e \in E$ with $\mathbf{e}(e) \neq \varepsilon$. In the following, we represent a scheduler simply as a word over E, with the understanding that in each step, the scheduler returns the corresponding connection from the word and whenever it returns a connection (v, v', o, i), the current configuration $(\mathbf{q}, \mathbf{e})$ is such that $\mathbf{e}(v, v', o, i)$ is not ε. Given a maximal scheduler $e_1 e_2 \ldots$, we write

$$(\mathbf{q}, \mathbf{e}) \xrightarrow{e_1} (\mathbf{q}_1, \mathbf{e}_1) \xrightarrow{e_2} \ldots$$

for the unique execution starting from $(\mathbf{q}, \mathbf{e})$ and following the scheduler. We say that the execution *terminates* if this sequence is finite. If the execution terminates in a configuration $(\mathbf{q}^*, \mathbf{e}^*)$, we call $\mathbf{q}^*$ a final state of the program, and note that $\mathbf{e}^*$ must map every connection in E to ε (since the scheduler was maximal).

Note that termination and the final state depends on the scheduler: it is easy to construct a program $\mathcal{P}$, a starting configuration $(\mathbf{q}, \mathbf{e})$, and two schedulers such that the program terminates for the first scheduler but does not terminate for the second, or one in which both executions terminate but in different final states.

To reason about probabilistic behaviors, we shall now extend the deterministic semantics with probabilities. We first recall some measure theoretic preliminaries.

3.3 Measure-Theoretic Preliminaries

A σ-algebra $\mathcal{B}$ on a set Ω is a collection of subsets of Ω such that $\emptyset \in \mathcal{B}$ and $\mathcal{B}$ is closed under complementation and countable union. A *measurable space* $(\Omega, \mathcal{B})$ consists of a set Ω and a σ-algebra $\mathcal{B}$ on Ω. We say a set $E \subseteq \Omega$ is *measurable* (w.r.t. $\mathcal{B}$) if $E \in \mathcal{B}$. Given measurable spaces $(\Omega, \mathcal{B}(\Omega))$ and $(\Omega', \mathcal{B}(\Omega'))$, a map $f : \Omega \to \Omega'$ is *measurable* if $f^{-1}(S) \in \mathcal{B}_\Omega$ for each $S \in \mathcal{B}(\Omega')$. Such a measurable map is called a $(\Omega', \mathcal{B}(\Omega'))$*-valued random variable* over $(\Omega, \mathcal{B}(\Omega))$.

Given a measurable space $(\Omega, \mathcal{B})$, we can define the *product space* $(\Omega \times \Omega, \mathcal{B}^2)$, where $\mathcal{B}^2$ is the smallest σ-algebra containing the measurable rectangles $A \times B$, where $A, B \in \mathcal{B}$. By induction, we can define a measurable space $(\Omega^k, \mathcal{B}^k)$ for all $k \geq 1$ and a measurable space $(\Omega^*, \mathcal{B}^*)$ over all finite sequences from Ω. Let $\mathcal{B}^*$ be the class of all sets $E = \bigcup_{k=0}^{\infty} E^k$ such that $E^k \in \mathcal{B}^k$ for all $k \geq 0$. Then, $\mathcal{B}^*$ is the minimal σ-algebra of sets in Ω^* containing all sets $E^k \in \mathcal{B}^k$ for $k \geq 0$.

Finally, we define the measurable space $(\Omega^{\mathbb{N}}, \mathcal{B}^{\mathbb{N}})$ consisting of infinite sequences over Ω and $\mathcal{B}^{\mathbb{N}}$ is the smallest σ-algebra that contains all subsets of $\Omega^{\mathbb{N}}$ of the form $C(A, n) = \{x \in \Omega^{\mathbb{N}} \mid x_n \in A\}$ for all $A \in \mathcal{B}$ and $n \in \mathbb{N}$.

Let $(\Omega, \mathcal{B}(\Omega), \mathbb{P}_\Omega)$ be a fixed probability space. Let $(Q, \mathcal{B}(Q))$ be a measurable space. A *stochastic process* over Q is a collection $\{X_n \mid n \in \mathbb{N}\}$ of $(Q, \mathcal{B}(Q))$-valued random variables defined on $(\Omega, \mathcal{B}(\Omega), \mathbb{P}_\Omega)$. These random variables induce probability distributions over $(Q, \mathcal{B}(Q))$ defined as $\mathbb{P}_\Omega \circ X_n^{-1}$ for each $n \in \mathbb{N}$.

An (infinite-state) Markov decision process (MDP) $((Q, \mathcal{B}_Q), A, T)$ consists of a measurable space $(Q, \mathcal{B}_Q)$ of *states*, where we assume Q is a Polish space, a finite set A of *actions*, and a transition kernel $T : Q \times \mathcal{B}(Q) \times A \to [0, 1]$ such that for each $q \in Q$ and $a \in A$, the map $T(q, \cdot, a)$ is a probability measure on $(Q, \mathcal{B}(Q))$, and for each $S \in \mathcal{B}(Q)$ and $a \in A$, the map $T(\cdot, S, a)$ is a $\mathcal{B}(Q)$-measurable function.

A scheduler $\mathbf{s}$ is an infinite sequence over the set A of actions. Given a scheduler $\mathbf{s}$, an MDP defines a stochastic process where

$$\mathbb{P}[X_n \in S \mid F_{n-1}, \mathbf{s}_{n-1}](\omega) = T(X_{n-1}(\omega), A, \mathbf{s}_{n-1}), \mathbb{P} - \text{a.s.}$$

for all $n \in \mathbb{N}$, $S \in \mathcal{B}(Q)$, where F_{n-1} is the smallest σ-algebra that makes all the random variables $X_0, \ldots, X_{n-1}$ measurable from (Ω, F_{n-1}) to $(Q, \mathcal{B}(Q))$.

3.4 Probabilistic FL and Probabilistic Semantics

We now define the semantics of probabilistic FL programs as (infinite-state) Markov decision processes.

Let $(\Omega, \mathcal{B}(\Omega), \mathbb{P}_\Omega)$ be a fixed probability space. Let $(Q, \mathcal{B}(Q))$ and $(\Sigma, \mathcal{B}(\Sigma))$ be measurable spaces. A *probabilistic filter*

$$\mathcal{F} = (I, O, Q, \delta, \mathcal{O}, q_0)$$

consists of a finite set of input ports I, a finite set of output ports O, a measurable space $(Q, \mathcal{B}(Q))$ of internal states, an output function $\mathcal{O} : Q \times I \times \Sigma \times O \to \Sigma^*$, and an initial state $q_0 \in Q$, as for filters, but the transition function $\delta : \Omega \times Q \times (I \times \Sigma) \to Q$, takes the sample space Ω into account.

We assume that $\delta(\cdot, q, (i, \sigma))$ is a measurable function from Ω to Q for any $q \in Q$ and $(i, \sigma) \in I \times \Sigma$. The first condition ensures that $\delta(\cdot, q, (i, \sigma))$ is a random variable and induces a probability measure over $\mathcal{B}(Q)$: $\mathbb{P}_Q[\delta(\cdot, q, (i, \sigma)) \in C] = \mathbb{P}_\Omega[\omega \mid \delta(\omega, q, (i, \sigma)) \in C]$. Second, we assume that the map $\mathcal{O} : Q \times I \times \Sigma \times O \to \Sigma^*$ is measurable (over the corresponding measurable spaces).

Finally, define $\delta^{-1} : \mathcal{B}(Q) \times Q \times (I \times \Sigma) \to \mathcal{B}(\Omega)$ as $\delta^{-1}(C, q, i, \sigma) = S$ iff $S = \{\omega \in \Omega \mid \delta(\omega, q, i, \sigma) \in C\}$. We assume that $\mathbb{P}_\Omega[\delta^{-1}(C, \cdot, \cdot, \cdot)] : Q \times I \times \Sigma \to [0, 1]$ is measurable for all $C \in \mathcal{B}(Q)$. We will need these conditions to define the transition kernel in the semantics.[3]

Note that a probabilistic filter defines a continuous-space, discrete time, stochastic dynamical system. FL programs combine such dynamical systems in an asynchronous way, with communication mediated through queues.

A probabilistic FL program is a graph (V, E), where V is a set of probabilistic filters and E is a set of connections. We define the semantics of probabilistic FL programs as an (infinite-state) Markov decision process (MDP). Fix a probabilistic FL program (V, E). As in the deterministic case, a configuration is an ordered pair $(\mathbf{q}, \mathbf{e})$, where $\mathbf{q}$ maps every probabilistic filter $v \in V$ to one of its internal states in Q_v, and $\mathbf{e}$ maps each connection in E to Σ^*. As before, let $\mathcal{C}$ be the set of all configurations. Since the product of measurable spaces is measurable, we can define a measurable space $(\mathcal{C}, \mathcal{B}(\mathcal{C}))$ over the set of configurations.

[3] In practice, these assumptions do not restrict the set of useful programs. Indeed, we do not know natural examples which do not satisfy these assumptions.

The semantics of (V, E) will be given as an MDP over the state space $(\mathcal{C}, \mathcal{B}(\mathcal{C}))$, and set of actions E. We now define the transition kernel.

We define the transition kernel $T : \mathcal{C} \times \mathcal{B}(\mathcal{C}) \times E \to [0, 1]$ as follows. Let $(\mathbf{q}, \mathbf{e}) \in \mathcal{C}$ be a configuration, $e = (v, v', o, i) \in E$, and $\mathbf{e}(e) = \sigma \cdot w$. We first define a transition function $\mathbf{\Delta} : \Omega \times \mathcal{C} \times E \to \mathcal{C}$ that states how a configuration is updated in one step. Roughly, $\mathbf{\Delta}$ is the same as the deterministic semantics, except that it takes the outcome Ω as an additional argument and passes it on to the $\delta_{v'}$ that is chosen by $e \in E$.

Formally, $\mathbf{\Delta}(\omega, (\mathbf{q}, \mathbf{e}), (v, v', o, i))$ updates $\mathbf{q}$ to $\mathbf{q}[v' \mapsto \delta_{v'}(\omega, \mathbf{q}(v'), (i, \sigma))$, and $\mathbf{e}$ to the map $\mathbf{e}''$ defined as:

$$\mathbf{e}''(e) = \begin{cases} \mathbf{e}'(e) \cdot \mathcal{O}(\mathbf{q}(v'), (i, \sigma), o') & \text{if } e \equiv (v', \cdot, o', \cdot) \\ \mathbf{e}'(e) & \text{otherwise} \end{cases}$$

where, as before, $\mathbf{e}' = \mathbf{e}[(v, v', o, i) \mapsto w]$. We define the inverse $\mathbf{\Delta}^{-1} : \mathcal{B}(\mathcal{C}) \times \mathcal{C} \times E \to \mathcal{B}(\Omega)$ as the inverse image of $\mathcal{B}(\mathcal{C})$ under $\mathbf{\Delta}(\cdot, \mathbf{q}, \mathbf{e}, e)$. Finally,

$$T(\mathbf{q}, \mathbf{e}, C, e) = \mathbb{P}_\Omega[\mathbf{\Delta}^{-1}(C, \mathbf{q}, \mathbf{e}, e)]$$

Notice that by our assumptions on δ_v for each $v \in V$, we have that $T(\cdot, C, e) : \mathcal{C} \to [0, 1]$ is a measurable function for all $e \in E$ and $C \in \mathcal{B}(\mathcal{C})$ and that $T(\mathbf{q}, \mathbf{e}, \cdot, e) : \mathcal{B}(\mathcal{C}) \to [0, 1]$ is a probability function for all $e \in E$ and all $(\mathbf{q}, \mathbf{e}) \in \mathcal{C}$.

We abuse notation to write $T_\mathbf{s}(\mathbf{q}, \mathbf{e})$ for the probability distribution over configurations obtained by starting from an initial point distribution at $(\mathbf{q}, \mathbf{e})$ and applying a scheduler $\mathbf{s}$.

4 Scheduler Independence and Local Commutativity

A desirable property for a FL program is *scheduler-independence*: the effect of a stream of inputs should be independent of the order in which events are consumed by the scheduler. A deterministic FL program $\mathcal{P}$ is *scheduler-independent* if for any initial configuration, either $\mathcal{P}$ does not terminate from the initial configuration for any scheduler, or $\mathcal{P}$ terminates in the same configuration for all schedulers.

For probabilistic FL programs, it is easy to construct examples of programs and schedulers such that the termination probabilities, starting from the same initial configuration, are different. For simplicity, we restrict to programs and schedulers which terminate almost surely. A probabilistic FL program is scheduler-independent if for any two schedulers, whenever the program terminates almost surely for both programs, both schedulers terminate with the same distribution on states.

One can give simple structural conditions for scheduler-independence. For example, all FL programs whose graph of connections is a tree is scheduler-independent.

We now give a sufficient condition for a FL program to be scheduler-independent in terms of local commutativity of filters. Again, we first describe the main ideas for the deterministic case and then extend to the probabilistic case.

4.1 Deterministic FL

The main idea to ensure scheduler independence is that each filter is "locally commutative," in a sense we make precise below. For a set Σ, we write $\mathbb{M}[\Sigma]$ for multisets over Σ. For an alphabet Σ, the *Parikh image* $\Pi \colon \Sigma^* \to \mathbb{M}[\Sigma]$ maps a word $w \in \Sigma^*$ to a multiset $\Pi(w)$ such that $\Pi(w)(a)$ is the number of occurrences of a in w. For a language L, we define $\Pi(L) = \{\Pi(w) \mid w \in L\}$.

A filter $\mathcal{F} = (I, O, Q, \delta, \mathcal{O}, q_0)$ is *commutative* if

transition commutativity for every pair of words $w_1, w_2 \in (I \times \Sigma)^*$ such that $\Pi(w_1) = \Pi(w_2)$, and each state $q \in Q$, we have $\delta(q, w_1) = \delta(q, w_2)$ and

output commutativity for each output port $o \in O$, we have $\Pi(\mathcal{O}(q, w_1, o))) = \Pi(\mathcal{O}(q, w_2, o))$.

For example, the map filter and the reduce filter for associative, commutative operators are locally commutative. A FL program (V, E) is *locally confluent* if each filter in V is commutative. Theorem 1 shows local confluence implies scheduler independence.

Theorem 1. *Locally confluent FL programs are scheduler independent: for any initial configuration, either the program does not terminate for any scheduler, or the program terminates in the same final configuration for any two schedulers.*

Notice that this theorem makes *no finiteness assumption* on the states or events. Thus, any FL program where individual filters implement commutative and associative operators (e.g., aggregation operations) are guaranteed to be scheduler-independent. We remark that the notion of commutativity generalizes commutative functions to commutative monoids. The proof of the theorem is based on an inductive argument similar to Newman's lemma from term rewriting [3]; we omit the details.

We now prove Theorem 1. First, we define an equivalence relation on configurations: $(\mathbf{q}, \mathbf{e}) \equiv (\mathbf{q}', \mathbf{e}')$ iff $\mathbf{q} = \mathbf{q}'$ and for each $e \in E$, $\Pi(\mathbf{e}(e)) = \Pi(\mathbf{e}'(e))$.

Lemma 1. *1. Let (V, E) be a locally confluent program. if $(\mathbf{q}, \mathbf{e}) \xrightarrow{e_1} (\mathbf{q}_1, \mathbf{e}_1)$ and $(\mathbf{q}, \mathbf{e}) \xrightarrow{e_2} (\mathbf{q}_2, \mathbf{e}_2)$, then there are configurations $(\mathbf{q}_3, \mathbf{e}_3)$ and $(\mathbf{q}_4, \mathbf{e}_4)$ such that $(\mathbf{q}_1, \mathbf{e}_1) \xrightarrow{e_2} (\mathbf{q}_3, \mathbf{e}_3)$, $(\mathbf{q}_2, \mathbf{e}_2) \xrightarrow{e_1} (\mathbf{q}_4, \mathbf{e}_4)$, and $(\mathbf{q}_3, \mathbf{e}_3) \equiv (\mathbf{q}_4, \mathbf{e}_4)$.*

2. Let $\mathbf{s}_1$ and $\mathbf{s}_2$ be two schedulers such that $\Pi(\mathbf{s}_1) = \Pi(\mathbf{s}_2)$. If $(\mathbf{q}, \mathbf{e}) \xrightarrow{\mathbf{s}_1} (\mathbf{q}_1, \mathbf{e}_1)$ and $(\mathbf{q}, \mathbf{e}) \xrightarrow{\mathbf{s}_2} (\mathbf{q}_2, \mathbf{e}_2)$ then $(\mathbf{q}_1, \mathbf{e}_1) \equiv (\mathbf{q}_2, \mathbf{e}_2)$.

3. Consider two schedulers $\mathbf{s}_1$ and $\mathbf{s}_2$ such that $\Pi(\mathbf{s}_1) \leq \Pi(\mathbf{s}_2)$. If $(\mathbf{q}, \mathbf{e}) \xrightarrow{\mathbf{s}_1} (\mathbf{q}_1, \mathbf{e}_1)$ and $(\mathbf{q}, \mathbf{e}) \xrightarrow{\mathbf{s}_2} (\mathbf{q}_2, \mathbf{e}_2)$, then for each $e \in E$, we have $\Pi(\mathbf{e}_1(e)) \leq \Pi(\mathbf{e}_2(e))$.

The first part of the lemma holds from the definition of local confluence and the second part by an inductive argument similar to Newman's lemma from term rewriting [3]. The third part follows by definition of commutativity and the monotonicty of the program (an event sent to a queue is never cancelled).

Lemma 2. *Let $(\mathbf{q}, \mathbf{e})$ be a configuration of a locally confluent program. Either there is no terminating execution from $(\mathbf{q}, \mathbf{e})$ or for every two schedulers $\mathbf{s}_1$ and $\mathbf{s}_2$ such that $(\mathbf{q}, \mathbf{e}) \xrightarrow{\mathbf{s}_1}$ and $(\mathbf{q}, \mathbf{e}) \xrightarrow{\mathbf{s}_2}$ both terminate, we have $\Pi(\mathbf{s}_1) = \Pi(\mathbf{s}_2)$.*

Proof. We show that there cannot be one terminating and a different non-terminating schedules at the same time. Consider two schedules $\mathbf{s}_1$ and $\mathbf{s}_2$ such that $\mathbf{s}_2$ terminates. Since $\mathbf{s}_1$ is non-terminating, Lemma 1 implies that $\Pi(\mathbf{s}_1)$ is greater than $\Pi(\mathbf{s}_2)$. Suppose $\mathbf{s}'_1$ is the minimal prefix of $\mathbf{s}_1$ such that $\mathbf{s}_1 = \mathbf{s}'_1 \cdot (e, \sigma) \cdot \mathbf{s}''_1$ for some $e \in E$, $\sigma \in \Sigma$, and scheduler $\mathbf{s}''_1$, that $\Pi(\mathbf{s}'_1)(e, \sigma) = \Pi(\mathbf{s}_2)(e, \sigma)$. By choice of minimality, we have that $\Pi(\mathbf{s}'_1) \leq \Pi(\mathbf{s}_2)$. Hence, the number of outstanding events after executing $\mathbf{s}'_1$ is at most the number of outstanding events after $\mathbf{s}_2$. However, since $\mathbf{s}_2$ is terminating, there are no outstanding events after it executes. Thus, (e, σ) could not be picked by $\mathbf{s}_1$. The same argument can be used to show that every two terminating schedules must have the same Parikh images. ∎

To prove the theorem, we note that if $\Pi(\mathbf{s}_1) = \Pi(\mathbf{s}_2)$, then (Lemma 1) the two schedules terminate in the same state and there are no outstanding events in any connection.

4.2 Probabilistic FL

A probabilistic filter is *commutative* if for each $\omega \in \Omega$, we have $\delta(\omega, \cdot, \cdot)$ is commutative. A probabilistic FL program is called *locally confluent* if each probabilistic filter is commutative.

Let e_1 and e_2 be edges in E. For each pair of outcomes $\omega_1, \omega_2 \in \Omega$, we have that the configuration reached by the scheduler $e_1 e_2$ is equivalent (in the sense of the $\equiv$ relation on configurations) to the configuration reached by the scheduler $e_2 e_1$. In the following, given two sets A and A', we write $A \equiv A'$ if there is a bijection $f : A \to A'$ such that $s \equiv f(s)$ for all $s \in A$. Note that if A is measurable, and $A \equiv A'$, then A' is measurable. We need the following lemma.

Lemma 3. *Let $(\mathcal{C}, \mathcal{B}(\mathcal{C}), E, T)$ be an MDP giving the semantics of a locally confluent probabilistic program (V, E). For $e_1, e_2 \in E$, configuration $(\mathbf{q}, \mathbf{e})$, $A \in \mathcal{B}(\mathcal{C})$, and $A \equiv A'$, we have*

$$\int T((\mathbf{q}, \mathbf{e}), dy, e_1) T(y, A, e_2) = \int T((\mathbf{q}, \mathbf{e}), dy, e_2) T(y, A', e_1)$$

For schedulers $\mathbf{s}_1, \mathbf{s}_2$ such that $\Pi(\mathbf{s}_1) = \Pi(\mathbf{s}_2)$, and sets $A \equiv A'$, we have

$$\mathbb{P}_{\mathbf{s}_1}[A \mid (\mathbf{q}, \mathbf{e})] = \mathbb{P}_{\mathbf{s}_2}[A' \mid (\mathbf{q}, \mathbf{e})]$$

The following theorem generalizes scheduler independence to the probabilistic case: for any two schedulers that terminate almost surely, we have that the distributions over states on termination are the same.

Theorem 2. *Let $\mathcal{P}$ be a locally confluent probabilistic FL program and let $(\mathbf{q}, \mathbf{e})$ be an initial configuration. Let $\mathbf{s}_1$ and $\mathbf{s}_2$ be schedulers such that $\mathcal{P}$ terminates almost surely from $(\mathbf{q}, \mathbf{e})$ under both schedulers. Then, $T_{\mathbf{s}_1}((\mathbf{q}, \mathbf{e})) = T_{\mathbf{s}_2}((\mathbf{q}, \mathbf{e}))$ almost surely.*

The proof of the theorem follows probabilistic arguments generalizing the deterministic case.

5 Implementation

We have implemented FL as a Python API called ThingFlow and we have used it to implement several event processing workflows in the domain of IoT and control systems. The core of our API are streams of events. A publisher is a source of events, and introduces new events into the workflow. Sources of these events may be sensors (any Python object that provides a `sample()` method to read its value can be used as a sensor), external systems (e.g. message queues), or other publishers. A subscriber consumes events. A filter implements both a publisher and a subscriber interface. A publisher may create multiple output event streams along its output ports. Likewise, a subscriber may accept multiple input event streams by subscribing to upstream publishers along its different input ports. When a subscriber subscribes to a publisher, it specifies a mapping between the publisher's output port and its input port:

```
publisher.subscribe(subscriber, mapping=('out_port', 'in_port'))
```

As syntactic sugar, there exists a special *default* topic, which is used when no topic is specified on a subscription.

```
publisher.subscribe(subscriber) # default mapping
```

We use Python's metaprogramming facilities to build pipelines of components, each of which accepts a single input stream on the default topic and outputs a single event stream on the default topic, without additional glue code.

The final component of the API is the scheduler. In many applications, it is important to have explicit control over the scheduling of event processing. First, sensor interfaces are often blocking, and the scheduler has to ensure that the entire pipeline does not block because a specific sensor is sleeping; instead, blocking sensors are read on a separate thread. Second, in IoT applications, it is desirable to process one event end-to-end through the pipeline than partially process many events. Third, a specific scheduler can optimize away asynchronous calls along a pipeline of processors and instead make a series of function calls. Accordingly, we allow the developer to specify a scheduler and the implementation provides "standard" implementations of schedulers on top of Python's asynchronous framework `asyncio`. A pipeline of publishers and subscribers must be explicitly scheduled. In particular, any publishers that source events into the system (e.g. sensors) must be made known to the scheduler. The default scheduling behavior replaces the asynchronous pipeline with synchronous calls on to the downstream filters.

We illustrate ThingFlow on a simple example. Consider a pipeline that captures the readings of a light sensor and uses the sensed value in a decision logic to turn a light on or off. The pipeline looks as follows:

```
lux = SensorAsOutputThing(LuxSensor())
lux.map(lambda e: e.val).transduce(SlidingMean(5)) \
   .map(lambda v: v > threshold).GpioPinOut()
```

A sensor is any object with a `sample()` method, in this case a light sensor. It is wrapped in a publisher by `SensorAsOutputThing`. The first `map` in the pipeline extracts the value from the sensed data. The `transduce` is a filter that calculates a window-based average of the last five readings. The second map transforms the sensed values into a Boolean stream, based on comparison against a threshold. Finally, `GpioPinOut()` outputs the Boolean stream to an actuator (in our implementation, an LED) through the Gpio bus.

The pipeline, by itself, does not process data. For this, we write a scheduler and expose all sources of inputs to the scheduler:

```
scheduler = Scheduler(asyncio.get_event_loop())
scheduler.schedule_periodic(lux, 2)
```

The scheduler is based on Python's asynchronous IO and periodically samples the sensor every 2s. Finally,

```
scheduler.run_forever()
```

starts scheduling the workflow (and continues scheduling until there are no more events or the program is externally stopped). The API provides a number of common schedulers.

Static analysis of ThingFlow programs is an obvious open question. We hope our semantics can provide the first step towards program analysis in this setting.

6 Conclusion

The notion of programming by composing stream transformers is a common idiom across many domains. Our starting point for FL was to provide a unifying formalism for a programming model encompassing transducers, probabilities, infinite-state, and asynchrony. While there are many languages and formalisms that cover a subset of these features, we wanted to provide a simple semantics based on transition systems for a model with all four features. One motivation for a simple programming abstraction is that verification and testing of programs can become easier. Unfortunately, the verification problems posed by FL are not amenable to existing "off-the-shelf" tools. We expect recent advances in abstraction-based *infinite-state* hybrid probabilistic system verification [10, 11] as well as analysis tools for asynchronous systems [17, 12] would lead to verification tools for FL programs. An interesting question is to understand useful abstract domains for sub-classes of FL programs. We leave this for future work.

We chose a transition systems semantics because, in our experience, domain experts in control and signal processing (our initial target domain) are more comfortable thinking "operationally." One could alternately study the denotational semantics for a compositional treatment of FL. This would lead to an abstract perspective on asynchronous composition of probabilistic transductions. We believe that the notion of *arrows* from functional reactive programming [15, 16, 23, 14, 26, 21]. can form the basis of such a semantics.

Acknowledgements. We thank Dave Beaver, Sze-Ning Chng, Holger Hermanns, Alexandra Silva, and Sadegh Soudjani for many helpful discussions.

References

[1] S. P. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2-3):261–278, 2005.

[2] Apache Spark. `http://spark.apache.org`.

[3] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[4] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[5] P. Billingsley. *Probability and measure.* Wiley, 4 edition, 2012.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL)*, pages 178–188. ACM, 1987.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong. Approximate data collection in sensor networks using probabilistic models. In *International Conference on Data Engineering, ICDE 2006*, page 48. IEEE Computer Society, 2006.

[9] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD 06: International Conference on Management of Data*, pages 73–84. ACM, 2006.

[10] S. Esmaeil Zadeh Soudjani and A. Abate. Precise approximations of the probability distribution of a Markov process in time: an application to probabilistic invariance. In *TACAS*, volume 8413 of *LNCS*, pages 547–561. Springer, 2014.

[11] S. Esmaeil Zadeh Soudjani, C. Gevaerts, and A. Abate. FAUST^2: Formal abstractions of uncountable-state stochastic processes. In *TACAS*, volume 9035 of *LNCS*, pages 272–286. Springer, 2015.

[12] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM TOPLAS*, 34(1):6:1–6:48, 2012.

[13] N. Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[14] P. Hudak. Principles of functional reactive programming. *ACM SIGSOFT Software Engineering Notes*, 25(1):59, 2000.

[15] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[16] J. Hughes. Programming with arrows. In *Advanced Functional Programming, 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 73–129. Springer, 2004.

[17] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL 2007*, pages 339–350. ACM, 2007.

[18] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. In *International Conference on Data Engineering, ICDE 2008*, pages 1160–1169. IEEE Computer Society, 2008.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM TOCS*, 18(3):263–297, 2000.

[20] D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

[21] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *J. Funct. Program.*, 21(4-5):467–496, 2011.

[22] Microsoft Linq. https://msdn.microsoft.com/en-us/library/bb308959.aspx.

[23] J. Peterson, G. D. Hager, and P. Hudak. A language for declarative robotic programming. In *1999 IEEE International Conference on Robotics and Automation*, pages 1144–1151. IEEE Robotics and Automation Society, 1999.

[24] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics.* MIT Press, 2006.

[25] S. Tripakis, C. Pinello, A. Benveniste, A. L. Sangiovanni-Vincentelli, P. Caspi, and M. D. Natale. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Computers*, 57(10):1300–1314, 2008.

[26] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages PADL 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2002.