Streams à la carte

Extensible Pipelines with Object Algebras

Abstract. Streaming libraries have become ubiquitous in object-oriented languages, with recent offerings in Java, C#, and Scala. All such libraries, however, suffer in terms of extensibility: there is no way to change the semantics of a streaming pipeline (e.g., to fuse filter operators, to perform computations lazily, to log operations) without changes to the library code. Furthermore, in some languages it is not even possible to add new operators (e.g., a `zip` operator, in addition to the standard `map`, `filter`, etc.) without changing the library.
We address such extensibility shortcomings with a new design for streaming libraries. The architecture underlying this design borrows heavily from Oliveira and Cook's object algebra solution to the expression problem, extended with a design that exposes the push/pull character of the iteration, and an encoding of higher-kinded polymorphism. We apply our design to Java and show that the addition of full extensibility is accompanied by high performance, matching or exceeding that of the original, highly-optimized Java streams library.

1 Introduction

Recent years have seen the introduction of declarative streaming libraries in modern object-oriented languages, such as Java, C#, or Scala. Streaming APIs allow the high-level manipulation of value streams (with each language employing slightly different terminology) with functional-inspired operators, such as `filter`, or `map`. Such operators take user-defined functions as input, specified via local functions (lambdas). The Java example fragment below shows a "sum of even squares" computation, where the even numbers in a sequence are squared and summed. The input to `map` is a lambda, taking an argument and returning its square.

```
int sum = IntStream.of(v)
  .filter(x -> x % 2 == 0)
  .map(x -> x * x)
  .sum();
```

We observe that streaming operators introduce a separate sub-language that is interpreted during program run-time. This is quite evident in the architecture of the Java 8 streams library, which aggressively manipulates the streaming pipeline. A pipeline of the form "`of(...).filter(...).map(...).sum()`" is formed with `sum` being at the outermost layer, i.e., right-to-left as far as surrounding code is concerned. However, when the terminal operator (`sum`) is reached, it starts evaluation over the stream data by eventually invoking an iteration method

in operator `of`. It is this method that drives iteration and calls the operators left-to-right. The result of such manipulation is significant performance. Recent benchmarking studies [2] report that, by changing external (*pull-style*) iteration into internal (*push-style*), the library avoids a number of indirect calls and allows much better downstream optimizations.

The problem with existing library designs is that there is no way to alter the semantics of a streaming pipeline without changing the library itself. This is detrimental to library extensibility. For instance, a user may want to extend the library in any of the ways below:

- Create push-vs-pull versions of all operators.
- Create a logging interpretation of a pipeline, which logs actions and some intermediate results.
- Create an interpretation computing delayed versions of an evaluation (futures-of-values instead of values).
- Create an optimizing interpretation that fuses together operators, such as neighboring `filter`s or `map`s.

Additionally, the current architecture of streaming libraries prevents the introduction of new operators, precisely because of the inflexible way that evaluation is performed. As discussed above, Java streams introduce push-style iteration by default. This approach would yield semantic differences from pull-style iteration if more operators, such as `zip`, were added to the library. Furthermore, in some languages the addition of new operators requires editing the library code or using advanced facilities: in Java such addition is only possible by changing the library itself, while in C# one needs to use extension methods, and in Scala one needs to use implicits.

In our work, we propose a new design and architecture for streaming libraries for Java-like languages, to maximize extensibility without sacrificing on any other axis. Our approach requires no language changes, and only leverages features found across all examined languages—i.e., standard parametric polymorphism (generics).

Underlying our architecture is the object algebra construction of Oliveira and Cook [13] and Oliveira et al. [14]. This is combined with a library design that dissociates the push or pull nature of iteration from the operators themselves, analogously to the recent "defunctionalization of push arrays" approach in the context of Haskell [20].

Based on this architecture, we have implemented an alternative stream library for Java[1] In our library, the pipeline shown earlier gets inverted and parameterized by an `alg` object, which designates the intended semantics. For instance, a plain Java-streams-like evaluation would be written:

[1] Code in *public repo, removed for anonymity.*

```
PushFactory alg = new PushFactory();
int sum = Id.prj(
           alg.sum(
            alg.map(x -> x * x,
             alg.filter(x -> x % 2 == 0,
              alg.source(v))))).value;
```

(The `Id.prj` and `value` elements, above, are part of a standard pattern for simulating higher-kinded polymorphism with plain generics. They can be ignored for the purposes of understanding our architecture. We discuss the pattern in detail in Section 4.)

Although the above fragment is slightly longer than the original, its elements are highly stylized. The user can adapt the code to other pipelines with trivial effort, comparable to that of the original code fragment in Java 8 streams. Most importantly, if the user desired a different interpretation of the pipeline, the only necessary change is to the first line of the example. An interpretation that has pull semantics and fuses operators together only requires a new definition of `alg`:

```
FusedPullFactory alg = new FusedPullFactory();
... // same as earlier
```

Such new semantics can be defined externally to the library itself. Adding `FusedPullFactory` requires no changes to the original library code, allowing for semantics that the library designer had not foreseen.

This highly extensible design comes at no cost to performance. The new architecture introduces no extra indirection and does not prevent the JIT compiler from performing any optimization. This is remarkable, since current Java 8 streams are designed with performance in mind (cf. the earlier push-style semantics). As we show, our library matches or exceeds the performance of Java 8 streams.

Overall, our work makes the following contributions:

- We introduce a new design and architecture[2] for streaming libraries and argue for its benefits, in terms of extensibility and low adoption barrier (i.e., use of only standard language features), all without sacrificing performance.
- We demonstrate extensibility and provide several alternative semantics for streaming pipelines, all in an actual, publicly available implementation.
- We provide an example of the use of object algebras in a real-world, performance-critical setting.

[2] We follow the textbook distinction that "design" refers to how elements are separated into modules, while "architecture" refers to components-and-connectors, i.e., the machinery determining how elements of the design are composed. Our work shows a new library design, albeit one that would not be possible without a different underlying architecture.

2 Background

We next discuss streaming libraries in Java, Scala, C#, and F#. We also introduce *push/internal* vs. *pull/external* iteration, via reference to specific facilities in these libraries.

2.1 Java

Java is a relative newcomer among streaming facilities, yet features a library that has received a lot of engineering attention. We already saw examples of the Java API for streaming in the Introduction. In terms of implementation, the Java library follows a scheme that is highly optimized and fairly unique among statically typed languages.

In the Java 8 declarative stream processing API, operators fall into two categories: intermediate (*always lazy*—e.g., map and filter) and terminal (which can produce a value or perform side-effects—e.g., sum and reduce). For concreteness, let us consider the pipeline below. The expression (serving as a running example in this section) calculates the sum of all values in an array of doubles.

```
public double sumOfSquaresSeq(double[] v) {
  double sum = DoubleStream.of(v)
  .map(d -> d * d)
  .sum();
  return sum;
}
```

The code first creates a sequential, ordered `Stream` of `double`s from an array that holds all values. The calls `map` and `sum` are an intermediate and a terminal operation respectively. The `map` operation returns a `Stream` and it is lazy. It simply declares the transformation that will occur when the stream will be traversed. This transformation is a stateless operation and is declared using a lambda function. The `sum` operation needs all the stream processed up to this point, in order to produce a value; this operation is *eager* and it is effectively the same as reducing the stream with the lambda `(x,y) -> x+y`.

Implementation-wise, the (stateless or stateful) operations on a stream are represented by objects chained together sequentially. A terminal operation triggers the evaluation of the chain. In our example, *if no optimization were to take place*, the `sum` operator would retrieve data from the stream produced by `map`, with the latter being supplied the necessary lambda expression. This traversing of the elements of a stream is realized through the `Spliterator` interface. This interface offers an API for traversing and partitioning elements of a source. A key method in this interface is `forEachRemaining` with signature

```
void forEachRemaining(Consumer<? super T> action);
```

Normally, for the general case of standard stream processing, the implementation of `forEachRemaining` will internally call methods `hasNext` and `next` to tra-

verse a collection, as well as `accept` to apply an operation to the current element. Thus, three virtual calls per element will occur.

However, stream pipelines, such as the one in our example, can be optimized. For the array-based `Spliterator`, the `forEachRemaining` method performs an indexed-based, do-while loop. The entire traversal is then transformed: instead of `sum` requesting the next element from `map`, the pipeline operates in the inverse order: `map` pushes elements through the `accept` method of its downstream `Consumer` object, which implements the `sum` functionality. (A `Consumer` in Java is an operation that accepts an argument and returns no result.) In this way, the implementation eliminates two virtual calls per step of iteration and effectively uses internal iteration, instead of external. This also enables further optimizations by the JIT compiler, often resulting in fully fused code.

The following (simplified for exposition) snippet of code is taken from the `Spliterators.java` source file of the Java 8 library and demonstrates this special handling, where `a` holds the source array and `i` indexes over its length:

```
do { consumer.accept(a[i]); } while (++i < hi);
```

The internal iteration can be seen in this code. Each of the operators applicable to a stream needs to support this inverted pattern by supplying an `accept` operation. That operation, in turn, will call `accept` on whichever `Consumer<T>` may be downstream.

A term often used to describe such an internal iteration scheme is "*push iteration*". The consumer of a push stream will provide a consumer function that is instantiated into the iteration block of the stream. The dual of a push stream is a *pull stream*. Every combinator of a pull stream will build an iterator that will propagate some effect (e.g., apply a function `f` if this combinator is `map`) to each next element. C#, F# and Scala implement deferred execution over pipelines (all described in their respective sections) as pull streams. Java, on the other hand, supports push streams by default. Java additionally provides pull capabilities through the `iterator` combinator—we shall see in Section 3 why this facility is not equivalent to full pull-style iteration functionality.

2.2 Scala

Scala is an object-functional programming language for the JVM. Scala has a rich object system offering traits and mixin composition. As a functional language, it has support for higher-order functions, pattern matching, algebraic data types, and more. Since version 2.8, Scala comes with a rich collections library offering a wide range of collection types, together with common functional combinators, such as `map`, `filter`, `flatMap`, etc. The most general streaming API for Scala is that for lazy transformations of collections, which also avoids the creation of intermediate, allocated results.

To achieve lazy processing, one has to use the `view` method on a collection. This method wraps a collection into a `SeqView`. The following example illustrates the use of `view` for performing such transformations lazily:

```
def sumOfSquareSeq (v : Array[Double]) : Double = {
  val sum : Double = v
  .view
  .map(d => d * d)
  .sum
  sum
}
```

Ultimately, `SeqView` extends `Iterable[A]`, which acts as a factory for iterators. As an example, we can demonstrate the common `map` function by mapping the transformation function to the source's `Iterable` iterator:

```
def map[T, U](source: Iterable[T], f: T => U) = new Iterable[U] {
  def iterator = source.iterator map f
}
```

The `Iterator`'s `map` function can then be implemented by delegation to the source iterator:

```
def map[T, U](source: Iterator[T], f: T => U): Iterator[U] = new
    Iterator[U] {
  def hasNext = source.hasNext
  def next() = f(source.next())
}
```

Note that we have 3 virtual calls (`next`, `hasNext`, `f`) per element pointed by the iterator. This is standard pull-style iteration, as in the unoptimized Java case, discussed earlier. Each operator has to "request" elements from the one supplying its input, rather than having a push-style pattern, with the producer calling the consumer directly.

2.3 C#/F#

C# is a modern object-oriented programming language targeting the .NET framework. An important milestone for the language was the introduction of several features in C# 3.0 in order to enable a more functional style of programming. These new features, under the umbrella of LINQ [11, 12], can be summarized as support for lambda expressions and function closures, extension methods, anonymous types and special syntax for query comprehensions. All of these enable the creation of new functional-style APIs for the manipulation of collections.

F# is a modern .NET functional-first programming language based on OCaml, with support for object-oriented programming, based on the .NET object system.

In C# we can program data streams as fluent-style method calls:

```
nums.Select(x => x * x).Sum();
```

or with the equivalent query comprehension syntactic sugar:

```
(from x in nums
 select x * x).Sum();
```

In F#, stream manipulation can be expressed as a direct pipeline of various combinators.

```
nums  |> Seq.map (fun x -> x * x)
      |> Seq.sum
```

C# and F# have near-identical operational behaviors and both C# methods (`Select`, `Where`, etc.) and F# combinators (`Seq.map`, `Seq.filter`, etc.) operate on `IEnumerable<T>` objects. The `IEnumerable<T>` interface can be thought of as a factory for creating iterators, i.e., objects with `MoveNext` and `Current` methods. The lazy nature of the iterators allows the composition of an arbitrary number of operators without worrying about intermediate materialization of collections between each call. For instance, the `Select` method returns an `IEnumerable` object that produces the iterator below:

```
class SelectEnumerator<T, R> : IEnumerator<R> {
  private readonly IEnumerator<T> inner;
  private readonly Func<T, R> func;
  public SelectEnumerator(IEnumerator<T> inner,
                          Func<T, R> func) {
    this.inner = inner;
    this.func = func;
  }
  bool MoveNext() { return inner.MoveNext(); }
  R Current { get { return func(inner.Current); } }
}
```

`SelectEnumerator` implements the `IEnumerator<R>` interface and delegates the `MoveNext` and `Current` calls to the inner iterator. From a performance point of view, it is not difficult to see that there is virtual call indirection between the chained enumerators. We have 3 virtual calls (`MoveNext`, `Current`, `func`) per element per iterator. Iteration is similar to Scala or to the general, unoptimized Java iteration: it is an external (pull-style) iteration, with each consumer asking the producer for the next element.

3 Stream Algebras

We next describe our stream library architecture and its design elements, including separate push and pull semantics, enhanced interpretations of a pipeline, optimizations and more.

3.1 Motivation

The goal of our work is to offer extensible streaming libraries. The main axis of extensibility that is not well-supported in past designs is that of pluggable

semantics. There is no way to change the evaluation behavior of a pipeline so that it performs, e.g., lazy evaluation, augmented behavior (e.g., logging), operator fusing, etc. Currently, the semantics of a stream pipeline evaluation is hard-coded in the definition of operators supplied by the library. The user has no way to intervene.

The original motivation for our work was to decouple the pull- vs. pull-style iteration semantics from the library operators. As discussed in Section 2, Java 8 streams are push-style by default, while Scala, C#, and F# streams are pull-style. A recent approach in the context of Haskell [20] performs a similar decoupling of push- vs. pull-style semantics through defunctionalization of the interface, yet affords no other extensibility.

Although Java 8 streams allow some pull-style iteration, they do not support fully pluggable pull-style semantics. The current pull-style functionality is via the `iterator()` combinator. This combinator is a terminal operator and adapts a push-style pipeline into an iterator that can be used via the `hasNext`/`next` methods. This is different from changing the semantics of an entire pipeline into pull-style iteration. For instance, the `flatMap` combinator takes as input a function that produces streams, applies it to each element of a stream, and concatenates the resulting streams. In a true pull-style iteration, it is not a problem if any of the intermediate streams happen to be infinite (or merely large): their elements are consumed as needed. This is not the case when a `flatMap` pipeline is made pull-style with a terminal `iterator` call. Listing 3.1 shows a simple example. Stream `s` is infinite: it starts with zero and its step function keeps adding 2 to the previous element. The `flatMap` application produces modified copies of the infinite stream `s`, with each element multiplied by those of a finite array, `v`. Evaluation does not end until an out-of-memory exception is raised.

Listing 3.1: Infinite streams and flatMap.

```
Stream<Long> s = Stream.iterate(0L, i -> i + 2);

Iterator<Long> iterator = Stream
  .of(v)
  .flatMap(x -> s.map(y -> x * y))
  .iterator();

iterator.hasNext();
```

Our library design removes such issues, allowing pipelines with pluggable semantics. Although the separation of pull- and push-style semantics was our original motivation, it soon became evident that an extensible architecture offers a lot more options for semantic extensibility of a stream pipeline. We discuss next the new architecture and several semantic additions that it enables.

3.2 Stream as Multi-Sorted Algebras

Our extensible, pluggable-semantics design of the library is implemented using an architecture based on object algebras. Object algebras were introduced by

Oliveira and Cook [13] as a solution to the *expression problem* [23]: the need to have fully extensible data abstraction while preserving the modularity of past code and maintaining type safety. The need for extensibility arises in two forms: adding new data variants and adding new operations. We next present the elements of the object algebra approach directly in our streaming domain.

In our setting, the set of variants to extend are the different combinators: `map`, `take` (called `limit` in Java), `filter`, `flatMap`, etc. These are the different cases that a semantics of stream evaluation needs to handle. The "operations" on those variants declare the manipulation/transformation that will be employed for all produced data items. We will use the term "behavior" for such operations.

Our abstraction for streams is a multi-sorted algebra. The two sorts that can be evolved as a *family* are the type of the stream, which can hold some type of values, and the type of the value produced by terminal operations. The signature of the former is called `StreamAlg` while the latter is `ExecStreamAlg`. The `Exec*` prefix is used to denote that this is the algebra for the types that perform execution. The algebras are expressed as generic interfaces and classes implementing these interfaces are *factories*. In our multi-sorted algebra these two distinct parts are connected with the subtyping relation and classes that implement the two interfaces can evolve independendly, to form various combinations. For example, two stream factories and three exec stream factories can form $2 \times 3 = 6$ combinations.

Intermediate Combinators. Our base interface, `StreamAlg`, is shown below.

```
interface StreamAlg<C<_>> {
    <T>    C<T> source(T[] array);
    <T, R> C<R> map(Function<T, R> f, C<T> s);
    <T, R> C<R> flatMap(Function<T, C<R>> f, C<T> s);
    <T>    C<T> filter(Predicate<T> f, C<T> s);
}
```

As can be seen, `StreamAlg` is parameterized by a unary type constructor that we denote by the `C<_>` syntax. This is a device used for exposition. That is, for the purposes of our presentation we assume *type-constructor* polymorphism (a.k.a. higher-kinded polymorphism): the ability to be polymorphic on type constructors. This feature is not available in Java (although it is in, e.g., Scala).[3] In our actual implementation, type-constructor polymorphism is emulated via a standard stylized construction, which we explain in Section 4.

Every combinator of streams is also a constructor of the corresponding algebra; it returns (*creates*) values of the abstract set. Each constructor of the algebra creates a new intermediate node of the stream pipeline and, in addition to the value of the previous node (parameter `s`) that it will operate upon, it takes a *functional interface*. (A functional interface has exactly one abstract method and is the type of a lambda in Java 8.)

[3] The original object algebras work of Oliveira and Cook [13] did not require type-constructor polymorphism for its examples. Later work by Oliveira et al. [14] used type-constructor polymorphism in the context of Scala.

Terminal Combinators. The `ExecStreamAlg` interface describes terminal operators, which perform execution/evaluation of the pipeline. These operators can return a scalar value, e.g., `count` can return `Long`, hence having blocking semantics, or return a value of some container type, parameterized by some other type, e.g. `count` can return `Future<Long>` to offer asynchronous execution.

```
interface ExecStreamAlg<E<_>, C<_>> extends StreamAlg<C> {
    <T> E<Long> count(C<T> s);
    <T> E<T>    reduce(T identity, BinaryOperator<T> acc, C<T> s);
}
```

Once again, this algebra is parameterized by unary type constructors and it also carries as a parameter the abstract stream type that it will pass to its super type, `StreamAlg`.

3.3 Adding New Behavior for Intermediate Combinators

We next discuss the extensibility that our design affords, with several examples of different interpretation semantics.

Push Factory. The first implementation in our library is that of a push-style interpretation of a streaming pipeline, yielding behavior equivalent to the default Java 8 stream library.

Push-style streams implement the `StreamAlg<Push>` interface (where `Push` is the *container* or *carrier type* of the algebra). All combinators return a value of some type `Push<...>`, i.e., a type expression derived from the concrete constructor `Push`. Our `PushFactory` implementation, restricted to combinators `source` and `map`, is shown below.

Listing 3.2: Example of a PushFactory.

```
class PushFactory implements StreamAlg<Push> {
  public <T> Push<T> source(T[] array) {
    return k -> {
      for(int i=0 ; i < array.length ; i++){
        k.accept(array[i]);
      }
    };
  }

  public <T, R> Push<R> map(Function<T, R> mapper, Push<T> s) {
    return k -> s.invoke(i -> k.accept(mapper.apply(i)));
  }
}
```

A `Push<...>` type is the embodiment of a push-style evaluation of a stream. It carries a function, which can be composed with others in a *push-y* manner. In the context of Java, we want to be able to assign lambdas to a `Push<...>` reference. Therefore we declare `Push<X>` as a functional interface, with a single method,

`void invoke(Consumer<T>)`. The `Consumer<T>` argument is itself a lambda (with method name `accept`) that takes as a parameter an item of type `T` and returns void. This consumer can be thought of as the *continuation* of the evaluation (hence the conventional name, `k`). The entire stream is evaluated as a loop, as shown in the implementation of the `source` combinator, above. `source` returns a lambda that takes as a parameter a `Consumer<T>`, iterates over the elements of a source, `s`, and passes elements one-by-one to the consumer.

Similarly, the `map` operator returns a push-stream embodiment of type `Push<...>`. This stream takes as argument another stream, `s`, such as the one produced by `source`, and invokes it, passing it as argument a lambda that represents the `map` semantics: it calls its continuation, `k`, with the argument (i.e., the element of the stream) as transformed by the mapping function. This pattern follows a similar continuation-passing-style convention as in the original Java 8 streams library. (As discussed in Section 2.1, this reversal of the pipeline flow enables significant VM optimizations and results in faster code.)

The next combinator, whichever it is, will consume the transformed elements of type `R`. The implementation of other combinators, such as `filter` and `flatMap`, follows a similar structure.

Pull Factory. As discussed earlier, Java 8 streams do not have a full pull-style iteration capability. They have to fully realize intermediate streams, since the pull semantics is implemented as a terminal combinator and only affects the external behavior of an entire pipeline. (As we will see in our experiments of Section 6, this is also a source of inefficiency in practice.) Therefore, the first semantic addition in our library is pull-style streams.

Pull-style streams implement the `StreamAlg<Pull>` interface. In this case `Pull<T>` is an interface that represents iterators, by extending the `Iterator<T>` interface. For pull semantics, each combinator returns an anonymous class—one that implements this interface by providing definitions for the `hasNext` and `next` methods. In Listing 3.3 we demonstrate the implementation of the `source` and `map` operators, which are representative of others. We follow the Java semantics of iterators (the effect happens in `hasNext`). Each element that is returned by the `next` method of the `map` implementation is the transformed one, after applying the needed mapper lambda to each element that is retrieved. The retrieval is realized by referring to the `s` object, which carries the iterator of the previous pipeline step.

Listing 3.3: Example of PullFactory functionality.

```
class PullFactory implements StreamAlg<Pull> {
  public <T> Pull<T> source(T[] array) {
    return new Pull<T>() {
      final int size = array.length;
      int cursor = 0;

      public boolean hasNext() { return cursor != size; }
```

```
      public T next() {
        if (cursor >= size)
          throw new NoSuchElementException();
        return array[cursor++];
      }
    };
  }

  public <T, R> Pull<R> map(Function<T, R> mapper, Pull<T> s) {
    return new Pull<R>() {
      R next = null;

      public boolean hasNext() {
        while (s.hasNext()) {
          T current = s.next();
          next = mapper.apply(current);
          return true;
        }
        return false;
      }

      public R next() {
        if (next != null  this.hasNext()) {
          R temp = this.next;
          this.next = null;
          return temp;
        }
        throw new NoSuchElementException();
      }
    };
  }
}
```

Note how dissimilar the `Push` and `Pull` interfaces are (a lambda vs. an iterator with `next` and `hasNext`). Our algebra, `StreamAlg<C<_>>` is fully agnostic regarding `C`.

Log Factory. With a pluggable semantics framework in place, we can offer several alternative interpretations of the same streaming pipeline. One such is a logging implementation. The log factory expresses a cross-cutting concern, one that interleaves logging capabilities with the actual execution of the pipeline. Although the functionality is simple, it is interesting in that it takes a mixin form: it can be merged with other semantics, such as push or pull factories. The code for the `LogFactory`, restricted to the `map` and `count` operators, is shown in Listing 3.4,

Listing 3.4: Example of LogFactory functionality.

```
class LogFactory<E<_>, C<_>> implements ExecStreamAlg<E, C> {
  ExecStreamAlg<E, C> alg;
```

```
  <T, R> C<R> map(Function<T, R> mapper, C<T> s) {
    return alg.map(i -> {
      System.out.print("map: " + i.toString());
      R result = mapper.apply(i);
      System.out.println(" -> " + result.toString());
      return result;
    }, s);
  }

  public <T> E<Long> count(C<T> s) {
    return alg.count(s);
  }
}
```

The code employs a delegation-based structure, one that combines an implementation of an execution algebra (of any behavior for intermediates and orthogonally of any behavior for terminal combinators) with a logger. We parameterize `LogFactory` with an `ExecStreamAlg` and then via delegation we pass the intercepted lambda as the mapping lambda of the internal algebra. For example, if the developer has authored a pipeline `alg.reduce(0L, Long::sum, alg.map(x -> x + 2, alg.source(v)))`, then, instead of using an `ExecPushFactory` that will perform push-style streaming, she can pass a `LogFactory<>(new ExecPushFactory())` effectively mixing a push factory with a log factory.

Fused Factory. An interpretation can also apply optimizations over a pipeline. We provide an extension of a `PullAlgebra` that performs fusion of adjacent operations. Using a `FusionPullFactory` the user can transparently enable fusion for multiple `filter` and multiple `map` operations. In this factory, the two combinators are redefined and, instead of creating values of an anonymous class of type `Pull`, they create values of a refined version of the `Pull` type. This gives introspection capabilities to the `map` and `filter` operators. They can inspect the dynamic type of the stream that they are applied to. If they operate on a fusible version of `map` or on a fusible version of `filter` then they proceed with the creation of values for these extended types with the composed operators. We elide the definition of the factory, since it is lengthy.

3.4 Adding New Combinators

Our library design also allows adding new combinators without changing the library code. In case we want to add a new combinator, we first have to decide in which algebra it belongs. For instance, we have added a `take` combinator without disturbing the original algebra definitions. A `take` combinator has signature `C<T> take(int n)` so it clearly belongs in `StreamAlg`. We have to implement the operator for both push and pull streams, but we want to allow the possibility of using `take` with any `ExecStreamAlg`. Our approach again uses delegation, much like the `LogFactory`, shown earlier in Listing 3.4. We create a generic `TakeStreamAlg<E,`

C> interface and orthogonaly we create an interface ExecTakeStreamAlg<E, C> that extends TakeStreamAlg<C> and ExecStreamAlg<E, C>. In the case of push streams, ExecPushWithTakeFactory<E> implements the interface we created, where C = Push, by defining the take operator. All other operators for the push case are inherited from the PushFactory supertype. The ExecPushWithTakeFactory<E> factory is parameterized by ExecStreamAlg<E, Push> alg, remaining open to receive any algebra for terminal operators.

3.5 Adding New Behavior for Terminal Combinators

Future Factory. Our library design also enables adding new behavior for terminal combinators. The most interesting example in our current library components is that of FutureFactory: an interpretation of the pipeline that causes a delayed evaluation. Instead of returning scalar values, a FutureFactory parameterizes ExecStreamAlg with a concrete type constructor, Future<X>. (This is in much the same way as, e.g., a PushFactory parameterizes StreamAlg with type constructor Push, in Listing 3.2.) Future is a type that provides methods to start and cancel a computation, query the state of the computation, and retrieve its result.

FutureFactory defines terminal operators count and reduce, to return Future<Long> and Future<T> respectively. Intermediate combinators are defined similarly to the terminal ones, but are omitted from the listing.

Listing 3.5: Count and reduce operators in FutureFactory.

```
class ExecFutureFactory<C<_>> implements ExecStreamAlg<Future, C> {
  private final ExecStreamAlg<Id, C> execAlg;

  public <T> Future<Long> count(C<T> s) {
    Future<Long> future = new Future<>(() -> {
      return execAlg.count(s).value;
    });
    future.run();
    return future;
  }

  public <T> Future<T> reduce(T identity,
                              BinaryOperator<T> accumulator,
                              C<T> s) {
    Future<T> future = new Future<>(() -> {
      return execAlg.reduce(identity, accumulator, s).value;
    });
    future.run();
    return future;
  }
}
```

4 Emulating Type-Constructor Polymorphism

As noted earlier, our presentation so far was in terms of type-constructor polymorphism, although this is not available in Java. For our implementation, we simulate type-constructor polymorphism via a common technique. The same encoding has been used in the implementation of object-oriented libraries—e.g., in type classes for Java [6] and in finally tagless interpreters for C# [10]. The technique was also recently presented formally by Yallop and White [25], and used to represent higher-kinded polymorphism in OCaml.

In this encoding, for an unknown type constructor `C<_>`, the application `C<T>` is represented as `App<t, T>`, where `T` is a Java class and `t` is a marker class that identifies the type constructor `C`. For example, our stream algebra shown in Section 3.2 is written in plain Java as follows:

```
public interface App<C, T> { }

public interface StreamAlg<C> {
    <T> App<C, T> source(T[] array);
    <T, R> App<C, R> map(Function<T, R> f, App<C, T> app);
    <T, R> App<C, R> flatMap(Function<T, App<C, R>> f, App<C, T> app);
    <T> App<C, T> filter(Predicate<T> f, App<C, T> app);
}
```

A subtle point arises in this encoding: given `C`, how is `t` generated? This class is called the "brand", as it tags the application so that it cannot be confused with applications of other type constructors; this brand should be extensible for new types that may be added later to the codebase. This means that (a) `t` should be a fresh class name, created when `C` is declared; and (b) there should be a protocol to ensure that the representation is used safely.

Brand freshness. The freshness of the brand name is addressed by declaring `t` as a nested class inside the class of the new type constructor. Since `t` exists at a unique point in the class hierarchy, no other class may declare a brand that clashes with it, and its declaration happens at the same time as `C` is declared. In the following, we see the encoding of the type constructor `Pull<T>`, with its `t` brand:

```
public interface Pull<T> extends App<Pull.t, T>, Iterator<T> {
    static class t { }
    static <A> Pull<A> prj(App<Pull.t, A> app) { return (Pull<A>) app; }
}
```

We see that the encoding above has an extra method `prj`, which does a downcast of its argument. The OCaml encoding of Yallop and White needs two methods `inj` and `prj` (for "inject" and "project") that cast between the concrete type and the instantiation of the type application. In Java, we define `prj`, which takes the representation of the type application and returns the actual `Push<T>` instantiation. In contrast to OCaml, Java has subtyping, so `inj` functions are not

needed: a `Pull<T>` object can always be used as being of type `App<Pull.t, T>`. The `Iterator` interface in the declaration above is not related to the encoding, but is part of the semantics of pull-style streams.

Safely using the encodings. This encoding technique has a single unchecked cast, in the `prj` function. The idea is that the cast will be safe if the only way to get a value of type `App<Pull.t, X>` (for any `X`) is if it is really a value of the subtype, `Pull<X>`. This property clearly holds if values of type `App<Pull.t, X>` (or values of any type involving `Pull.t`) are never constructed. In the Yallop and White technique for OCaml, this is ensured syntactically by the "freshness" of the brand, `t`, which is private to the type constructor. In Java, the property is ensured by convention: every subtype `S` of `App` has a locally defined brand `t` and no subtype of `App<S.t, X>` other than `S` exists.

Type expressions without type-constructor polymorphism. Another detail of the encoding is the representation of type expressions that are not parametric according to a type constructor; for those we need an identity type application, `Id`.

```
public class Id<T> implements App<Id.t, T> {
    public final T value;
    public Id(T value) { this.value = value; }
    public static class t { }
    public static <A> Id<A> prj(App<Id.t, A> app) { return (Id<A>) app; }
}
```

Using the class above, the type expression `List<Integer>` can then be represented as `Id<List<Integer>>`.

5 Using Streams

With the encoding of type-constructor polymorphism, our description of the library features is complete. A user can employ all combinators to build pipelines, and can flexibly choose the semantics of these pipelines.

The example of Listing 5.1 declares a pipeline that filters long integers and then counts them. The expression assumes an implementation, `alg`, of a stream algebra. Note that the prefix, `Id.prj`, and suffix, `value`, of the pipeline expression are only needed for our type-constructor polymorphism simulation.

Listing 5.1: Count of filtered items.

```
Long result = Id.prj(
                alg.count(
                  alg.filter(x -> x % 2L == 0,
                    alg.source(v)))).value;
```

Similarly, Listing 5.2 constructs a sum of the cartesian product pipeline between two arrays. The factory object (implementing the algebras) is factored out and becomes a parameter of the method `cart`.

Listing 5.2: Sum of the cartesian product.

```
<E, C> App<E, Long> cart(ExecStreamAlg<E, C> alg) {
  return alg.reduce(0L, Long::sum,
            alg.flatMap(
              x -> alg.map(y -> x * y,
                     alg.source(v2)),
              alg.source(v1)));
}
```

The above can be used with any of the various semantics factories presented in Section 3, depending on the kind of evaluation the user wants to perform. The first five expressions below return a scalar value `Long` and the last two a `Future<Long>`.

Listing 5.3: Examples

```
Id.prj(cart(new ExecPushFactory())).value;
Id.prj(cart(new ExecPullFactory())).value;
Id.prj(cart(new ExecFusedPullFactory())).value;
Id.prj(cart(new LogFactory<>(new ExecPushFactory()))).value;
Id.prj(cart(new LogFactory<>(new ExecPushFactory()))).value
Future.prj(cart(new ExecFutureFactory<>(new ExecPushFactory())));
Future.prj(cart(new ExecFutureFactory<>(new ExecPullFactory())));
```

6 Performance

It is interesting to assess the performance of our approach, compared to the highly optimized Java 8 streams. Since our techniques add an extra layer of abstraction, one may suspect them for inefficiency. However, there are excellent reasons why our design is actually good for performance:

- Object algebras are used merely for pipeline construction and not for execution. Once the data processing loop starts, it should be as efficient as in standard Java streams.
- Our design offers fully pluggable semantics. This is advantageous for performance. We can leverage fusion of combinators, proper pull-style iteration without materialization of full intermediate results, and more.

Our benchmarks aim to showcase these two aspects. In this sense, some of the benchmarks are unfair to Java 8 streams: they explicitly target cases for which we can optimize better. We point out when this is the case.

We use a set of microbenchmarks offering various combinations of streaming pipelines:

- **reduce**: a sum operation.

- **filter/reduce**: a filter-sum pipeline.
- **filter/map/reduce**: a filter-map-sum pipeline.
- **cart/reduce**: a nested pipeline with a `flatMap` and an inner operation, with a `map` (capturing a variable), to encode the sum of a Cartesian product.
- **fused filters**: 8 consecutive filter operations and a count terminal operation. The implementation is push-style for Java 8, push-style, pull-style & fused for our library.
- **fused maps**: 8 consecutive map operations and a count terminal operation. The implementation is push-style for Java 8, push-style, pull-style & fused for our library.
- **count**: a count operation (pull-style).
- **filter/count**: a filter-count pipeline (pull-style).
- **filter/map/count**: a filter-map-count pipeline (pull-style).
- **cart/take/count**: a nested pipeline with a `flatMap` and an inner operation, with a `map`, to encode taking the first few elements of a Cartesian product and then counting them (pull-style).

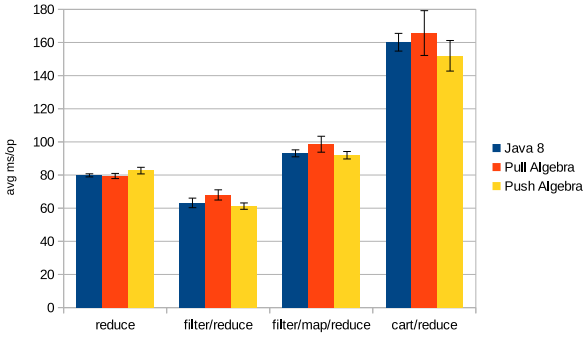
Although our library is not yet full-featured, it faithfully (relative to Java 8 streams) implements the facilities tested in these benchmarks.

Input: All tests were run with the same input set. For all benchmarks except **cart/reduce** and **cart/take/count** we used an array of $N = 10,000,000$ `Long` integers (boxed),[4] produced by N integers with a `range` function that fills the arrays procedurally. The **cart/reduce** test iterates over two arrays. An outer one of $10,000,000$ long integers and an inner one of 10. For the **cart/take/count** test, the sizes of the inner and outer arrays are reversed and the `take` operator draws only the first $n = 100,000$ elements. Fusion operations use $1,000,000$ long integers.

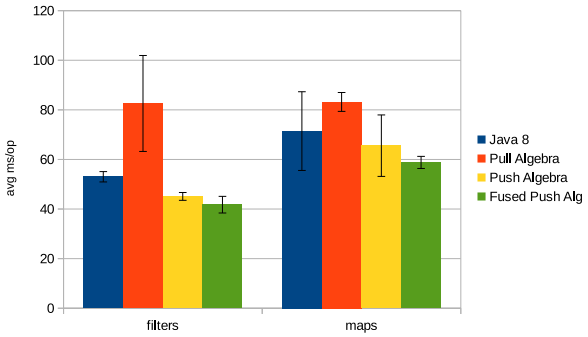
Setup: We use a Fedora Linux x64 operating system (version 3.17.4-200.fc20) that runs natively on an Intel Core i5-3360M vPro 2.8GHz CPU (2 physical x 2 logical cores). The total memory of the system is 4GB.

Automation: We used the Java Microbenchmark Harness (JMH) [16]: a benchmarking tool for JVM-based languages that is part of the OpenJDK. JMH is an annotation-based tool and takes care of all intrinsic details of the execution process, in order to remove common experimental biases. The JVM performs JIT compilation so the benchmark author must measure execution time after a certain warm-up period to wait for transient responses to settle down. JMH offers an easy API to achieve that. In our benchmarks we employed 10 warm-up iterations and 10 proper iterations. We also force garbage collection before benchmark execution. Additionally, we used 2 VM-forks for all tests, to measure potential run-to-run variance. We have fixed the heap size to 3GB for the JVM to avoid heap resizing effects during execution.

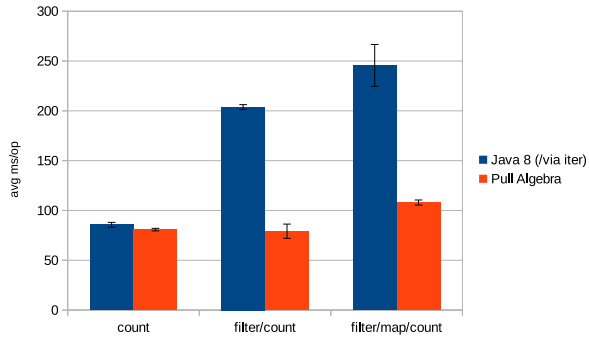
[4] Specialized pipelines for primitive types is not supported in our library, but should be a valuable future engineering addition.



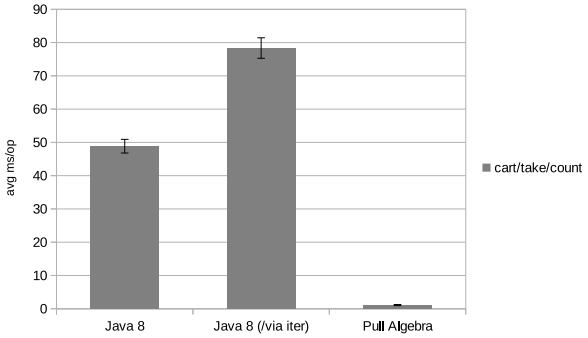
(a) Basic pipelines.



(b) Fusible pipelines.



(c) Pull based reduction.



(d) Pull & Push based flatMap/take.

Fig. 1: Microbenchmarks on JVM in milliseconds / iteration (average of 10).

Results: The benchmarks are cleanly grouped in 4 sets:

- In Figure 1a we present the results of the first 4 benchmarks: **reduce**, **filter/reduce**, **filter/map/reduce**, and **cart/reduce**. These are "fair" comparisons, of completely equivalent functionality in the two libraries. As can be seen, the performance of our push algebra implementation matches or exceeds that of Java 8, validating the claim that our approach does not incur undue overheads.
- Figure 1b presents the results for the next two benchmarks: **fused filters** and **fused maps**. These benchmarks are intended to demonstrate the improvement from our fusing semantics. The Java 8 implementation compared is push-style. Still, our fused pull-style semantics yield a successful optimization, outperforming even the efficient, push-style iteration. Due to our design, this optimization is achieved modularly and independently of the rest of the stream implementation.
- Figure 1c includes the next 3 benchmarks: **count**, **filter/count**, and **filter/map/count**. These are benchmarks of semantically uneven implementations. Java 8 streams support pull-style functionality by transforming the stream into an iterator, but this is not equivalent to full pull-style iteration. As can be seen, a true pull semantics for all operators can be much faster.
- Finally in Fig 1d we show the **cart/take/count** benchmark. This contains a pipeline that is pathological for the Java pull-style streams, in much the same way as the infinite evaluation of Section 3.1. (Push-style streams have the same pathology, by definition: they cannot exploit the fact that only a small number of results are needed in the final operator.) Instead of an infinite stream we reproduce the nested `flatMap`/`map` pipeline but with the larger array being the nested one. As `flatMap` needs to materialize nested arrays (effectively applying the inner `map` function to create the inner stream) it suffers from the effect of inner allocations. Our proper pull-style pipeline does not present this behavior. The result is spectacular in favor of our pull algebra implementation, because of the small number of elements actually needed by the `take` operator.

7 Discussion

We next present observations related to our library design and its constituent elements.

7.1 Fluent API

Object Algebras drive the "interpretation" of streams in our work, so a nested, reversed pattern occurs when declaring the combinators of a pipeline: instead of "`of(...).filter(...).count()`", our pipeline looks like "`alg.count(alg.filter(..., alg.source(...)))`". This reversed pattern follows the declaration order of the combinators, contradicting the natural ordering of a fluent API.

We have created an experimental fluent API in Java using static methods in the interface of the object algebra,[5] but the result was cumbersome. In contrast, in C# the user can create a fluent API through the use of extension methods. Extension methods enable the user to "add" methods to *existing types* without creating a new derived type, recompiling, or modifying the original type. Extension methods are simply a compiler shorthand that enables static methods to be called with instance syntax. Using that feature, the user can create a static class enclosing extension methods that capture the reversed flow.

We created a skeletal implementation of our library design for C#—given its object algebra architecture, our approach is applicable to any modern OO language with generics. In Listing 7.1 the user wraps all methods of an algebra with methods that, instead of returning `C<T>`, return a function that takes as parameter an algebra object. All extension methods are defined for the aforementioned function type, `Func<F, App<C, T>>`.

Listing 7.1: Example of Fluent API creation in C#.

```
static class Stream {
  public static Func<F, App<C, T>>
  OfArray<F, C, T>(T[] array) where F : IStreamAlg<C>
  { return alg => alg.OfArray(array); }

  public static Func<F, App<C, T>>
  Filter<F, C, T>(this Func<F, App<C, T>> streamF,
                       Func<T, bool> predicate) where F : IStreamAlg<C>
  { return alg => alg.Filter(streamF(alg), predicate); }

  public static Func<F, App<E, int>>
  Count<F, E, C, T>(this Func<F, App<C, T>> streamF)
  where F : IExecStreamAlg<E, C>
  { return alg => alg.Count(streamF(alg)); }

  public static App<E, int> Example<E, C, F>(int data, F alg)
  where F : IExecStreamAlgebra<E, C> {
    Func<F, App<E, int>> streamF =
          Stream.OfArray(data)
                .Filter(x => x % 2 == 0)
                .Count();

    return streamF(alg);
  }
}
```

This enables fluent ordering, as shown in the `Example` method of the listing. The fluent API also has immediate side benefits in current programming tools: the user is able to retrieve the list of combinators via the intelligent code completion feature of the IDE.

[5] Static methods in interfaces is a feature introduced in Java 8.

7.2 Generalized Algebraic Data Types

We observe that the encoding of type-constructor parameterization that we employed in Section 4 is sufficient for emulating Generalized Algebraic Data Types (GADTs) in Java. GADTs are algebraic data types that permit data constructors to specify their exact return type. The standard example of a GADT is a generic expression evaluator, which can be captured via an abstract visitor that uses type-constructor polymorphism:

```
abstract class Visitor<R<_>> {
  abstract     R <Integer> caseIntLit (IntLit expr);
  abstract     R <Boolean> caseBoolLit (BoolLit expr);
  abstract     R <Integer> casePlus (Plus expr);
  abstract <Y> R <Y>       caseIf (If<Y> expr);
}
```

The emulation of GADTs with type-constructor polymorphism is not new—it is, for instance, mentioned by Altherr and Cremet [1] and by Oliveira and Cook [13].

As discussed in Section 4 the safety of the emulation depends on following the convention that the only subtype of `App<S.t, ...>` is `S`.

8 Related Work

"Stream programming" is an overloaded term, encompassing streaming algorithms (a subarea of the theory of algorithms), synchronous dataflow, reactive systems, signal processing applications, spreadsheets, and embedded systems [4, 18, 22]. The context of our work is streams in the sense of the Java Stream API [15], which provides stream-like functionality for data collections. In the rest of this section, we discuss related work both from dedicated streaming systems, and from streaming APIs built on top of general-purpose programming languages.

Streaming DSLs and interpreters. From the implementor's point of view, our stream algebras expose a DSL for stream programming. The DSL is embedded in Java and takes advantage of the optimizing nature of the underlying JIT-based implementation. In a similar fashion, the StreamIt language, which needed a special implementation with stream-specific analyses and optimizations [21], was recently implemented atop the Java platform, as StreamJIT [3]. However, while StreamJIT required a full implementation effort, our technique is more lightweight, being available as a Java library, with an API extending that of native Java streams. StreamIt generally has a very different focus from our work: although it offers flexible, declarative combinators, its domain of applicability is multimedia streams of very large data, as opposed to general functional programming over data collections.

The DirectFlow DSL of Lin and Black also supported push and pull configurations for information-flow systems with extensible operators [9]. Compared to

our design, it uses a compiler, exposes the internal "pipes" that connect different stream operators, and requires the management (instantiation and connection) of objects for these pieces of the flow graph.

The operator fusion semantics that we showed is only one example of stream-based optimizations. Other optimizations that can be unlocked by our technique are operator reordering, redundancy elimination in the pipeline, and batching [8].

Our DSL representation follows a shallow embedding with Church-style encodings [13]; as Gibbons and Wu have demonstrated, this makes it natural and simple to implement the interpreter in recursive style [5].

Our design permits not only the creation of completely new operators, but also composite operators that reuse existing ones. This is another way of constructing sub-graphs of the streaming application, a feature common with high-level streaming languages such as SPL [7].

Collections and big data (including Java streams). Su *et al.* showed how Java streams can support different compute engines in the same pipeline [19], for the domain of distributed data sets. Unlike our design, there is no infrastructure for the change of evaluation engine without affecting the library code.

Our approach can process the pipeline, so in that respect is similar to the "application DSL" of ScalaPipe [24]. In contrast to ScalaPipe's separation between application and block DSLs, we show how to do both the pipeline processing and the operations programming in Java code.

StreamFlex offers high-throughput, low-latency streaming capabilities in Java, taking advantage of ownership types [17]. It follows a different API design and requires changes to the Java Virtual Machine, while our design follows the Stream API without needing additional infrastructure.

Svensson and Svenningsson demonstrated how pull-style and push-style array semantics can be combined in a single API using a defunctionalized representation and a shallow embedding for their DSL [20]. However, they propose a new API and a separate DSL layer that passes through a compiler, while we remain compatible with existing Java-like stream APIs. Furthermore, our approach enables full semantic extensibility, beyond just changing the pull vs. pull style of iteration.

9 Future Work and Conclusions

We presented an alternative design for streaming libraries, based on an object algebras architecture. Our design requires only standard features of generics and is, thus, widely applicable to modern OO languages, such as Java, Scala, and C#. We implemented a Java streaming library based on these principles and showed its significant benefits, in terms of transparent semantic extensibility, without sacrificing performance.

Given our extensible library design, there are several avenues for further work. The clearest path is towards enriching the current library implementation with shared-memory parallel evaluation semantics, cloud evaluation semantics,

distributed pipeline parallelism, GPU processing, and more. Since we expose the streaming pipeline, such additions should be transparent to current evaluation semantics, and can even be performed by third-party programmers.

References

1. Altherr, P., Cremet, V.: Adding type constructor parameterization to Java. In: Workshop on Formal Techniques for Java-like Programs (FTfJP'07) at the European Conference on Object-Oriented Programming (ECOOP) (2007)
2. Biboudis, A., Palladinos, N., Smaragdakis, Y.: Clash of the lambdas. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), available at http://arxiv.org/abs/1406.6631 (2014)
3. Bosboom, J., Rajadurai, S., Wong, W.F., Amarasinghe, S.: StreamJIT: A commensal compiler for high-performance stream programming. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 177–195. OOPSLA '14, ACM, New York, NY, USA (2014)
4. Choi, Y., Lin, Y., Chong, N., Mahlke, S., Mudge, T.: Stream compilation for real-time embedded multicore systems. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 210–220. CGO '09, IEEE Computer Society, Washington, DC, USA (2009)
5. Gibbons, J., Wu, N.: Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 339–347. ICFP '14, ACM, New York, NY, USA (2014)
6. Gronau, D.: HighJ - Haskell-style type classes in Java. Google project hosting (2011), `https://code.google.com/p/highj/`
7. Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M., Nasgaard, H., Schneider, S., Soulé, R., Wu, K.L.: IBM Streams Processing Language: Analyzing big data in motion. IBM Journal of Research and Development 57(3-4), 7:1–7:11 (May 2013)
8. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Computing Surveys 46(4), 46:1–46:34 (Mar 2014)
9. Lin, C.k., Black, A.P.: DirectFlow: A domain-specific language for information-flow systems. In: Proceedings of the 21st European Conference on Object-Oriented Programming. pp. 299–322. ECOOP'07, Springer-Verlag, Berlin, Heidelberg (2007)
10. Magi, S.: Abstracting over type constructors using dynamics in C#. `http://higherlogics.blogspot.ca/2009/10/abstracting-over-type-constructors.html` (Oct 2009)
11. Meijer, E.: The world according to LINQ. Communications of the ACM 54(10), 45–51 (Oct 2011)
12. Meijer, E., Beckman, B., Bierman, G.: LINQ: Reconciling Object, Relations and XML in the .NET Framework. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. pp. 706–706. SIGMOD '06, ACM, New York, NY, USA (2006)
13. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: Proceedings of the 26th European Conference on Object-Oriented Programming, pp. 2–27. ECOOP'12, Springer-Verlag, Berlin, Heidelberg (2012)

14. Oliveira, B.C.d.S., Storm, T.v.d., Loh, A., Cook, W.R.: Feature-oriented programming with object algebras. In: Castagna, G. (ed.) ECOOP 2013 – Object-Oriented Programming, pp. 27–51. No. 7920 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2013)
15. Oracle: Stream (Java Platform SE 8), `http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html`
16. Shipilev, A., Kuksenko, S., Astrand, A., Friberg, S., Loef, H.: OpenJDK: jmh, `http://openjdk.java.net/projects/code-tools/jmh/`
17. Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: StreamFlex: High-throughput stream programming in Java. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 211–228. OOPSLA '07, ACM, New York, NY, USA (2007)
18. Stephens, R.: A survey of stream processing. Acta Informatica 34(7), 491–541 (1997)
19. Su, X., Swart, G., Goetz, B., Oliver, B., Sandoz, P.: Changing engines in midstream: A Java stream computational model for big data processing. In: Proceedings of the VLDB Endowment. vol. 7, pp. 1343–1354 (2014)
20. Svensson, B.J., Svenningsson, J.: Defunctionalizing push arrays. In: Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing. pp. 43–52. ACM (2014)
21. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Proceedings of the 11th International Conference on Compiler Construction. pp. 179–196. CC '02, Springer-Verlag, London, UK, UK (2002)
22. Vaziri, M., Tardieu, O., Rabbah, R., Suter, P., Hirzel, M.: Stream processing with a spreadsheet. In: Jones, R. (ed.) ECOOP 2014 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 8586, pp. 360–384. Springer Berlin Heidelberg (2014)
23. Wadler, P.: The Expression Problem (Dec 1998), `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`
24. Wingbermuehle, J.G., Chamberlain, R.D., Cytron, R.K.: ScalaPipe: A streaming application generator. In: Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines. pp. 244–. FCCM '12, IEEE Computer Society, Washington, DC, USA (2012)
25. Yallop, J., White, L.: Lightweight higher-kinded polymorphism. In: Functional and Logic Programming, pp. 119–135. Springer (2014)