Deconstructing *yield* operator to enhance streams processing

Diogo Poeira[1] and Fernando Miguel Carvalho[1][a]
[1] *CCISEL, cc.isel.pt, Polytechnic Institute of Lisbon, Portugal*
mcarvalho@cc.isel.ipl.pt, dpoeira@cc.isel.ipl.pt

Keywords: Yield, Generators, Streams, Lazy Sequences, Iterators, Extensions

Abstract: Customizing streams pipelines with new user-defined operations is a well-known pattern regarding streams processing. However, programming languages face two challenges when considering streams extensibility: 1) provide a compact and readable way to express new operations, and 2) keep streams' laziness behavior. From here, we may find a consensus around the adoption of the generator operator, i.e. *yield*, as a means to fulfil both requirements, since most state-of-the-art programming languages provide this feature. Also, previous work has already stated a formal model for a generalized *yield* operator, that unifies different technological implementations. Yet, what is the performance overhead of interleaving a *yield*-based operation in streams processing? In this work we present a benchmark based on realistic use cases of two different web APIs, namely: Last.fm and world weather online, where custom *yield*-based operations may degrade the streams performance in twofold. We also propose a purely functional and minimalistic design, named *tinyield*, that can be easily adopted in any programming language and provides a concise way of chaining extension operations fluently, with low overhead in the evaluated benchmarks. The *tinyield* proposal was deployed in three different libraries, namely for Java (*jayield*), JavaScript (*tinyield4ts*) and .Net (*tinyield4net*).

1 INTRODUCTION

Lazy evaluation was a well-known technique introduced with lazy lists in Lisp in 1976 (Friedman and Wise, 1976). Yet, its straightforward application to object-oriented languages gave rise to ad hoc iterator classes, that increase substantially their implementations in complexity and verbosity (Baker, 1993).

The use of the generator operator (i.e. *yield*) to implement streams suppresses the aforementioned problem and was widely adopted by mainstream programming languages (with the exception of Java). The *yield* operator allows programmers to develop user-defined operations on streams in a compact manner, while still preserving their laziness property.

Simply put, a generator is like a function that generates a sequence of values. However, instead of building a sequence at once (e.g. array or vector), a generator yields the values one at a time, i.e. it returns a "new" value every time it is called. This idea was first introduced in CLU programming language (Liskov, 1983), but its recent popularity may be attributed to its use first in C# 2.0 (Borins et al., 2006) and later in Ruby 1.9 (Thomas and Hunt, 2007). In CLU and C#, generators are known as iterators, and in Ruby, enumerators. Also, Python, Php, JavaScript, Scala, Dart and Kotlin provide variants of the *yield* operator.

Despite all the advantages of using the *yield* operator, we observe an emerging offer of alternative streams libraries over the standard libraries of every programming environment. And, with those libraries also come distinct extensibility approaches, namely in Java that lacks the *yield* operator. This panoply of libraries includes:

- Java: Guava, Protonpack, Vavr, Eclipse Collections, jOOλ and StreamEx.
- JavaScript: IxJs, LazyJs, Lodash, Sequency and Underscore.
- Dotnet: Cister.ValueLinq, LinqFaster, LinqAF, StructLinq and Hyperlinq.

Given that, how should we elect an auxiliary library to our project?

Not only did we find a lack of benchmarks that assess the effectiveness of each alternative, but also the evaluated workloads have little in common with real use cases.

This work aims to answer the questions and problems stated in this Introduction, and more specifically, the main contributions of this paper are:

[a] https://orcid.org/0000-0002-4281-3195

- A novel benchmark that merges state-of-the-art toolkits such as kotlin-benchmarks (Ryzhenkov, 2014) and JMH (Shipilev, 2013), with the idea of processing streams from realistic data sources, interleaved with user-defined operations.
- A minimalist and functional design of generators, named *tinyield* (Poeira and Carvalho, 2020b) that is the first proposal to unify a generalized *yield* model only focused on traversal and not supported by co-routines as proposed in previous works(James and Sabry, 2011; Prokopec and Liu, 2018).
- *tinyield* outperforms both state-of-the-art libraries and custom *yield*-based operations (Poeira and Carvalho, 2020a).
- Also, *tinyield* allows verboseless and *fluent* extensibility. It provides concise extension of streams operations in an equivalent idiom to *yield*-based generators, without requiring compiler instrumentation support. And, those user-defined extensions can be *fluently chained* in streams pipelines (Fowler, 2015).

The remainder of this paper is organized as follows. In the next section we establish the terminology and we propose a generalized API level design of *yield*, named *tinyield*(Poeira and Carvalho, 2020b). After that, Section 3 explains what tests were devised to analyze the sequence alternatives and discuss the results of the benchmarks. Section 4 describes the related work and existing alternative libraries. Finally we conclude in Section 5 and discuss some future work.

2 *yield* GENERALIZED DESIGN

The variants of *yield* operator are beyond the scope of this paper and we are only establishing a common terminology according to its formal model (James and Sabry, 2011).

After that, we will present our proposal of a generalized design of *yield* that can be implemented in any programming language with higher-order functions support.

2.1 *yield* generator operator

We will dictate the *yield* operator using JavaScript (ecm, 2020) as the lingua franca to focus on the relevant properties that are shared among different programming languages. JavaScript is largely based on well-established C language syntax and influenced by Scheme features. In our examples, we will avoid specific JavaScript particularities and we mostly use its generalized keywords and operators common to other languages, such as: `for`, `var`, `[]`, `!=`, `++`, `<<`, and others.

The generator operator *yield* is inspired by the coroutine primitive yield. In coroutines, the *yield* provides a means of suspending a computation, so that execution can be resumed later (Conway, 1963). In the same way, the term generator (or iterator) refers to a computation that: 1) yields values to the caller and, 2) is resumed after the yielded value has been consumed by the caller (Liskov, 1996). Like a coroutine, the caller must interact with the generator by reading the yielded values and resuming.

To exemplify the yield semantics in the context of generators, we will start with a generator of a sequence of Cullen numbers defined by $C_n = n \cdot 2^n + 1$ and implemented in JavaScript according to Listing 1.

```
function* cullen() {
    for (var i = 0; true; i++)
        yield  (1 << i) * i + 1
}
```

Listing 1: Javascript generator of Cullen numbers.

Terminology. We use the term ***generator*** to refer to computations that *yield* values. Only generator functions can use the *yield* keyword. A free *yield* results in a compiler error. Finally, the argument to the *yield* operator becomes an *output* of the generator. We refer to these outputs as *yielded values*.

In this sense, the function `cullen` of Listing 1 is a *generator*. Notice, that in JavaScript a generator differentiates from a regular function by the suffix character `*`, which indicates that it yields a sequence of values (potentially infinite). On the other hand, in strongly typed languages, a generator may be identified by the function's returned type (e.g. `IEnumerable` in C#).

Traversals allow composing separately written generators. It must be possible for one generator to call into another generator and retain the same yielding context.

Consider for example a `map` with closed addressing, which consists of a hash table whose entries are arrays of elements. We would like a generator `flatten` that traverses the elements of the `map`. Given the separately implemented `list` generator of Listing 2, which yields from an array, it is handy that the `flatten` generator of Listing 3 can reuse this existing functionality by passing arrays from which to *yield*. The `yield*` expression is used to delegate to another generator. It iterates over the operand and yields each value returned by it.

```
function* list(items) {
  for (const value of items)
    yield value
}
```

Listing 2: *yield*-based implementation of `list` generator

```
function* flatten(map) {
  const es = Object.values(map)
  for (const e of es)
    yield* list(e)
}
```

Listing 3: *yield*-based implementation of `flatten` generator that combines the use of `list`.

This requirement is equivalent to that stated by a monad combinator, where given a type constructor M that builds up a monadic type MT and a monadic function such as $T \rightarrow MU$, we have:

$(MT, T \rightarrow MU) \rightarrow MU$

This is the same behavior of `yield* list(e)`. Given the generator `list` of type $MElement$, then each entry of the `map` is $MArray$ that is unwrapped in $MElement$.

2.2 *Tinyield* design

We choose the .Net Type System (CTS, 2012) to specify the *tinyield* types design, because it has support for first-class *function types*. Notice, for example in Java, function types are defined by interfaces that may mislead their real purpose. According to .Net type system, every *function type* has an `Invoke` method that conforms to its descriptor, i.e. type of the arguments and return type.

The *tinyield* generator is based on the `Traverser` *function type* that specifies how the elements of a sequence are traversed. A `Traverser` corresponds to a delimited subroutine that marks the boundary of a generator and delimits the action of *yield*. The argument of `Traverser` function is an opaque computation that can yield. This immediately suggests a monadic encapsulation for the effectful generator computations with *yield* as the only effect operator of the monad. Since `Traverser` marks the boundary of this effect, it can be used as the operation that escapes the monad.

Notice that in C#, Ruby and JavaScript, the equivalent to `Traverser` is hidden in the implementation of the loop construct, that is the `for( of )` statement of Listings 2 and 3.

In Figure 1 we depicted the types design of `Traverser` and `Yield`, which are implemented in the three distributions of the *tinyield* (Poeira and Carvalho, 2020b), in Java, C# and Typescript (a strict syntactical superset of JavaScript).

The generator parameters are not identified in Figure 1 and are captured by the traverser lexical scope from the generator function (*closure*). In Listings 4 and 5 we present the corresponding implementations of generators `list` and `flatten` according to *tinyield* types `Traverser` and `Yield` defined in C#.

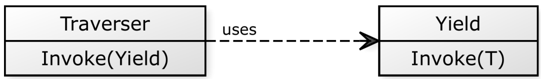

Figure 1: Class diagram of `Traverser` and `Yield` types.

```
Traverser<T> list(... items) {
  return yield => {
    foreach (T value in items)
      yield(value);
  };
}
```

Listing 4: *tinyield* based implementation of `list` generator

```
Traverser<T> flatten(IEnumerable<T> map) {
  return yield => {
    foreach(var entry in map.Values)
      list(entry)(yield);
  };
}
```

Listing 5: *tinyield* based implementation of `flatten` generator

Each lambda (i.e. `=>`) returned by each function encloses the generator boundary that captures the generator parameters (i.e. `items` and `map`).

These implementations do not require any compiler instrumentation support since we do not use any kind of special primitive, like *yield*. Notice that in Listing 4 and 5, `yield` is of type `Yield<T>` and it is the argument of the `Traverser`. This `Yield<T>` instance encloses the context that can be preserved across different generators' calls, e.g. `list(entry)(yield)`, complying to the *stackful* property stated in Section 2.3. The call to `Invoke` is implicit in `list(entry)(yield)`, which is a simplification for `list(entry).Invoke(yield)`.

We only take advantage of higher-order functions and the ability to define local functions (i.e. lambdas), which are closed over their free lexical variables (i.e. *closures*) (Landin, 1964).

The other difference from the *tinyield* proposal to the JavaScript generator is that the resulting sequence from the JavaScript generator may be traversed with a `for( of )` loop whereas the resulting *tinyield* `Traverser` cannot be traversed with the equivalent C# `foreach( in )`. The `Traverser` can be traversed only through its invocation, for example as presented in Listing 6. The difference between the two forms of traversing is usually denoted as *pull* versus *push* access, where *pull* denotes getting items (*ask*) and *push* regards expressing what to do with those items (*tell*) (Hunt and Thomas, 2003)

```
flatten(map)(Console.WriteLine);
```

Listing 6: Traversing elements from a `Traverser` in a *push* style idiom.

Yet, the *tinyield* `Traverser` has a limitation regarding the *yield* primitive: a suspended `Traverser` is not a first-class value. A `Traverser` performs a single *bulk* computation. Hence, the caller relinquishes control, and many algorithms cannot do this, such as any algorithm that needs to manipulate two sequences simultaneously. For example, the *zip* (also known as convolution) is an operation that takes a tuple of sequences and transforms them into a sequence of tuples. This problem is easily solved if we convert at least one of the sequences to an explicit data structure. Yet, there is a useless overhead in case of that sequence being very large and the streams do not match. Hence, we will have performed a great deal of work for nothing.

Thus, we need a suspendable traversal that is able to iterate element by element, rather than all elements in *bulk*. To that end we have designed an alternative way of traversing elements individually, which is specified by the *tinyield* `Advancer` function type depicted in Figure 2.

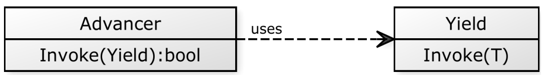

Figure 2: Class diagram of `Advancer` and `Yield` types.

The `Advancer` is similar to the `Traverser` descriptor but returns a Boolean instead (i.e. `bool`). An `Advancer` function is expected to *yield* the next element of the sequence, if there are any, and returns whether an element was processed, or not. Simply put, it essentially merges the behavior of `hasNext()` and `next()` of Java `Iterator` interface in a single subroutine.

To traverse all elements of an `Advancer` we need to perform a `foreach( in )` loop, as presented in the next statement that traverses an hypothetical `Advancer<T> adv` and prints all its elements:

```
while(adv(Console.WriteLine)) { }
```

In Listing 7, we present an `Advancer` based implementation of a `zip( upstream, other, zipper)` that applies the specified `zipper` function to the corresponding elements between `upstream` and `other`, producing a new sequence of results. Both `upstream` and `other` are of type `Advancer`. Notice that the call to `other()` produces a Boolean value according to the `Advancer` idiom. We use this value to let the resulting `Advancer` inform whether it has yielded a value, or not. Also, when the `upstream` is empty the inner *lambda* is not performed and the variable `yielded` remains false. So, only when we successful advance on both streams, the *zip* produces a new value and the variable `yielded` is changed to `true`.

```
Advancer<R> Zip<T, U, R>(
  Advancer<T> upstream,
  Advancer<U> other,
  Func<T, U, R> zipper)
{
  return yield => {
    bool yielded = false;
    upstream(e1 =>
      yielded = other(e2 =>
        yield(zipper(e1, e2))
      )
    );
    return yielded;
  }
}
```

Listing 7: Zip operation for `Advancer` based sequences.

Our `Advancer` based implementation is much more compact than its equivalent counterpart in Java. For example, the accepted answer to the question ”*Zipping streams using JDK8 with lambda*”[1] gives an implementation with more than 30 lines of code. Moreover, our proposal outperforms Java streams in a realistic benchmark zipping sequences from Last.fm (Section 4).

Concluding, and like many others streams libraries, the *tinyield* library provides implementation of core streams processing operations, such as *map*, *filter*, *reduce*, *limit*, *takeWhile*, *zip*, and others (Fowler, 2015). These operations may require one, or both ways of traversal: `Traverser` and `Advancer`. To that end, the *tinyield* type `Query<T>` aggregates the two traversal methods in a single instance. Then, operations are built on top of the `Query<T>` type that allows chaining invocations fluently. In this case, the terminal operation will decide which traversal method to use.

Finally, we should not be restricted to the operations suite provided by a streams library. To that end, we included in *tinyield* a fluent way of chaining user-defined operations. Since, we give priority to `Traverser` type traversal, we provide in `Query` a method `then`, which receives a function that maps an *upstream* `Query` in a new `Traverser`, such that:

```
Then(Func<Query<T>, Traverser<R>> next)
```

Consider for example, that we would like to use an absent `distinctBy` operation to get a sequence of random numbers with distinct lengths of digits. In Listing 8 we show how to implement and chain this new operation fluently in such pipeline.

This is the most concise way of interleaving a user-defined operation. Yet, it fails if the terminal operation requires an `Advancer`, as is the case for *zip*.

[1] stackoverflow.com/a/23529010/1140754

```
Set<int> lengths = new HashSet<>();
Random rand = new Random();
Query
  .Generate(() => rand.next() * MAX)
  .Limit(1024)
  .Map(Convert.ToInt32)
  .Then(upstream => yield => upstream.
      Traverse(
    item => {
      int nrOfDigits = item.Length;
      if (lengths.Add(nrOfDigits))
          yield(item);
  }))
  .Traverse(Console.WriteLine);
```

Listing 8: User-defined `distinctBy` fluently chained in a *tinyield* pipeline.

For those cases, we provide an alternative overloaded `Then` that receives two mapping functions to produce both ways of traversal. However, that alternative will incur in verbosity.

3 PERFORMANCE EVALUATION

To avoid I/O operations during benchmark execution, we have previously collected all data into resource files, loading all that data into in-memory data structures on benchmark bootstrap. Thus, we avoid any I/O by providing the sequences sources from memory. You may find further environment details on *sequences-benchmark* repository (Poeira and Carvalho, 2020a).

To achieve the most unbiased and precise results we relied our benchmarks in state-of-the-art platforms for performance analysis in the three evaluated environments: JMH (Shipilev, 2013) in Java, BenchmarkDotNet (Akinshin, 2019) in .Net and benchmark.js (Bynens and Dalton, 2014) in JavaScript.

We ran our tests on a local machine which has the following specs: Microsoft Windows 10 Home, Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz. We have used the following runtimes: Openjdk 15.0.1 (build 15.0.1+9-18), Dotnet core 5.0.102 and Node.js v12.13.1.

Custom Operation *Every* The custom operation, *Every* is based on the Stackoverflow question on "*how to zip Streams in Java*". The question also discussed how the lack of a zip operation in Java Stream was significant. Our benchmark leveraged some ideas from kotlin-benchmarks (Ryzhenkov, 2014), such as testing with three different categories of types: `Integer`, `String` and a `Value` class holding an `int` and a `String` field that are combined to implement `equals` and `hashCode`.

Every is an operation that, based on a user-defined predicate, tests if all the elements of a sequence match between corresponding positions. To implement the `every()` operation we simply combine the `zip()` and `allMatch()` operations in sequence, such as:

```
seq1
  .zip(seq2,pred::test)
  .allMatch(Boolean.TRUE::equals);
```

The *tinyield* is the most performant in both Java and JavaScript environments as depicted in charts of Figure 3. JavasScript arrays present the most close performance to *tinyield*, but they become unfordable for large data sets due to their eager nature.

Last.fm To benchmark use cases with real-world data, we resorted to publicly available Web APIs, namely REST Countries and Last.fm. We retrieved from REST Countries a list of 250 countries and then used them to query Last.fm, retrieving both the top Artists and the top Tracks by country, resulting in a total of 7500 records each.

The domain model for these benchmarks can be summarized by the entities: `Country`, `Language`, `Track`, and `Artist`.

We devised two benchmarks using data from Last.fm, "*Distinct Top Artist and Top Track by Country*" identified in Figure 4 as "Distinct", and"*Artists Who Are in A Country's Top Ten Who Also Have Tracks in The Same Country's Top Ten*" identified as "*Filter*"(Poeira and Carvalho, 2020a). Both benchmarks start off the same way. We first query all the countries, filter the non-English speaking countries and, from these, we retrieve two sequences: one pairing Country with it's top Tracks and another pairing Country with it's top Artists(Poeira and Carvalho, 2020a).

Tinyield is only overtaken on Last.fm benchmark most significantly on .Net and Java, where *yield*-based implementations perform well, except in JavasScript where the *yield* counterpart used by IxJs is the worst performant in all benchmarks. Nevertheless, *tinyield* is the second most performant library in Java for the Last.fm and with many advantages on extensibility over Java streams (i.e. *verboseless* and *fluency*).

On JavaScript, Lazy.js and Sequency are also well performant alternatives for Last.fm that avoid *yield* primitive as is *tinyield*. Arrays also present good behavior but can be unviable for larger data sets.

Weather and user-defined operations We used another realistic data source from WorldWeatherOnline to benchmark interleaved user-defined operations. For these benchmarks, we created two custom operations: `oddLines` and `collapse`. We then

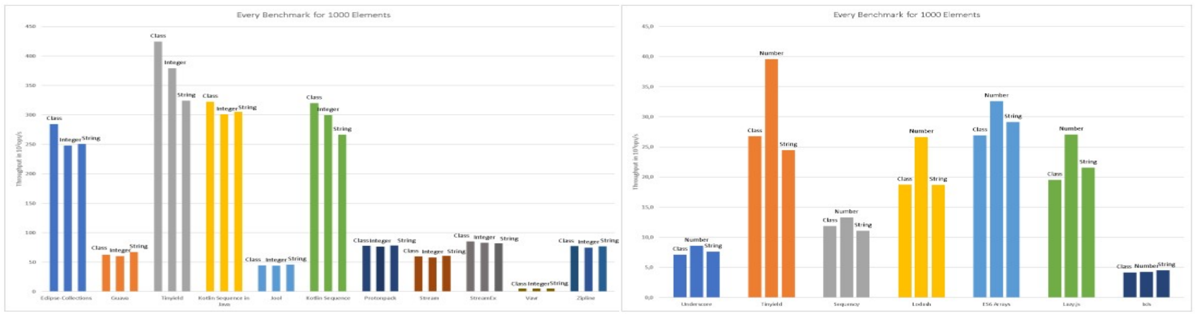

Figure 3: Performance in throughput on Every benchmark.

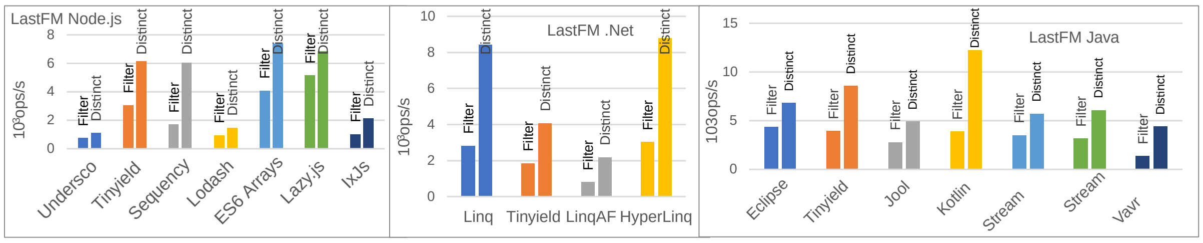

Figure 4: Performance in throughput on Last.fm benchmark.

queried WorldWeatherOnline for the weather in Lisbon, Portugal between the dates of 2020-05-08 and 2020-11-08, providing us with a CSV file that we manipulated with the operations above in a benchmark to perform the following queries: 1) maximum temperature; 2) count distinct temperatures values, and 3) count temperature transitions.

Sequency is a JavaScript library developed in TypeScript like *tinyield4net*. However *tinyield* is between 2 and 3-fold faster than Sequency on weather as depicted in Figure 5.

For comparison, Prokopec (Prokopec and Liu, 2018) also has observed that lazy functional lists are 12-17x slower. We have also experimented that same behavior in Java most significantly on weather benchmark for Vavr, which is a purely functional and immutable-based data structure, and also for StreamEx. Vavr and StreamEx approach to user-defined operations consist on using a *cons* (Friedman and Wise, 1976) in conjunction with the *head* method and a supplier for the new *tail* of the sequence recursively.

4 RELATED WORK

4.1 State-of-the-art

Tinyield performance gains are due to the fast-path iteration protocol that has less overhead when bulk traversing a sequence than a common iterators does. This approach reduces the overhead of per-element access, and increases the effectiveness of other optimizations such as inlining, code motion, bounds check elimination, and others.

Not only that but *tinyield* is the only Java library with a *verboseless* way of defining new stream operations while also maintaining the fluency of the pipeline.

The Java programming language does not provide a *yield* primitive and extending streams API incurs in inevitable verbose implementations.

We identified a few advantages of Kotlin's Sequence, namely on operations that in Java would return `Optional`, return nullable in Kotlin, meaning no wrapper is created resulting in less overhead. Moreover Kotlin's terminal operations are inline so there is no indirection when calling terminal operations.

Eclipse Collections has a lot of optimizations in place regarding the data-source of the pipeline, namely if an array was at the source then iteration will be as fast as using a for loop.

The main gain of StreamEx, jOOλ and Vavr is the fact that these libraries bring extra functionality to the user out of the box with almost no need of creating new user-defined operations.

JavaScript supports, since EcmaScript5 in 2009 (ecm, 2020), operation chaining over sequences, in other words, sequence *pipelines*. JavaScript's sequence type is the Array type, distinguishing itself from other sequence type implementations by having an eager approach. Generators and the *yield* keyword were later introduced with ES6 in 2015, yet, no lazy

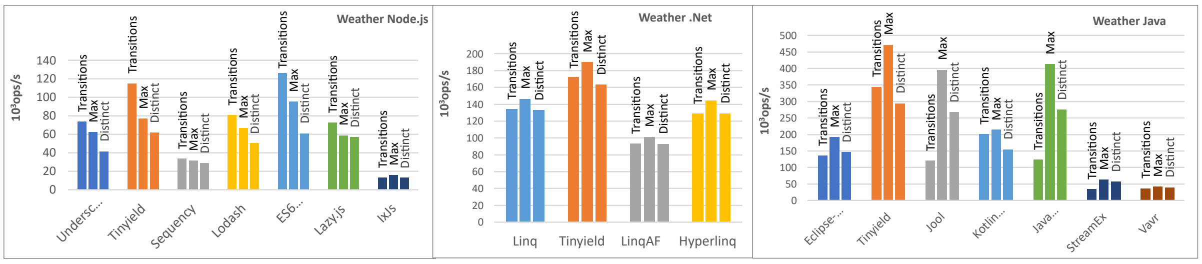

Figure 5: Performance in throughput on weather benchmark.

sequence type implementations were provided by this new standard either forcing developers to look for this feature in third-party libraries.

Lodash and Underscore, although they are quite popular in the Javascript world, they suffer from the same problems of ES6 Arrays. When processing a sequence pipeline these libraries will calculate all intermediate results before proceeding to the next operation, which incurs in the same unnecessary processing observed in ES6 Arrays.

Lazy.js and Sequency propose alternative traversal designs as is *tinyield*, but their proposals have no foundation on a unified *yield* model. Moreover, Sequency is one of the worst performant libraries on weather benchmark.

4.2 Background

Lazy traversal is inspired by the concept of *lazy lists*, also known as *streams*, first described in 1965 by Landin (Landin, 1965). It was Landin who proposed the use of *delayed evaluation* to avoid ”item-by-item” representation of collections. Friedman and Wise (Friedman and Wise, 1976) introduced lazy lists in Lisp in 1976 and the idea was then adopted in other languages too, either as a fundamental data structure, as in Haskell(Jones, 2003).

Alphard, developed at CMU in the late 1970 was the first programming language to introduce the generator operator (Shaw et al., 1977). That construct inspired iterators in CLU (Liskov, 1983) as a procedure that returns a sequence of elements, that allows to get at the elements one at a time.

The idea of a single iteration method was introduced in Python 2.2, where iterators provide a single method `next` that returns the next element in a sequence, or raises an exception when no more elements are available (Yee and van Rossum, 2001). This feature is described in the proposal PEP 234 (Python Enhancement Proposal 234) Iterators (Yee and van Rossum, 2001). The advantages of a single traversal subroutine were highlighted in (Baker, 1993), where H. Baker shows how higher-order functions, taking as argument functions which are closed over their free lexical variables (*closures*) can be used to provide iteration capabilities.

Many use-cases evaluated sequence traversal performance through the use of benchmarks, namely kotlin-benchmarks(Ryzhenkov, 2014), which provides benchmarks over Kotlins features such as the use of `Sequence`. Another example are the benchmarks devised by Angelika Langer and Klaus Kreft (Langer and Kreft, 2015) with the aim of better understanding how Java `Streams` perform and when *parallel()* outperforms sequential processing of the same `Stream`. Nicolai Parlog also tackled this point in his benchmarks on Parallel `Stream` Vectorization (Parlog, 2019), evaluating the performance gained using `Stream` *parallel()* when computing factorials.

5 CONCLUSIONS

Generators are heavily inspired by co-routines, which generally follow two approaches to implement control flow: *call stack manipulation* and program transformation, i.e. *instrumentation*. The main problem with both approaches regards their overheads due to context switch manipulation.

In the first approach, the runtime is augmented with call stack introspection or the ability to swap call stacks during the execution of the program. Several attempts in the context of the JVM runtime (Dragos et al., 2007; Stadler et al., 2009) did not become official and most recently OpenJDK is still working on Project Loom (OpenJDK, 2020) that aims to provide fibers and continuations for the Java Virtual Machine.

In the second approach, the compiler transforms the program to translate coroutines into an equivalent program without coroutines. This is the approach followed by most generators such as in C#, JavaScript, and others.

We claim that leash to co-routines induce heavyweight approaches that incur in performance overheads on generators. On the other hand, object-oriented iterators incur in useless complexity and verbosity that affects readability and expressiveness of streams operations (Baker, 1993).

The *tinyield* design proposal suppresses those limitations with advantages in both performance and extensibility conciseness.

Our model has been already extended for asynchronous processing and has evidences that may overtake alternatives such as *reactive streams* (Kaazing et al., 2017) achieving better throughput under some non-blocking IO scenarios. Previous work has already been made in this field (Prokopec and Liu, 2018) but again, it is tight with co-routines subject, which we have shown in this work that is harmful for streams traversal.

ACKNOWLEDGEMENTS

This work was supported by *Instituto Politecnico de Lisboa*, Lisbon, Portugal, for funding the projet "*Reactive Web streams for Big Data*" (IPL/2020/WebFluid_ISEL).

REFERENCES

(2020). *ECMAScript 2020 language specification, 11th edition*. ECMA, 11 edition.

Akinshin, A. (2019). *Pro .NET Benchmarking: The Art of Performance Measurement*. Apress.

Baker, H. G. (1993). Iterators: Signs of weakness in object-oriented languages. *SIGPLAN OOPS Mess.*, 4(3):18–25.

Borins, M., Braun, A. R., Palmer, R., and Terlson, B. (2006). *ECMA-334 C# language specification*. ECMA, 5 edition.

Bynens, M. and Dalton, J.-D. (2014). benchmarkjs: A benchmarking library that supports high-resolution timer.

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408.

CTS (2012). *ECMA-335 Common Language Infrastructure (CLI), 6th edition, June 2012*. ECMA, 6 edition.

Dragos, I., Cunei, A., and Vitek, J. (2007). Theory and practice of coroutines with snapshots. In *ICOOOLPS'2007*, Technische Universität Berlin.

Fowler, M. (2015). Collection pipeline.

Friedman, D. P. and Wise, D. S. (1976). CONS should not evaluate its arguments. In Michaelson, S. and Milner, R., editors, *Automata, Languages and Programming*, pages 257–284, Edinburgh, Scotland. Edinburgh University Press.

Hunt, A. and Thomas, D. (2003). The art of enbugging. *IEEE SOFTWARE*.

James, R. and Sabry, A. (2011). Yield: Mainstream delimited continuations. *Workshop on the Theory and Practice of Delimited Continuations*.

Jones, S. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Journal of functional programming: Special issue. Cambridge University Press.

Kaazing, Lightbend, Netflix, Pivotal, and Hat, R. (2017). Reactive streams specification for the jvm.

Landin, P. J. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320. ZSCC: 0001262.

Landin, P. J. (1965). Correspondence between algol 60 and church's lambda-notation: Part i. *Commun. ACM*, 8(2):89–101.

Langer, A. and Kreft, K. (2015). Stream performance. JAX London Online Conference.

Liskov, B. (1983). *CLU Reference Manual*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Liskov, B. (1996). *A History of CLU*, page 471–510. Association for Computing Machinery, New York, NY, USA.

OpenJDK (2020). Loom - fibers, continuations and tail-calls for the jvm.

Parlog, N. (2019). Github.

Poeira, D. and Carvalho, F. M. (2020a). Benchmark for different sequence operations in java and kotlin. Technical report, https://github.com/tinyield/sequences-benchmarks.

Poeira, D. and Carvalho, F. M. (2020b). Towards sequence traversal optimization and extensibility. Technical report, https://github.com/tinyield.

Prokopec, A. and Liu, F. (2018). Theory and practice of coroutines with snapshots. In *European Conference on Object-Oriented Programming*.

Ryzhenkov, I. (2014). JetBrains.

Shaw, M., Wulf, W. A., and London, R. L. (1977). Abstraction and verification in alphard: Defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564.

Shipilev, A. (2013). Java microbenchmark harness (the lesser of two evils).

Stadler, L., Wimmer, C., Würthinger, T., Mössenböck, H., and Rose, J. (2009). Lazy continuations for java virtual machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, page 143–152, New York, NY, USA. Association for Computing Machinery.

Thomas, D. and Hunt, A. (2007). *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley.

Yee, K.-P. and van Rossum, G. (2001). Pep 234 – iterators. Technical report, Python.