The Art of the Meta Stream Protocol

Torrents of Streams

Christophe De Troyer[a], Jens Nicolay[a], and Wolfgang De Meuter[a]

a Vrije Universiteit Brussel, Brussels, Belgium

Abstract **Context** The rise of streaming libraries such as Akka Stream, Reactive Extensions, and LINQ popularized the declarative functional style of data processing. The stream paradigm offers concise syntax to write down processing pipelines to consume the vast amounts of real-time data available today.

Inquiry These libraries offer the programmer a domain specific language (DSL) embedded in the host language to describe data streams. These libraries however, all suffer from extensibility issues. The semantics of a stream is hard-coded into the DSL language and cannot be changed by the user of the library.

Approach We introduce an approach to modify the semantics of a streaming library by means of meta-programming at both the run-time and compile-time level, and showcase its generality.

Knowledge We show that the expressiveness of the meta-facilities is strong enough to enable push and pull semantics, error handling, parallelism, and operator fusion.

Grounding We evaluate our work by implementing the identified shortcomings in terms of a novel stream meta-architecture and show that its design and architecture adhere to the design principles of a meta-level architecture.

Importance The state of the art offers plenty of choice to programmers regarding reactive stream processing libraries. Expressing reactive systems is otherwise difficult to do in general purpose languages. Extensibility and fine-tuning should be possible in these libraries to ensure a broad variety of applications can be expressed within this single DSL.

ACM CCS 2012

- **Software and its engineering** → **Software libraries and repositories**; *Functional languages*;
- **Theory of computation** → *Functional constructs*;

Keywords stream processing, functional programming, extensibility, meta-programming, library design

The Art, Science, and Engineering of Programming

Perspective The Engineering of Programming

Area of Submission Metaprogramming and reflection

1 Introduction

Stream programming is used for expressing computations that are naturally specified as a functional pipeline of operators between data sources and sinks. Examples of this include GUI programming [9, 16, 25], distributed event-driven systems such as IoT applications [10, 15], and complex event processing [27]. Because of its many applications, stream programming has become omnipresent in contemporary software [1, 10, 13] and most mainstream programming languages offer stream processing libraries or embed a domain-specific language (DSL) for this purpose. Because the concepts and basic building blocks are quite homogeneous across most stream implementations, all implementations can in fact be considered as a DSL for the stream programming domain, and we will therefore use the term *stream DSL* to refer to these implementations.

Contemporary and widely used stream DSLs such as Akka Streams [14], RxJava [19], and GenStage [7] offer a high-level flow-based API to construct and manipulate streams of data by using functional-inspired building-blocks called "operators", such as map and filter. These basic building-blocks can be classified into two categories, which are central to our problem statement:

- *Functional operators* such as map, filter, and scan, that manipulate the data flowing through streams.
- *Non-functional operators* that manipulate *how* the data streams through the stream such as buffer and async.

As we already mentioned, this API for expressing the *functional* requirements of a stream program are homogeneous between the different DSLs. It is a different story, however, when it comes to expressing the *non-functional* requirements of stream programs. Surveying four existing stream DSLs embedded in popular general-purpose languages [22] (Akka Streams in Scala, Java Streams and RxJava in Java, and GenStage in Elixir), we identified three problems with respect to expressing non-functional requirements in these DSLs. These issues have been addressed in the past in the context of aspect-oriented programming [11, 12] and we wish to address a subset of them in our work.

1. Stream DSLs provide non-functional extensions in a non-canonical and ad-hoc fashion, hindering transplantability of programs between languages.
2. Entanglement of functional stream logic and non-functional stream execution logic, leading to reduced readability and maintainability of stream programs.
3. Limited or no means for extending and adapting the stream execution semantics from within the DSL beyond what is built into language.

In this paper, we propose meta-programming as an approach to overcome the problems we identified in a structured manner. More in particular, we examine how meta-programming can be used for expressing non-functional requirements in stream programming in a language agnostic way, disentangled from the functional requirements, and extensible by stream programmers. Central to realizing our approach is the meta-level architecture μCreek. Our approach is two-pronged: run-time meta-programming (μCreek$_r$) and compile-time meta-programming (μCreek$_c$). μCreek$_r$ allows the programmer to modify the run-time behaviour of a stream (e.g., push vs. pull based). μCreek$_c$ hands the programmer control over the construction of streams at compile-time (e.g., introduce parallelism or timestamping). To define and implement μCreek, we designed Creek, an extensible stream DSL that incorporates our meta-programming facilities. To validate our approach we implement 5 examples prominent in related work [2, 21]: operator fusion, push and pull semantics, error handling, timestamping, and operator-level parallelism.

The contributions of the work are as follows:

- Creek, an extensible stream DSL with stream-based meta-programming facilities.
- μCreek, a stream meta protocol that offers the programmer control over the structure of the streams (μCreek$_c$) by means of DAG rewriting, and the run-time semantics of the underlying stream DSL (μCreek$_r$) by exposing and hooking into a high-level communication protocol, all in the same paradigm (i.e., streams) as the base language.
- An evaluation of the expressivity and applicability of μCreek by expressing prominent problems from related work in μCreek.

Overview In Section 2 we introduce and discuss a number of examples to demonstrate the three problems we identified, and use them to motivate our approach. To explain our approach we first introduce Creek (Section 3), our extensible stream DSL, and present its compile-time meta-protocol μCreek$_c$ (Section 4), and conclude with its run-time meta-protocol, μCreek$_r$ (Section 5). We validate our meta-protocols by showing that μCreek$_c$ and μCreek$_r$ can express common shortcomings of stream DSLs identified in related work such as operator fusion, operator parallelism, and alternative propagation algorithms (e.g., push vs. pull) without making changes to the base language. After discussing related work (Section 6), we conclude the paper and mention some directions for future work and avenues for future research. (Section 7).

2 Problem Statement

We have presented three problems in the state of the art regarding stream DSLs in contemporary programming languages in Section 1. In this section we substantiate these claims. More specifically, we show by example that contemporary stream DSLs provide non-functional extensions in a non-canonical and ad-hoc fashion (Section 2.1), that they entangle functional and non-functional stream logic (Section 2.2), and that there are limited or no means for extending and adapting the stream execution semantics from within the DSL beyond what is offered by the language (Section 2.3).

Although we will discuss each problem in the context of only one or two of the prominent streaming libraries we surveyed for this purpose, we observed all languages suffer from at least one or more of the posited problems.

```
source
.map(i ->
  longLastingComputation(i))
.subscribe()
```

```
source
.subscribeOn(Schedulers.computation())
.flatMap(val ->
    just(val)
    .subscribeOn(Schedulers.computation())
    .map(i -> longLastingComputation(i))
    .subscribeOn(Schedulers.single()))
.subscribe();
```

Figure 1 Sequential program containing a long-lasting computation (left), and its parallel version (right) in RxJava.

```
source
.map((i) =>
  longLastingComputation(i))
.deploy()
```

```
val processor: Flow[Int, Int, NotUsed] =
 Flow.fromGraph(GraphDSL.create() { implicit b =>
  val balance = b.add(Balance[Int](2))
  val merge = b.add(Merge[Int](2))
  val f = Flow[Int].map(longLastingComputation)

  balance.out(0) ~> f.async ~> merge.in(0)
  balance.out(1) ~> f.async ~> merge.in(1)
  FlowShape(balance.in, merge.out)
})
source.via(processor).runForeach(println)
```

Figure 2 Sequential program (left) from Figure 1 and its parallel version (right) expressed in Akka Streams.

2.1 Problem 1: Lack of Canonical Non-Functional Operators

Stream execution semantics is concerned with how a definition of a stream is executed at run-time to reduce the stream to a result. Given the wide range of applications in which stream DSLs are applicable, there are situations in which the default stream execution semantics offered by a stream DSL does not (or no longer) match with the intended semantics. Stream DSLs cater to this problem by offering non-functional operators to manipulate how a stream is executed with respect to propagation semantics (push vs. pull, back-pressure), concurrency and parallelism, buffering, error handling, etc. These operators are used to improve on one or more non-functional requirements such as scalability, performance, monitoring, maintainability, or recoverability. However, while the typical building blocks for expressing the functionality of stream programs are extremely similar across different stream DSLs, this is not the case for the non-functional operators. Moreover, different DSLs offer different sets of non-functional operators for supporting different non-functional requirements.

2.1.1 Example: Ad-hoc Parallelization in RxJava and Akka Streams

Suppose that a stream program must map some long-lasting computation (represented by the function longLastingComputation) over a stream of values, and that applying this operator in parallel would increase throughput. Figure 1 shows a small RxJava program containing the longLastingComputation. The left-hand side is the sequential version, and the right-hand side is a parellelized version. Due to the poor integration of parallelism in RxJava, the stream programmer is forced to modify the layout of the stream substantially to facilitate for the parallelism (lines 4–6), making its functionality less readable. Figure 2 contains the same sequential example program on the left, and the parallelized version on the right, but written in Akka Stream. As in RxJava, parallelizing a single operator introduces a significant amount of non-essential complexity, significantly increasing the number of LoC. From the code in Figures 1 and 2 we make the following observations:

- Although both RxJava and Akka Streams allow the parallelization of operators, the mechanism to do so is very different. RxJava requires the programmer to manually place computations on different threads of computation, while Akka Stream requires the programmer to manually create parallel pipelines of computation.
- Although parallelization is a non-functional concern, programmers in RxJava and Akka Streams have to modify the original stream to express this requirement.

2.1.2 Example: Error Handling in Akka Streams and RxJava

Listing 1 Error handling in Akka Streams using Decider. All IllegalArgumentExceptions cause a restart, and other exceptions stop the stream execution.

```
val decider: Supervision.Decider = {
  case _: IllegalArgumentException => Supervision.Restart
  case _                           => Supervision.Stop
}
val flow = Flow[Int]
  .scan(0) { (acc, elem) =>
    if (elem < 0) throw new IllegalArgumentException("negative not allowed")
    else acc + elem
  }
  .withAttributes(ActorAttributes.supervisionStrategy(decider))
val source = Source(List(1, 3, -1, 5, 7)).via(flow)
val result = source.runWith(Sink.seq)
```

Stream DSLs offer facilities for the programmer to deal with exceptions during stream execution. For instance, Akka Streams modularizes exception handling of a stream by extracting the behaviour into a separate object called a Decider. The programmer defines what should happen in case of a specific exception class. The example in Listing 1 defines a Decider object that, when an IllegalArgumentException occurs, restarts the stream entirely. If the stream produces any other type of error the entire stream is aborted and is reduced to an error value. If the source of the stream produces the erroneous value again the stream will restart, possibly creating an infinite loop if no limit is placed on the restarting.

Decider behavior can be added in a modular fashion, separated from the actual stream logic. For instance, the stream program in Listing 1 instructs the scan operator to employ the decider object. However, not all stream DSLs offer this kind of (modular) error handling. In RxJava, for example, one cannot express something similar to the Decider object. The only way for the programmer to handle errors is by changing the definition of the stream by inserting additional non-functional operators such as retry and retryWithValue. We do note that any approach discussed here reexecutes the entire stream. Ignoring an erroneous value is impossible.

2.1.3 Problem 1: Conclusion

Stream DSLs feature non-functional operators that address non-functional concerns. However, opposed to the work in Object-Oriented Programming [11, 12], there is no canonical approach to deal with non-functional concerns. Stream DSLs approach non-functional concerns in an ad-hoc fashion, resulting in completely different approaches for the same problem in different DSLs.

2.2 Problem 2: Entanglement of Functional and Non-functional Operators

In stream programming the domain logic of the stream (i.e., *what* the stream computes) is a separate concern from the execution semantics of the stream (i.e., *how* the stream executes). The software engineering principle of “Separation of Concerns” dictates that the different concerns of an application should be defined in separate parts of code, separated by a clean interface. This implies that functional and non-functional operators should not be mixed in the specification of a stream. Additionally, using non-functional operators in stream DSLs often forces the specification of the stream program to be modified to suit the inclusion of those operators. In Section 2.1 we have already encountered examples where this was the case; in the larger example that follows we revisit parallelization in Akka Streams.

2.2.1 Example: Entangled Parallelization in Akka Streams

Consider the example program in Listing 2, that analyzes a stream of tweets containing the keywords “covid19” and “sars-covid” for its sentiment: positive, neutral, or negative. Every minute, the program prints the ratio of positive, negative, and neutral sentiments to the console. Given that sentiment analysis is a resource-intensive computation, this step is parallelized with the goal of improving the throughput of the entire stream.

When starting out from a completely sequential, program two changes need to be made. Firstly, a parallel pipeline needs to be created that balances the tweets over two sentiment analyzers (lines 5–10 in Listing 2). Secondly, the sentiment analysis step must be factored out into a separate flow (lines 1–3 in Listing 2) so that it can be referenced twice in the parallel pipeline. While the sequential program is straightforward and declares what should happen to each tweet, the parallel version mixes the domain logic with the parallelization logic.

Listing 2 Computing the average sentiment of tweets related to COVID-19 in parallel in Akka Streams.

```
val analyze = Flow[Status].flatMapConcat((tweet: Status) => {
  var result  = Sentiment.computeSentiment(tweet.getText())
  Source(result)
})
val analyzer = Flow.fromGraph(GraphDSL.create() { implicit builder =>
  val dispatchTweets = builder.add(Balance[Status](2))
  val mergeSentiments = builder.add(Merge[(String, Sentiment)](2))
  dispatchTweets.out(0) ~> analyze.async ~> mergeSentiments.in(0)
  dispatchTweets.out(1) ~> analyze.async ~> mergeSentiments.in(1)
  FlowShape(dispatchTweets.in, mergeSentiments.out)
})
val source = TwitterActor.mkActor()
val actorRef = source
  .via(analyzer)
  .map(tuple => tuple._2)
  .sliding(60)
  .map((win: Seq[Sentiment]) => {
        (win.count(s => s == Sentiment.POSITIVE),
         win.count(s => s == Sentiment.NEUTRAL),
         win.count(s => s == Sentiment.NEGATIVE))
  })
  .async
  .toMat(Sink.foreach(println))(Keep.left).deploy()
TwitterActor.pipeInto(Array("covid19", "sars-covid"), actorRef)
```

2.2.2 Problem 2: Conclusion

Non-functional concerns are mixed with functional concerns in the same code-base, which burdens the programmer with additional complexity, reduces the readability of the code, and hard-codes the parallel processing factor.

2.3 Problem 3: Hard-coded Execution Semantics

Every Stream DSL offers a specific set of non-functional operators for expressing different non-functional requirements. Because stream DSLs are applicable in many different situations, the intended semantics of the language does not always match the language‘s semantics. None of the stream DSLs we surveyed, however, provide a structured means for extending and adapting the stream execution semantics from within the DSL itself.

An example of a non-functional requirement is the propagation semantics of a stream. All the stream DSLs we investigated implement either *push-based* or *pull-based* propagation by default, or a variation thereof [6]. Push-based semantics is a good fit for *producer-driven* applications in which, every time the producer produces a new datum, the remainder of the stream computes its result. In this semantics, the producer

Listing 3 Processing temperature measurements simulating pull-based semantics in Akka Streams.

```
remoteThermometer
.buffer(Int.MaxValue, OverflowStrategy.dropHead)
.map(f => logToDB(f))
.map(f => celsiusToFahrenheit(f))
.runWith(foreach(f => {
      printf("Current temp: %fF\n", f))})
```

dictates the rate at which the stream computes. However, some applications are inherently *consumer-driven* instead: the consumer demands data when it is needed downstream. For consumer-driven applications *pull-based* propagation is a better fit than push-based propagation: the consumer will “pull” data as is needed. The consumer dictates the rate of the stream.

We illustrate this problem in Akka Streams. Akka Streams uses a complex back pressure algorithm [6] that ensures that production and consumption rate in the stream are as compatible as possible. However, in producer-driven scenarios the programmer wants pure push-based semantics. Consider the example of thermometers monitoring a thermal processing pipeline. For safety reasons it may be important that the thermometers measure as frequently as possible, so it is considered a producer-driven stream. Consider the code in Listing 3, that contains an example program that processes a remote stream of measurements from a thermometer. The programmer intended to write a producer-driven stream in this case, but due to the default semantics of the stream library (i.e., back pressured pull-based) this is not the resulting behavior.

2.3.1 Example: Emulating Push-Based Propagation in Akka Streams

The stream will run at an equilibrium between the highest rate the thermometer can measure and the rate the system can process the measurements, if the buffer operator is omitted. The buffer operator can process elements at unbounded speed (at the cost of data loss), so when it is inserted in the stream, the equilibrium of the propagation algorithm in Akka Streams between the thermometer and the buffer will be at the maximum rate of the thermometer, resulting in the intended semantics between the thermometer and the buffer, namely push-based semantics.

2.3.2 Problem 3: Conclusion

Propagation semantics are hard-coded into the stream DSL implementations. The programmer can only manipulate the streams in a limited fashion by using non-functional operators such as buffer. Moreover, the intricacies of the underlying implementation of the propagation semantics needs to be well understood to use these operators. With the current state of affairs, there is no structured approach to modify the propagation semantics.

Listing 4 List of the squares of all the even numbers between 1 to 100 written in Creek.

```
defmodule MyDAGs do
  defdag evens as filter(fn x -> even?(x) end)
  defdag squares as map(fn x -> x * x end)

  defdag square_of_evens(input, output), do: input ~> evens ~> squares ~> output

  def program() do
    src = Source.range(1, 100)
    snk = Sink.all(self())
    deploy(square_of_evens, [input: src, output: snk)
    receive do
      result -> IO.puts "Squares: #{result}"
    end
  end
end
```

3 Stream Programming with Creek

All of the popular stream DSLs we surveyed suffer from one or more problems related to expressing non-functional requirements that we identified in Section 2. In this paper we argue that meta-programming offers an elegant solution to these problems, but before we introduce our approach in Sections 4 and 5, we introduce Creek, a prototypical generic stream programming DSL that will serve as a vehicle for explaining our meta-programming approach.

3.1 Creek DSL

We introduce Creek and its features by means of the example program shown in Listing 4 that computes the list of all the squares of the even numbers between 1 to 100. Creek is a stream DSL embedded in Elixir [8], a general-purpose language that builds on top of Erlang and runs on the same virtual machine [23]. Every Creek program must be defined inside an Elixir module by using defmodule. Module MyDAGs in Listing 4 defines three DAGs (Directed Acyclic Graphs) named evens, squares, and square_of_evens using defdag. The former define DAGs that filter out the even numbers from a stream and compute the squares of each number respectively. The latter combines these two into a DAG that creates the squares of all the even numbers. DAGs form the blueprint of a stream in Creek, and they are defined by combining operators and DAGs into larger DAGs.

A DAG must be deployed with actors that provide the source and sinks. The deploy function is boundary between Creek and the host language (i.e., Elixir). Creek offers convenience functions to create source and sink actors, but sources and sinks can be any Elixir actor that implements the Creek protocol (Section 5). In Listing 4 two actors are created using the helper functions Source.range and Sink.all. Source.range(a,

Table 1 Functional operators in Creek, which represent a subset of operators that are present in most contemporary streaming DSLs.

Name	Description
map(f)	Applies f (arity 1) to each data.
filter(f)	Filters out elements for which the predicate f (arity 1) fails.
dup(n)	Creates a joint which duplicates the incoming stream n times (i.e., n output ports). $(n > 0)$
balance(n)	Creates a joint which alternates the incoming stream over n streams. $(n > 0)$
merge(n)	Interleaves n streams into one. $(n > 0)$
zip()	Combines two streams into one by creating tuples of each value of each stream.
scan(f, acc)	Scans the stream with initial value acc and function f (arity 2).

b) creates an actor that will emit all the integers going from a to b. Sink.all(pid) creates an actor that consumes all the values from the stream and sends them as a list to another actor (pid), in the example that is self(), the current process. Because we embed Creek into Elixir, the idiomatic code to handle asynchronous events is message sends. In other languages the idiomatic approach could be futures, promises, or other streams.

3.1.1 Operators

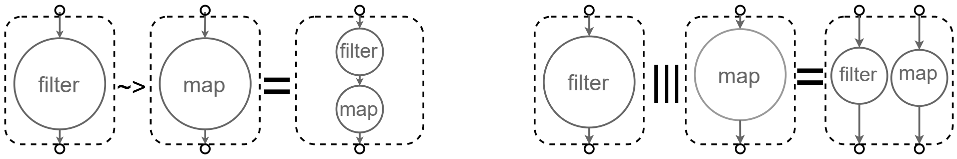
Creek uses operators as the basic building blocks for DAGs. An operator is a modular unit of data transformation logic that can be instantiated with an argument. For example, the map operator requires a user-defined function to produce values by transforming input values, and the merge operator manipulates data by combining two streams into one.

Each Creek operator has an input and output arity that determines the number of input and output "ports" they have. For example, the map operator has a single input port and a single output port, while the merge operator has 2 input ports and 1 output port. Creek implements a subset of the functional operators present in contemporary stream DSLs. Table 1 lists the operators in Creek.

Contemporary stream libraries such as Akka Streams [14] and RxJava [19] support higher-order operators (e.g., flatMap and switch) that can work with streams of streams. In essence, these operators rewrite dependencies between operators at run-time. Creek currently does not support higher-order streams. We briefly discuss what the requirements would be on the meta-level to support higher-order operators in Section 5.

3.1.2 DAGs

A composition of operators in a DAG forms the blueprint of a stream. The order of the composition determines the order of the transformations that will happen to each datum "flowing" through the DAG. An operator itself is the smallest DAG possible, consisting of only the operator itself. DAGs are created by composing DAGs using the



(a) Vertical Composition.

(b) Horizontal Composition.

Figure 3 Graphical depiction of the semantics of both composition functions.

composition functions ~> and |||. The vertical composition function (~>) connects two DAGs by connecting the output ports of the left DAG to the input ports of the right DAG. When ~> is applied to left operand a with a_i input ports and a_o output ports and right operand b with b_i input ports and b_o output ports where $a_o = b_i$, the result is a DAG with a_i input ports and b_o output ports. For example, the vertical merge of two DAGs with one input and one output port each (Figure 3a) results in a DAG that has one input port and one output port. Creek requires that $a_o = b_i$; otherwise the program is invalid.

The horizontal composition function (|||) composes two DAGs in parallel. When ||| is applied to DAGs a and b with input and output ports as denoted above, the result is a DAG with $a_i + b_i$ input ports and $a_o + b_o$ output ports. Figure 3b shows the horizontal composition of two DAGs with 1 input and 1 output port each, resulting in a DAG with 2 input ports and 2 output ports.

DAGs in Creek are classified into two types: *open* and *closed* DAGs. A DAG is called *open* if for any operator in the DAG there is an input or output port that is not connected to another operator. An open DAG is an abstraction mechanism that allows the composition of other open DAGs and individual operators. In Creek an open DAG is defined with the "defdag v as d" syntax.

Conversely, a DAG is called *closed* if it is not open; i.e., every in- and output port is connected to either a DAG, or an "actor socket". An actor socket serves as a placeholder for an actor, for when the DAG is deployed. Listing 4 line 5 defines two actor sockets: input and output. This approach decouples the DAG from actual data sources and sinks when it is deployed, facilitating reuse. In Creek a closed DAG is defined with the "defdag $v(var^*)$, do: d" syntax. Each var introduces an actor socket into the scope of the DAG definition, similar to formal parameters in function definitions. Each actor socket can have either one output or input port. In contrast to open DAGs, closed DAGs cannot be composed into larger DAGs.

3.1.3 Streams

A stream in Creek is a *deployment* of a closed DAG in which all actor sockets are replaced by actors that either inject data into the stream, or consume data from the stream. To deploy a closed DAG, a list of actor and label pairs is required where each label in a pair must reference an actor socket in the DAG. Once deployed, a stream runs asynchronously and the only way to interact with the stream is through the sources and sinks given at deployment time. The syntax for deploying a stream in Creek is deploy(d, $(actorref, var)^*$), where d is a DAG, followed by a list of actors and labels.

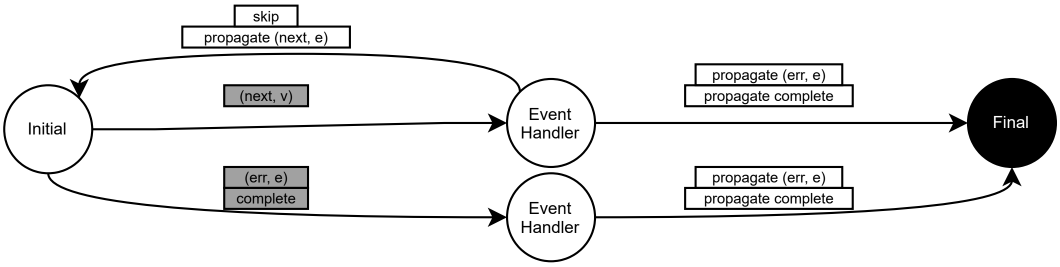


Figure 4 The protocol of an operator represented as an statemachine.

3.2 Creek Operator Protocol

Now that we have discussed the main components of a Creek program (i.e., operators, DAGs, and streams), we delve deeper into the communication protocol between operators, this will be the abstraction level at which our run-time meta μCreek$_r$ (Section 5) operates.

Due to the modularity of stream operators, an operator can only communicate with the operators directly connected to its in- and output ports. We call the communication protocol between two directly connected operators the "canonical stream protocol". This protocol is based on the communication protocol present in RxJava [18].

From the perspective of an operator, the protocol can be modelled as a statemachine (Figure 4). The statemachine in Figure 4 shows that an operator understands four distinct incoming messages (grey boxes). For each incoming message an event-handler is called. Depending on the result of the event-handler, the operator either propagates data down the stream, or ignores the event. As soon as the operator receives either an (err, e) or complete event it ends up in the final state and stops processing messages.

Source and sink actors are required to implement similar protocols, with the difference that source actors do not receive data at input ports, and sink operators do not put data on output ports. A source operator must understand one additional event in its initial state: tick. The tick event is sent by the runtime signaling the actor to propagate a value. In return it can either propagate a value ((next, v)), or it can propagate an error ((err, e)) or complete and transition to its final state.

Compile-Time Guarantees In the base language of Creek, any DAG that is deployed must adhere to a set of constraints: 1) each out- and input port for each operator is connected to another port, 2) operators have exactly as many connections as ports, 3) actor sockets have exactly one in- or output port connected, and 4) no cycles are created in the DAG. If all these constraints are met the DAG can be safely deployed. If not, the program is invalid.

4 μCreek$_c$: Modifying DAG Structure

In Section 2 we introduced three problems with contemporary stream DSLs: non-canonical operators for non-functional concerns, mixing of functional and non-functional operators, and the inability to modify the stream execution semantics from within the DSL itself.

As part of our two-pronged approach in μCreek, we propose *structural intercession* [4] to solve the mixing of functional and non-functional operators and the fact that non-functional operators are not canonicalized across stream DSLs.

In Creek, each DAG is compiled into an internal representation, as shown in Figure 6. *structural intercession* offers hooks into this process to modify its semantics. We propose *structural intercession* because (i) it is defined separately from the domain logic and it decouples from any specific domain logic by construction, (ii) if the intercession language is expressive enough it can subsume a set of individual (non-)functional operators, and (iii) the concept of intercession is not bound to a specific stream DSL, but rather the API it exposes.

In the remainder of this section we introduce our approach to *structural intercession* in our stream DSL, μCreek$_c$, which allows the meta-level to manipulate the structure of the DAG at compile-time in a stream paradigm. We then introduce the intercession language of μCreek$_c$, and we conclude with an example of applying *structural intercession* for fusing consecutive map operators in a stream. Finally, we discuss the design principles (Section 4.4).

4.1 μCreek$_c$ Architecture

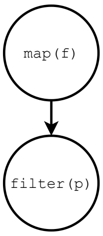
Before a DAG in a Creek program is deployed as a stream, the DAG is first compiled into an *internal representation* that is used as the blueprint for the stream. *structural intercession* gives the programmer control over the creation process of this internal representation at certain points. The structural intercession is exposed in the stream paradigm to the user. I..e, the compilation process is a stream itself.

Instruction Sequences In μCreek$_c$ each user-defined DAG is reified as an *instruction sequence* that can be transformed using a user-defined DAG. There are three different types of instructions that can be processed by this DAG. The operator and edge instruction reifiy operators and edges into their compile-time meta representation before they are added to the DAG. This allows intercession and manipulation of their definition. An operator is reified as a datastructure containing its arguments, name, and definition. An edge is reified as a datastructure containing the operator references in the DAG and the ports the edge is drawn between.

- **Operator Instruction:** For example, the instruction {:operator, :map, arg} inserts a :map operator with argument arg into the DAG.
- **Naming Instruction:** For example, the instruction {:name_it, "x"} aliases the result of the last operator instruction to the variable name "x", for reference in other instructions.
- **Edge Instruction:** For example, the instruction {:edge, "x", m, "y", n} draws an edge going from the "x" operator to the "y" operator going from port *m* to port *n*.

Figure 5a shows an example of an instruction sequence and Figure 5b the graphical representation of the resulting internal representation.

```
{:operator, :map, f}
{:name_it, "x"}
{:operator, :filter, p}
{:name_it, "y"}
{:edge, "x", 0, "y", 0}
```



(a) Instruction sequence in Elixir syntax. **(b)** Graphical representation of the result.

Figure 5 Example of evaluating an instruction sequence.

Compile-time Meta-DAG In μCreek$_c$ these sequences are streamed through the user-defined meta-DAG, which results in the compile-time DAG. Each instruction can be intercepted and transformed, altering the resulting internal representation. Each datum streamed is a pair of an instruction and the current internal representation. When an instruction has been applied to the internal representation by the compiler, the result – a new internal representation – is paired with the next instruction. Figure 6 depicts the pipeline of this process. First, the DAGs defined in the program are compiled to instruction sequences.

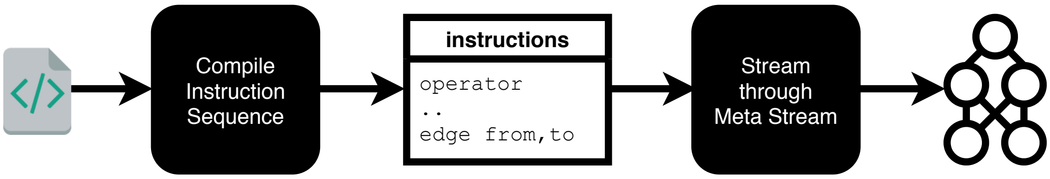


Figure 6 Graphical depiction of the compiler pipeline and the *structural intercession*.

4.2 μCreek$_c$ Language

The goal of *structural intercession* is to influence the compilation stream of the DAG. It facilitates the manipulation of the compilation by giving control to the programmer over the instructions and DAG they define.

To this end, μCreek$_c$ offers primitive functions which manipulate operators.

- fetch!(name) fetches the operator instance bound to the name var in the DAG.
- fuse!(op1, op2) fuses exactly two existing operator instances together into a new operator. For example, if two map operators o = map(f) and p = map(g) exist in the DAG, fuse!(o,p) creates a new operator equivalent to map(fn v -> g(f(v)). Multiple operators can be fused by composing fuse calls to pairs of operators.
- swap!(op1, op2) swaps two operators in the DAG while preserving the connections.
- inputs(op) returns a list of operators that are directly connected to an input port of op.

μCreek$_c$ also offers a set of primitives that help in modifying the partial compile-time DAG.

- add!(op) and delete!(op) adds and deletes operators, resp.

- connect!(op_from, fidx, op_to, tidx) and disconnect!(op_from, fidx, op_to, tidx) adds and deletes an edge, resp., between existing operators.

Compile-Time Guarantees μCREEK$_c$ allows the programmer to transform the compilation instructions at compile-time. Because the language does not prohibit the programmer to violate the constraints mentioned in Section 4.1 an additional step is transparently executed at compile-time to ensure that the μCREEK$_c$ program did not generate an invalid DAG. If the meta-program produces an invalid DAG the program will be rejected by the compiler.

In conclusion, the μCREEK$_c$ programmer can modify the instructions with compile-time DAGs as they are streamed through the CREEK compiler. The goal of the DSL offered is to build abstractions in the form of DAGs that can be composed into specific compile-time behaviours. For example, creating a DAG that fuses map operators can be composed with a DAG that parallelizes map operators to increase the throughput of a specific stream. The DAG which is being constructed is reified as a compile-time DAG structure that can be manipulated using the μCREEK$_c$ language. The result of streaming the instruction sequence is the final internal DAG.

4.3 Validation

To validate our approach we implement an example of operator fusion. Operator fusion optimizes a DAG by fusing together similar operators into a single one. Semantically the result is equivalent, but the amount of operators is reduced by combining them, thus increasing performance and reducing DAG size. Operator fusion can be considered a validation because it uses all the features present in μCREEK$_c$: 1) removing and swapping operators in the DAG at compile-time, 2) inserts operators at compile-time, and 3) changing the connections between operators in the DAG based on stream events. In summary, operator fusion subsumes problems such as logging, timestamping, and parallelizing a stream. At the end of this section we briefly discuss the approach to implement these concepts in μCREEK$_c$.

Listing 5 shows the relevant code required to implement the fusion of consecutive map operators in μCREEK$_c$. The full code is shown in Listing 1 in Appendix A.

Fusing two operators together at compile-time in μCREEK$_c$ is possible because each operator is represented as a datastructure that contains the operator type (e.g., map, filter,..), and as its arguments (i.e., the function of a map operator). Fusing two map operators together involges wrapping the arguments of the individual operators into a single function that is then used as the argument of a new map operator.

To fuse operators together it suffices to focus on the edge instructions. If an edge being drawn is between two map operators a and b it is possible to fuse a and b into c, and connect the input of a to c. All edges that are going to be connected to b must now be connected to c.

The edge DAG is defined at lines 1–20, and contains the transformation that will be applied to each edge in the DAG. First of all, the operators between which the edge is being put are fetched. If those two operators are not map operators, the instruction is unmodified (lines 17–18), otherwise it is modified as follows. To fuse the two operators (i.e., *a* and *b*) first of all a new operator *c* is created with the behaviour of applying *b* after *a*. Second, *a* and *b* are deleted from the compile-time DAG. Third, *c* is inserted into the compile-time DAG. Finally, the original instruction is transformed to create an edge between the output port of the source of *a* and the input port of *c*.

All other events (i.e., operators and name events) can be passed without modification. Combining those DAGs with the edge DAG results in a closed DAG that can be plugged into the compiler to achieve the desired behaviour. We refer to Listing 1 in Appendix A for the full implementation.

Listing 5 Example of a meta stream fusing consecutive map operators together.

```
defdag edge as filter(fn event ->
               match?({{:edge, _, _, _, _}, _, _}, event)
          end)
          ~> map(fn {{:edge, from, fidx, to, toidx}, dag, it} ->
               a = fetch!(dag, from)
               b = fetch!(dag, to)

               case {a.name, b.name} do
                    {"map", "map"} ->
                    [x] = inputs(dag, a)
                    c = fuse(a, b)
                    dag = delete(dag, a)
                    dag = delete(dag, b)
                    dag = add!(dag, c)
                    {{:edge, x.ref, 0, c.ref, 0}, dag, it}

                    _ ->
                    {{:edge, from, fidx, to, toidx}, dag, it}
               end
          end)
```

To use a compile-time module in a Creek program the structure Fusion pragma needs to be added to the module. The compiler will use the meta-DAG defined from the Fusion module defined in Listing 5. An example of a program that uses operator fusion is shown in Listing 2 in Appendix A.

4.3.1 Additional Use Cases

Timestamping Timestamping the values of a stream means that each datum is timestamped at the time it passes through an operator in a DAG. This can be achieved in contemporary stream DSLs by adding a map operator that adds a timestamp to each value. However, inserting map operators that wrap values with timestamps impacts every other domain logic operator that expects values without timestamps, so boxing and unboxing logic has to be added to other operators to handle the timestamps. Additionally, timestamps are stripped from the datum as soon as it passes through a regular operator.

Using μCREEK$_c$, this can be implemented by making two changes to the DAG at compile-time by place a timestamping map operator in front of every operator in the DAG and modifying every operator that has a function as argument (e.g., map and filter) by replacing the function argument with a wrapped function that unboxes and boxes its arguments.

Parallelization Parallelizing a DAG or operator requires that some operators, such as map operators are executed in parallel. Figure 2 shows how this is achieved in Akka Streams by duplicating operators in the DAG.

Similar to the approach for operator fusion, a map operator can be replaced with two operators: a balance operator that spreads its data over *n* instances of the original map operator, and a merge operator which merges those parallel streams together. Just like the Akka Stream and RxJava implementation in Figure 2 this approach loses the original ordering of the data. The full implementation can be found in [24].

4.4 Discussion

The design of a meta-level architecture should respect certain design principles defined by earlier work in meta-level programming. We discuss each of the principles introduced by [4] and how they are addressed in μCREEK$_c$

Encapsulation dictates that a meta-level entity should *encapsulate* its implementation details. The meta-level programs should be written against an API that decouples the underlying implementation from the meta-level programs, enabling reusability. For μCREEK$_c$ we have chosen to write meta-programs against three data structures (DAGs, edges, and operators) that offer an API that is completely decoupled from their actual implementation. The result is that μCREEK$_c$ programs can be reused for implementations in other languages, as long as the meta-level architecture offers the same API.

Stratification dictates that meta-level entities should be cleanly separated from base-level entities. This ensures that there as little coupling as possible between the base- and meta-level, and that the meta-level behaviour can be removed without breaking the base-level program. A meta-level program should not reference base-level entities (e.g., a base-level DAG with variable name proceed can not have impact on the proceed DAG at the meta-level), or vice versa. Additionally, a base-level operator should be prohibited to create explicit references to its meta-level representation. These references would make it impossible for the compiler to completely strip the meta-behaviour, because the base-level is coupled to the meta-level. If stratification is respected the meta-level architecture can be completely removed from an application if it is never used.

In μCREEK$_c$ stratification is ensured in two ways. First of all, the meta-architecture is only loaded on-demand by using the structure MyModule pragma. Secondly, the meta-level does not have access to base-level concepts, and thus no references can be made.

Structural Correspondence dictates that each language construct has a reified representation at the meta-level. μCreek$_c$ represent the components of a stream language at the meta-level: DAGs, edges, and operators. While strictly speaking the function arguments of operators should also be reified, we have chosen not to do so, as this requires a meta-level representation of concepts from the host language (i.e., Elixir), coupling μCreek$_c$ to a specific implementation. In conclusion, we have reified only the concepts that Creek introduces and draw the line at the host-level language.

Unified Programming Model The unified programming model dictates that the paradigm for the meta-level and the base-level should be the same (e.g., streams) and has a number of advantages. On the one hand, it simplifies reasoning over the code as only a limited number of programming concepts need to be considered. On the other hand, it simplifies the interaction between both levels. For this reason, μCreek$_c$ represents the construction of the DAG as a stream of instructions. Any μCreek$_c$ program is represented as DAG, which is built out of the standard Creek operators, with the additional proceed built-in DAG.

Portability The protocol (Figure 6) discussed here reifies the underlying DAG into a stream of instructions that represent changes to a DAG. These instructions can be manipulated using the base-level stream DSL and are executed by the compiler. To ensure portability this architecture cannot rely on intricasies of the underlying host language (i.e., Elixir), and therefore does not. Expressing μCreek$_c$ in any other stream language requires adapting the compiler of the stream language to generate the instruction stream and allow injecting a DAG to manipulate this stream.

5 μCreek$_r$: Modifying Stream Behaviour

In Section 4 we tackled two out of three problems posited in Section 2. The third problem posited in Section 2 is that the stream execution semantics is hard-coded into the language and cannot be changed from within the language itself.

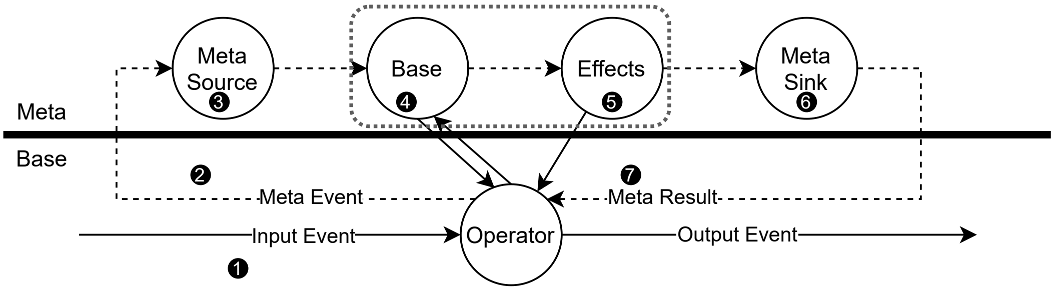
We propose – complementary to μCreek$_c$ – a *behavioural intercession* architecture. *behavioural intercession* allows a program to modify the run-time semantics of the underlying language itself. In the context of stream DSLs this means that streams can manipulate how they are executed at run-time. For example, introduce different propagation semantics, encrypt an entire stream, or catch errors to avoid retrying a stream by intercepting the execution of event-handlers.

In the remainder of this section we introduce the architecture of *behavioural intercession* for a stream DSL by implementing it in our own stream DSL Creek (Section 5.1), discuss the intercession language of μCreek$_r$ (Section 5.2), and conclude with a validation by implementing pull-based semantics, while Creek is push-based by default. Additionally we discuss the approach on how an entire stream can be encrypted (Section 5.3).

5.1 μCreek$_r$ Architecture

When a stream is executing its individual operators are exchanging messages under the hood (Section 3.2), thereby propagating the data. *behavioural intercession* hands the programmer control over this process in order to influence its semantics.

In μCreek$_r$ a meta-DAG is coupled to each operator in a stream. For any operator, each incoming event will be reified into a meta-event and pushed through this meta-DAG, circumventing the interpreter its default behaviour. Summarized, this means calling the appropriate event-handler and ensuring that in response the right events are emitted to the connected operators (i.e., effects).



■ **Figure 7** Architecture of meta-level stream: each base-level value is reified to the meta-level, and each meta-response is deified to the base-level.

Figure 7 depicts a flow diagram of how an incoming event is processed. The process starts when an event (e.g., (next, v)) arrives at the base-level (1). The runtime reifies the event to a meta-level representation (2), i.e., a meta-event, and emits it through its meta source actor (3). The user-defined meta stream should have two conceptual stages: base-level and effects. First of all, the stream should call the relevant base-level function for handling the meta-level event should be executed (4). Secondly, the stream should emit the necessary side-effects in response to the reply of the event-handler (5). For example, in response to (next, v), a value must be propagated to all operators connected to the output ports. The final value of the stream, representing the new operator state is captured by the sink actor (6). And finally, the result of the meta stream is installed as the new operator state (7), and the operator can receive a new base-level message.

5.2 μCreek$_r$ Language

In the meta-DAG operators can transform meta-level events into other meta-events, call the base-level event-handlers, and send out events to connected operators. μCreek$_r$ offers a language that simplifies these tasks, and two open DAGs that implement default behaviour for calling the base-level and handling the side-effects. Table 2 summarizes the functions available to the operators of a meta-DAG. Three functions are available to handle side-effects (i.e., propagate_*), a getter function for the operator state (i.e., state), and a function that calls the base level event-handlers (i.e., call_base).

The user-defined meta-DAG has a high degree of freedom over the entire processing pipeline of an event. However, some meta-level behaviours do not require extensive

Table 2 Functions available in μCreek$_r$

Function	Description
propagate_up(e)	Emit e to all the operators connected to an input port.
propagate_down(e)	Emit e to all the operators connected to an output port.
propagate_self(e)	Emit event e to itself.
call_base(name, arg*)	Calls the event-handler name with optional argument arg. name can be next, error, complete,tick or initialize
state()	Returns the current state of the operator.

changes, such as logging a stream. μCreek$_r$ offers two globally defined open DAGs which implement default behaviour for both cases: proceed and effects. The former expects a meta-event as input, calls the event-handler, and returns the response of the event handler. The latter takes in the response of an event-handler and executes the effects accordingly. Combined they serve as the default meta-level behaviour.

5.3 Validation

To validate μCreek$_r$ a form of pull-based propagation semantics is implemented in Creek, which is push-based by default. We consider an alternate form of propagation semantics a validation because 1) related work [2, 21] posits it as one of the key problems of contemporary stream DSLs, and 2) it requires changes in the effects of sinks, sources, and operators.

The basis of pull-based semantics is a new demand message in the protocol. This message flows from the sinks to the sources whenever data must be injected into the stream.

Source Listing 6 shows part of the implementation of pull-based propagation semantics in Creek, namely the meta-DAG for a source operator. The full implementation can be found in Appendix B. In base-level Creek, when a source receives a tick event it produces a datum. This event is sent on two occasions: when the operator is initialized, and after the operator has emitted a datum. In a pull-based system this event is omitted and will only be sent if a source demands data by sending a demand event. Additionally, a source operator must understand the demand message. In response to the demand event a source operator sends itself the tick event to trigger a datum production.

Operator If an operator receives the demand message it has to propagate it upward such that it eventually reaches a source. Additionally, if the operator does not always propagate data in repsonse to an incoming event (e.g., filter) it must propagate a new demand event upstream every time it ignores an incoming event. Operators that rely on multiple sources (e.g., zip and combineLatest [19]) can maintain some state to keep track of which sources have been sent demand and replied. This avoids sending demand twice to an operator. An example implementation can be found in Listing 9 in Appendix B.

Sink First of all, as soon as the sink operator is initialized it must propagate demand upstream to initiate the data propagation. Secondly, when a sink operator receives a value event, next to the default behaviour it must also propagate a demand message upstream to initiate the propagation of a new datum. The full implementation for pull-based semantics can be found in Appendix B.

Listing 6 μCreek$_r$ code for source operators in a pull-based stream.

```
defdag tick_src as filter(&match?({p, :tick}, &1))
            ~> base
            ~> map(fn base_result ->
              case base_result do
              {p, {state, :complete}} ->
                effects_complete(nil, p.ds, p.us, p.pid)
                %{p | state: state}

              {p, {state, :tick, value}} ->
                propagate_down({:next, value}, p.ds)
                %{p | state: state}
              end
            end)
```

5.4 Discussion

As we did in Section 4.4 we discuss μCreek$_r$'s design with respect to the design principles outlined in [4].

Encapsulation A μCreek$_r$ relies on the communication protocol defined in Figure 7 and a reification of operators. This approach almost completely decouples the μCreek$_r$ program from the base-level, as no references can be made to the base-level. The only requirement for any language to support μCreek$_r$ is to reify the inter-operator communication as the canonical stream protocol and to reify the representation of base-level nodes.

Stratification is achieved in μCreek$_r$ in the same way as it is for μCreek$_c$. The meta-program is only loaded if it has been explicitly imported using the "behavioural" pragma, and the meta-level does not have direct references to base-level concepts besides the values of the host language.

Structural Correspondence Similar to μCreek$_c$ we have chosen to only represent the concepts introduced by the Creek DSL at the meta-level (i.e., operators and events). Regular Elixir values are not reified because it would require the reification of all base-level values present in Elixir.

Listing 7 μCreek$_r$ code for sink operators in a pull-based stream.

```
defdag init_snk as filter(&match?({_, :init}, &1))
    ~> base
    ~> map(fn {p, {_, :ok}} ->
        propagate_up(:demand, p.us)
        p
    end)

defdag next_snk as filter(&match?({_, :next, _}, &1))
    ~> base
    ~> effects
    ~> map(fn {p, :ok} ->
        propagate_up_meta(:demand, p.us)
        p
    end)

defdag snk_default as filter(&(not match?({_, :init}, &1)))
    ~> filter(&(not match?({_, :next, _}, &1)))
    ~> base
    ~> effects

defdag  sink(src, snk) do
    src ~> dup(3) ~> (init_snk ||| snk_default ||| next_snk) ~> merge(3) ~> snk
end
```

Unified Programming Model A μCreek$_r$ program is a regular Creek program, with the only difference that its domain (i.e., the data) are meta-level values. We conclude that the paradigm of μCreek$_r$ and Creek are the same.

Portability To enable the implementation of the ideas of μCreek$_r$ in other languages, it cannot rely on the intricacies of the underlying language. In μCreek$_r$, the meta-level programs only process events emitted by the runtime (i.e., reified base-level events). While not every language implements the communication protocol of Creek, we argue that the message protocol presented in Figure 7 is sufficiently high-level to implement in most languages. To port the architecture of μCreek$_r$ to another language the runtime must generate and emit the events from the μCreek$_r$ protocol, and understand the messages it can generate (Table 2). Additionally, the protocol defined in [18] was used as inspiration to the μCreek$_r$ protocol and is the foundation of most JVM-based stream languages [14, 20].

Performance To assess the performance of our meta-level architecture, benchmarks have been run to measure it’s impact on execution time. Due to space constraints we do not discuss the results here, but refer the reader to Appendix C.

6 Related Work

Streaming libraries exhibit different complexities, ranging from simple embedded DSLs such as Java Streams in Java 8, to complex pipeline architectures such as Apache Kafka. Our work focusses on the smaller DSL-based libraries such as Java Streams, RxJava, and Akka Streams. Currently, work on meta-programming in the context of stream programming is sparse. However, stream DSL programming is considered "a cousin of reactive programming" [1], so we briefly discuss the intersection of reactive programming and metaprograming.

Metaprogramming in Reactive Programming In [26], Watanabe and Sawada present a reflective architecture for a functional reactive programming language called Efrp. Efrp implements a more traditional functional reactive programming approach based on time-varying values and a central clock to coordinate value propagation. An Efrp program is defined as a module containing sources, sinks and nodes depending on each other (i.e., the depedency graph). As soon as a source its value changes, all its dependencies are recomputed. The central clock ensures that glitches [5] are avoided. Recomputing a single variable and all its dependencies is called an "iteration". The reflective layer of Efrp reifies the variables in the program by name, their current value, their previous value, and the base-level expression which computes their value. A single iteration at the base-level (i.e., one variable changes) is represented as multiple iterations at the meta-level (i.e., recomputation of each of the dependencies). Because each meta-level iteration represents the update of a single dependency it is possible to intercept the incoming value for each variable, as well as influence the final result of the variable. This approach is similar to μCreek$_r$ and offers the same degree of control.

However, in μCreek$_r$ the entire communication protocol can be hooked into, which is not possible in Efrp. This results in hard-coded propagation semantics, but does offer the possibility to, for example, implement encryption.

Finally, Efrp is not an embedded DSL like Creek. This makes it easier to reify all base-level constructs all the way up to literal values. μCreek is an embedded DSL and is therefore limited to reifying concepts that are part of the DSL.

Propagation Semantics in Stream Libraries [2] presents an adaptable semantics for streaming libraries based on object algebras [21]. Current streaming libraries do not allow changing the push/pull semantics of a streaming library and they are lacking in terms of extensibility.

The authors provide a new streaming library API which allows changing the behavior of the underlying interpreter by implementing object factories. An object factory offers an API against which a semantics can be implemented (e.g., push, pull, ...). Such a factory then changes the behaviour of all the built-in operators at once. The work showcases for example logging, operator fusion, push and pull factories that each provide the behavior to the operators.

The work by Biboudis et al. offers increased flexibility regarding the run-time semantics and tackles a subset of the problems we discussed. The API offered in [2] relies on the work by S. Oliveira and Cook [21]. Object Algebras used in [2] rely on static type systems that at the least have generics and higher-kinded polymorphism. This means that the approach excludes a large set of programming languages, namely the dynamically typed languages without generics or higher-kinded polymorphism. Our approach, while requiring to be incorporated from the start of the design of a stream library, works on a higher abstraction level and is not tightly coupled to the underlying programming language. Furthermore, the approach offers an interface against which the semantics is to be programmed, but also requires to define an entire stream interpreter from scratch for even the smallest change. μCreek offers the programmer a framework to express non-functional concerns, while the runtime still does most of the heavy lifting (i.e., message sending, garbage collection,...), and offers two built-in meta-level behaviours which offer default interpreter behaviour.

7 Conclusion

We have shown a prototypical implementation of a stream DSL embedded in Elixir called Creek. We have extended this implementation with both meta-facilities at compile-time and run-time, and have shown by example that it is possible to implement various semantic changes from literature [2, 17] without having to change the implementation of the DSL itself. We validated our approach by addressing the three problems listed in Section 2 and demonstrated the applicability of the protocol. All these use-cases showcase an increase in extensibility of the framework, and flexibility for the user of the framework to tune it to its expectations.

Future Work We argued in this text that our meta-level is expressive enough to move the meta-concerns such as error handling, buffering, and propagation semantics from the base-level code to the meta-level. Additionally, we have argued that compile-time intercession is expressive enough to deal with meta-concerns such as operator-fusion and DAG optimization. However, the composability of the run-time meta-level is lacking. In future work it is worth investigating how multiple run-time meta-DAGs can be composed safely and efficiently to form a single behavior. Consider the scenario where a stream should not only be pull-based, but also timestamped. Another path worth investigating is extending the intercession to pairs of operators by means of adapting the behavior of ~>. This operator is in charge of making the connection between two or more operators and has the potential to change the behavior of how the operators communicate. Consider the scenario where one wants to connect to a remote stream on a sensor device. Hard- and software failure is very common in IoT systems, so the software should be able to deal with this in an elegant way. We propose to override the connect operator to add additional monitoring mechanics such as leasing proposed in [3]. Additionally, we would like to see support for higher-order operators. We argue that the typical scenario of “one value in, possibly one value out” does not apply for these operators, because a single input value (a higher order stream) can cause the emission of a possibly infinite amount of values. However, higher-order operators rewrite the DAG at run-time by adding sources to operators at run-time. Assuming an operator is executing with a meta behaviour M and anew operator is added as its source - as is the case in flatMap - the newly added source should understand the same meta-level messages as the original operator.

Finally, the composition of operators happens without any compile-time checks of the type of values. Possible avenues to improve upon this are contracts for run-time verification, or type annotations for compile-time verification of DAGs.

Acknowledgements This work has been funded by the Fonds Wetenschappelijk Onderzoek project D3-CPS.

References

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[2] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. "Streams a la carte: Extensible Pipelines with Object Algebras". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Edited by John Tang Boyland. Volume 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pages 591–613. DOI: 10.4230/LIPIcs.ECOOP.2015.591.

[3] Elisa Gonzalez Boix, Tom Van Cutsem, Jorge Vallejos, Wolfgang De Meuter, and Theo D'Hondt. "A Leasing Model to Deal with Partial Failures in Mobile Ad Hoc Networks". In: *Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings*. Edited by Manuel Oriol and Bertrand Meyer. Volume 33. Lecture Notes in Business Information Processing. Springer, 2009, pages 231–251. DOI: 10.1007/978-3-642-02571-6_14.

[4] Gilad Bracha and David M. Ungar. "Mirrors: design principles for meta-level facilities of object-oriented programming languages". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*. Edited by John M. Vlissides and Douglas C. Schmidt. ACM, 2004, pages 331–344. DOI: 10.1145/1028976.1029004.

[5] Gregory H. Cooper and Shriram Krishnamurthi. "Embedding Dynamic Dataflow in a Call-by-Value Language". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*. Edited by Peter Sestoft. Volume 3924. Lecture Notes in Computer Science. Springer, 2006, pages 294–308. DOI: 10.1007/11693024_20.

[6] Adam L. Davis. "Akka Streams". In: *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*. Berkeley, CA: Apress, 2019, pages 57–70. ISBN: 978-1-4842-4176-9. DOI: 10.1007/978-1-4842-4176-9_6. URL: https://doi.org/10.1007/978-1-4842-4176-9_6.

[7] *Elixir GenStage*. https://hexdocs.pm/gen_stage/GenStage.html. [Online; accessed 17-February-2020]. 2016.

[8] *Elixir Programming Language*. https://hexdocs.pm/elixir/Kernel.html. [Online; accessed 19-April-2018]. 2018.

[9] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA '13. Montpellier, France: Association for Computing Machinery, 2013. ISBN: 9781450320412. DOI: 10.1145/2489798.2489802.

[10] Shweta Khare, Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Ashish Meena. "Reactive Stream Processing for Data-Centric Publish/Subscribe". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. Oslo, Norway: Association for Computing Machinery, 2015, pages 234–245. ISBN: 9781450332866. DOI: 10.1145/2675743.2771880.

[11] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In: *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyvaskyla, Finland, June 9-13, 1997, Proceedings*. Edited by Mehmet Aksit and Satoshi Matsuoka. Volume 1241. Lecture Notes in Computer Science. Springer, 1997, pages 220–242. DOI: 10.1007/BFb0053381. URL: https://doi.org/10.1007/BFb0053381.

[13] Alexander Kirsanov, Iakov Kirilenko, and Kirill Melentyev. "Robotics Reactive Programming with F# Mono". In: *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*. CEE-SECR '14. Moscow, Russia: Association for Computing Machinery, 2014. ISBN: 9781450328890. DOI: 10.1145/2687233.2687249.

[14] Inc. Lightbend. *Akka Streams*. https://doc.akka.io/docs/akka/current/stream/index.html. [Online; accessed 19-April-2020]. 2020.

[15] H. lv, X. Ge, H. Zhu, C. Wang, Z. Yuan, and Y. Zhu. "Design and Implementation of Reactive Distributed Internet of Things Platform based on Actor Model". In: *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. 2019, pages 1993–1996. DOI: 10.1109/ITNEC.2019.8729169.

[16] Andrea Maglie. "RxJava and Android". In: *Reactive Java Programming*. Berkeley, CA: Apress, 2016, pages 95–105. ISBN: 978-1-4842-1428-2. DOI: 10.1007/978-1-4842-1428-2_6. URL: https://doi.org/10.1007/978-1-4842-1428-2_6.

[17] José Proença and Carlos Baquero. "Quality-Aware Reactive Programming for the Internet of Things". In: *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*. Edited by Mehdi Dastani and Marjan Sirjani. Volume 10522. Lecture Notes in Computer Science. Springer, 2017, pages 180–195. DOI: 10.1007/978-3-319-68972-2_12.

[18] *Reactive Streams*. http://www.reactive-streams.org/. [Online; accessed 17-February-2020]. 2014.

[19] ReactiveX. *ReactiveX: Observables done right*. https://reactivex.io/. [Online; accessed 19-April-2020]. 2014.

[20] ReactiveX. *RxJava*. https://github.com/ReactiveX/RxJava. [Online; accessed 19-April-2020]. 2020.

[21] Bruno C. d. S. Oliveira and William R. Cook. "Extensibility for the Masses - Practical Extensibility with Object Algebras". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Edited by James Noble. Volume 7313. Lecture Notes in Computer Science. Springer, 2012, pages 2–27. DOI: 10.1007/978-3-642-31057-7_2.

[22] *StackExchange 2020 Developer Survey*. https://insights.stackoverflow.com/survey/2020. [Online; accessed 27-May-2020]. 2020.

[23] Erik Stenman. *The Erlang Runtime System*. https://blog.stenmans.org/theBeamBook/. [Online; accessed 13-February-2019]. 2019.

[24] Christophe De Troyer. *Creek*. Version 0.1. Feb. 2021. DOI: 10.5281/zenodo.4487583. URL: https://doi.org/10.5281/zenodo.4487583.

[25] Sam Van den Vonder, Florian Myter, Joeri De Koster, and Wolfgang De Meuter. "Enriching the Internet By Acting and Reacting". In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. Programming '17. Brussels, Belgium: Association for Computing Machinery, 2017. ISBN: 9781450348362. DOI: 10.1145/3079368.3079407.

[26] Takuo Watanabe and Kensuke Sawada. "Towards Reflection in an FRP Language for Small-Scale Embedded Systems". In: *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming 2017, Brussels, Belgium, April 3-6, 2017*. Edited by Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter. ACM, 2017, 10:1–10:6. DOI: 10.1145/3079368.3079387.

[27] Carlos Zimmerle and Kiev Gama. "A web-based approach using reactive programming for complex event processing in internet of things applications". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Edited by Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir. ACM, 2018, pages 2167–2174. DOI: 10.1145/3167132.3167365.

A Operator Fusion

Listing 1 Example of a meta-stream fusing consecutive map operators together.

```
defmodule Fusion do
    use Structural

    defdag metadag(src, snk) do
    src ~> dup(3) ~> (edge ||| default_operators ||| default_name ) ~> merge(3) ~> proceed ~> snk
    end

    defdag edge as filter(fn event ->
                    match?({{:edge, _, _, _, _}, _, _}, event)
                    end)
                    ~> map(fn {{:edge, from, fidx, to, toidx}, dag, it} ->
                    a = fetch!(dag, from)
                    b = fetch!(dag, to)

                    case {a.name, b.name} do
                        {"map", "map"} ->
                        [x] = inputs(dag, a)
                        c = fuse(a, b)
                        dag = delete(dag, a)
                        dag = delete(dag, b)
                        dag = add!(dag, c)
                        {{:edge, x.ref, 0, c.ref, 0}, dag, it}

                        _ ->
                        {{:edge, from, fidx, to, toidx}, dag, it}
                    end
                    end)

    defdag others as filter(fn event ->
                        not match?({{:edge, _, _, _, _}, _, _}, event)
                    end)
end
```

Listing 2 Creek program which enables operator fusion for a DAG computing the square of a number plus 1.

```
defmodule MyFusedDAGs do
  use Creek
  structure Fusion

  defdag test(src, snk) do
    src
    ~> map(fn x -> x * x end)
    ~> map(fn x -> x + 1 end)
    ~> snk
  end

  def main() do
    source = Creek.Source.list([1, 2, 3, 4])
    sink = Creek.Sink.doForAll(fn x -> IO.puts x end)
    Creek.Runtime.deploy(test(), [src: source, snk: sink])
  end
end
```

B Pull-Based Semantics

This section contains the source code for pull-based semantics in Creek. The first version in Listing 8 is a naive approach that makes sinks send a demand message every time they receive a value. While this approach works, it can be fine-tuned to remove some duplicate demand messages. This implementation is shown in Listing 9. In that implementation a demand message is sent only to operators that have delivered upon their previous demand. That is to say, demand will only be sent if the previous demand was met with a reply from the upstream.

```
defmodule Pull do
use Behaviour

# This DAG handles all events with default behaviour.
defdag default as base ~> effects

##################################################################
# Operators

# This DAG propagates every demand message upstream.

defdag forward_demand as filter(&match?({_, :demand}, &1))
                          ~> map(fn {p, :demand} ->
                            propagate_upstream(:demand, p.us)
                            {p, :ok}
                          end)

# If an operator does not propagate a vlaue the demand is "lost".
# As soon as no value is propagated in response, a new demand is sent.

defdag opr_next as filter(&match?({_, :next, _}, &1))
                     ~> base()
                     ~> map(fn {p, base_response} ->
                       if match?({_, :skip}, base_response) do
                         propagate_upstream(:demand, p.us)
                       end

                       {p, base_response}
                     end)
                     ~> effects()

defdag opr_default as filter(&(not match?({_, :demand}, &1)))
                        ~> filter(&(not match?({_, :next, _}, &1)))
                        ~> default

defdag operator(src, snk) do
  src
  ~> dup(3)
  ~> (opr_default ||| forward_demand ||| opr_next)
  ~> merge(3)
  ~> snk
end
```

```
##################################################################
# Sources
# Intercept the init event for sources to stop them from ticking themselves.
defdag init as filter(&match?({_, :init}, &1))
                    ~> base()
                    ~> map(fn {p, {state, :initialized}} ->
                      # Here we would normally send tick to ourselves,
                      # but we dont (pull).
                      {%{p | state: state}, :ok}
                    end)

# This DAG handles the demand messages.
defdag demand_src as filter(&match?({p, :demand}, &1))
                      ~> map(fn {p, :demand} ->
                        # If a source receives demand it ticks itself.
                        propagate_self({:tick}, p.pid)
                        {p, :ok}
                      end)

# If a source gets a tick event (from itself)
# it will produce a value and tick itself again.
# We intercept that tick and stop from sending it.
defdag tick_src as filter(&match?({p, :tick}, &1))
                    ~> base()
                    ~> map(fn base_result ->
                      case base_result do
                        {p, {state, :complete}} ->
                          effects_complete(nil, p.ds, p.us, p.pid)
                          {%{p | state: state}, :ok}

                        {p, {state, :tick, value}} ->
                          propagate_downstream({:next, value}, p.ds)
                          {%{p | state: state}, :ok}
                      end
                    end)

# This DAG handles all events except the ones we intercepted.
defdag src_default as filter(&(not match?({_, :init}, &1)))
                       ~> filter(&(not match?({_, :meta, _}, &1)))
                       ~> filter(&(not match?({p, :tick}, &1)))
                       ~> default

defdag source(src, snk) do
  src
  ~> dup(4)
  ~> (src_default ||| init ||| demand_src ||| tick_src)
  ~> merge(4)
  ~> snk
end

##################################################################
# Sinks

```

```
    # When a sink is initialized it normally doesnt do anything.
    # In pull-based we must send the first pull message.
    defdag init_snk as filter(&match?({_, :init_snk}, &1))
                         ~> base()
                         ~> map(fn {p, {_, :ok}} ->
                           # Normally no side-effects happen in a sink init,
                           # but now e must propagate demand upstream.
                           propagate_upstream(:demand, p.us)
                           {p, :ok}
                         end)

    # This DAG ensures that a new demand is sent when a next value arrived.
    defdag next_snk as filter(&match?({_, :next, _}, &1))
                         ~> default()
                         ~> map(fn {p, :ok} ->
                           # After the default, we send demand upstream.
                           propagate_upstream(:demand, p.us)
                           {p, :ok}
                         end)

    # This DAG handles all events except the ones we intercepted.
    defdag snk_default as filter(&(not match?({_, :init_snk}, &1)))
                            ~> filter(&(not match?({_, :next, _}, &1)))
                            ~> default()

    defdag sink(src, snk) do
      src
      ~> dup(3)
      ~> (init_snk ||| snk_default ||| next_snk)
      ~> merge(3)
      ~> snk
    end
  end
```

Listing 8 Complete Implementation of Pull-Based Propagation Semantics in μCreek$_r$.

```
defmodule SmartPull do
use Creek.MetaBehaviour

dag default as base ~> effects

# Operators
dag forward_demand as filter(&match?({_, :meta, :demand, from}, &1))
                    ~> map(fn {p, :meta, :demand, from} ->
                      demanded = p.meta_state
                      to_demand = p.us |> Enum.filter(&(not MapSet.member?(
                           ↪ demanded, &1)))
                      propagate_upstream_meta(:demand, to_demand, p.pid)
                      meta_state = MapSet.new(p.us)
                      {%{p | meta_state: meta_state}, :ok}
                    end)

dag opr_next as filter(&match?({_, :next, _, _}, &1))
```

```
              ~> map(fn {p, :next, v, from} ->
                meta_state = p.meta_state |> MapSet.delete(from)
                {%{p | meta_state: meta_state}, :next, v, from}
              end)
              ~> base()
              ~> map(fn {p, base_response} ->
                if match?({_, :skip}, base_response) do
                  demanded = p.meta_state
                  to_demand = p.us |> Enum.filter(&(not MapSet.member?(demanded,
                       ↪ &1)))
                  propagate_upstream_meta(:demand, to_demand, p.pid)
                end
                {p, base_response}
              end)
              ~> effects()

dag opr_default as filter(&(not match?({_, :meta, :demand, _}, &1)))
                 ~> filter(&(not match?({_, :next, _, _}, &1)))
                 ~> filter(&(not match?({_, :init_opr}, &1)))
                 ~> default

dag init_opr as filter(&match?({_, :init_opr}, &1))
              ~> base()
              ~> map(fn {p, resp} ->
                p = %{p | meta_state: MapSet.new()}
                {p, resp}
              end)
              ~> effects()

defdag operator(src, snk) do
  src
  ~> dup(4)
  ~> (opr_default ||| forward_demand ||| opr_next ||| init_opr)
  ~> merge(4)
  ~> snk
end

# Sources

dag init_src as filter(&match?({_, :init_src}, &1))
              ~> base()
              ~> map(fn {p, {state, :initialized}} ->
                {%{p | state: state}, :ok}
              end)

dag demand_src as filter(&match?({p, :meta, :demand, _}, &1))
                ~> map(fn {p, :meta, :demand, _} ->
                  # If a source receives demand it ticks itself.
                  send_self({:tick}, p.pid)
                  {p, :ok}
                end)

dag tick_src as filter(&match?({p, :tick}, &1))
```

```
                ~> base()
                ~> map(fn base_result ->
                  case base_result do
                    {p, {state, :complete}} ->
                      effects_complete(nil, p.ds, p.us, p.pid)
                      {%{p | state: state}, :ok}

                    {p, {state, :tick, value}} ->
                      propagate_downstream({:next, value}, p.ds, p.pid)
                      {%{p | state: state}, :ok}
                  end
                end)
  dag src_default as filter(&(not match?({_, :init_src}, &1)))
                   ~> filter(&(not match?({_, :meta, _, _}, &1)))
                   ~> filter(&(not match?({p, :tick}, &1)))
                   ~> default

  defdag source(src, snk) do
    src
    ~> dup(4)
    ~> (src_default ||| init_src ||| demand_src ||| tick_src)
    ~> merge(4)
    ~> snk
  end

  # Sinks

  dag init_snk as filter(&match?({_, :init_snk}, &1))
                ~> base()
                ~> map(fn {p, {_, :ok}} ->
                  propagate_upstream_meta(:demand, p.us, p.pid)
                  {p, :ok}
                end)

  dag next_snk as filter(&match?({_, :next, _, _}, &1))
                ~> default()
                ~> map(fn {p, :ok} ->
                  propagate_upstream_meta(:demand, p.us, p.pid)
                  {p, :ok}
                end)
  dag snk_default as filter(&(not match?({_, :init_snk}, &1)))
                   ~> filter(&(not match?({_, :next, _, _}, &1)))
                   ~> default()

  defdag sink(src, snk) do
    src
    ~> dup(3)
    ~> (init_snk ||| snk_default ||| next_snk)
    ~> merge(3)
    ~> snk
  end
end
```

■ **Listing 9** Implementation of pull-based sematics that works for two-pronged operators such as zip and merge.

C Performance Benchmarks

To evaluate the performance of the meta-level architecture we benchmarked an "identity meta" (Figure 10) and compared it to a version of Creek in which all meta-level machinery was removed, called Creek–

The identity meta (Figure 10) does not do any other computation at the meta-level than calling the base-level and executing the side-effects. This ensures that any overhead measured is introduced solely by μCreek$_r$. For performance reasons the identity meta is optimized by applying operator fusion.

To measure the performance overhead we ran two benchmarks for Creek as well as Creek–. The findings are reported below.

Fixed Amount of Values, Varying DAG Size Figure 9 shows the chart for the first benchmark. In this benchmark the amount of values propagated is fixed, and the amount of operators in the DAG varies from 0 (only sink and source) to 2000.

We observe that the execution time for Creek (blue) and Creek– (orange) are both polynomial, but that the performance of Creek is slower by a factor of 3,3. In other words, the performance of the DAG decreases polynomially with respect to the size of the DAG.

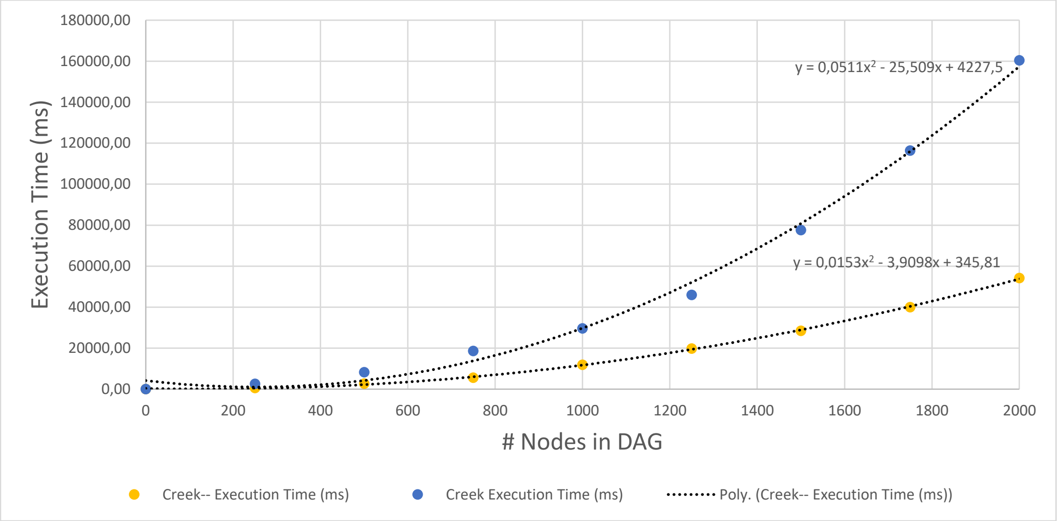


Figure 8 Execution time in miliseconds with varying DAG size and fixed load.

Fixed DAG Size, Varying Amount of Values Figure 8 shows the chart for the second benchmark. In this benchmark the amount of values propagated varies from 0 to 10000, and the amount of operators in the DAG is fixed at 250.

We observe that the execution time for Creek (orange) and Creek– (blue) are both linear with respect to the size of the input, but that the performance of Creek is slower by a factor of 77.

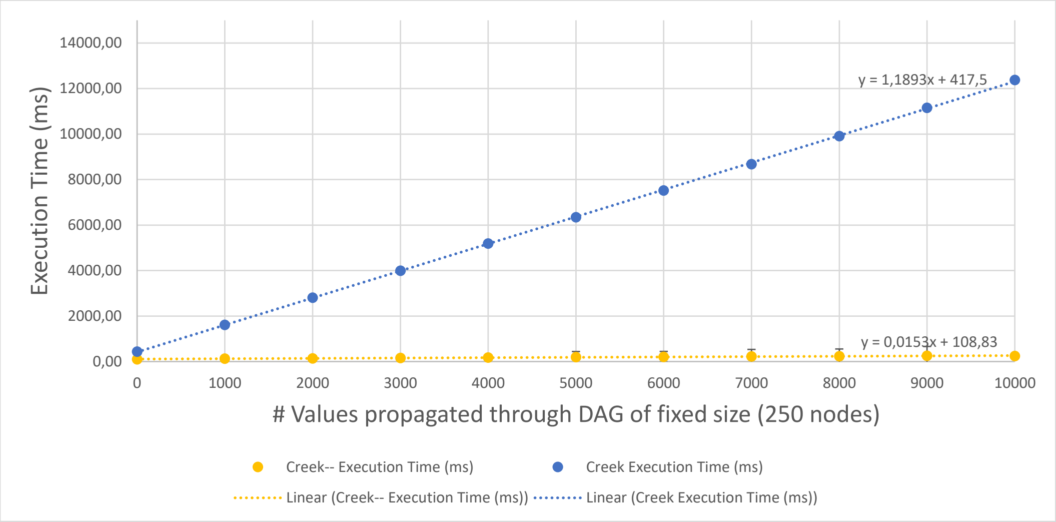


Figure 9 Execution time in miliseconds with varying load and fixed DAG size.

```
defmodule IdentityMeta do
  structure Merge

  defdag operator(src, snk) do
    src
    ~> base()
    ~> effects()
    ~> snk
  end

  defdag source(src, snk) do
    src
    ~> base()
    ~> effects()
    ~> snk
  end

  defdag sink(src, snk) do
    src
    ~> base()
    ~> effects()
    ~> snk
  end
end
```

Figure 10 Identity meta behaviour.

D Additional Use Cases

To showcase μCreek$_r$ and μCreek$_c$ we have implemented additional use cases using both the run-time and compile-time meta-levels. The code for each of these is contained in [24], and is not listed here. To make navigation in the source code easier we include a table that maps the uses cases on their respective implementation.

D.1 Compile-time Use Cases

Operator Fusion Operator fusion "fuses" together consecutive map and filter operators into a single operator.

Meta-Level	creek/lib/lang/compiler/meta_examples/merge.ex
Base-Level	creek/lib/examples/example_ctm_merge_merge.ex

Parallelization The parallelization compile-time meta allows the programmer to mark certain map operators with the "parallel: n" option. The compiler will create n parallel instances of the given map operator at compile-time.

Meta-Level	creek/lib/lang/compiler/meta_examples/par.ex
Base-Level	creek/lib/examples/example_ctm_par.ex

D.2 Run-time Use Cases

Encryption The underlying concept of encrypting a stream is that values need to "unboxed" before they are processed by an operator (i.e., removing the encryption) and "boxing" them after calling the base-level (i.e., encrypting the result). This needs to be done at every operator in the stream, assuming the source produces encrypted values.

Meta-Level	creek/lib/lang/runtime/meta/encrypted.ex
Base-Level	creek/lib/examples/example_rtm_encrypted.ex

Logging Logging is can be used to debug streams, or to log the operations of a program at deployment. The given example logs every value propagating through the stream from the source to the sink.

Meta-Level	creek/lib/lang/runtime/meta/logging.ex
Base-Level	creek/lib/examples/example_rtm_logging.ex

Pull Semantics Pull-semantics change the way data is propagated through the stream. Unchanged, Creek pushes values through the stream. That is to say, the sources dictate the rate at which data is propagated. In a pull-based scenario the sinks dictate the rate at which data is propagated through the stream.

Meta-Level	creek/lib/lang/runtime/meta/pull.ex
Base-Level	creek/lib/examples/example_rtm_pull.ex

Smart Pull Semantics This variation of the previousely mentioned pull-based semantics removes some redundant pull messages. The redundant messages are caused by operators that have two sources. If one source produces a value the operator will request data from both sources in the previous example. In this version the operator keeps track of which source has been "demanded" and which has not.

Meta-Level	creek/lib/lang/runtime/meta/pullsmart.ex
Base-Level	creek/lib/examples/example_rtm_pull.ex

About the authors

Christophe De Troyer cdetroyer@vub.ac.be

Jens Nicolay jnicolay@vub.ac.be

Wolfgang De Meuter wdmeuter@vub.ac.be