Clash of the Lambdas

Through the Lens of Streaming APIs

Aggelos Biboudis
University of Athens
biboudis@di.uoa.gr

Nick Palladinos
Nessos Information Technologies, SA
npal@nessos.gr

Yannis Smaragdakis
University of Athens
smaragd@di.uoa.gr

Abstract

The introduction of lambdas in Java 8 completes the slate of statically-typed, mainstream languages with both object-oriented and functional features. The main motivation for lambdas in Java has been to facilitate stream-based declarative APIs, and, therefore, easier parallelism. In this paper, we evaluate the performance impact of lambda abstraction employed in stream processing, for a variety of high-level languages that run on a virtual machine (C#, F#, Java and Scala) and runtime platforms (JVM on Linux and Windows, .NET CLR for Windows, Mono for Linux). Furthermore, we evaluate the performance gain that two optimizing libraries (ScalaBlitz and LinqOptimizer) can offer for C#, F# and Scala. Our study is based on small-scale throughput-benchmarking, with significant care to isolate different factors, consult experts on the systems involved, and identify causes and opportunities. We find that Java exhibits high implementation maturity, which is a dominant factor in benchmarks. At the same time, optimizing frameworks can be highly effective for common query patterns.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Code generation; D.3.2 [*Programming languages*]: Language Classifications—Multiparadigm languages

General Terms Languages, Measurement, Performance

Keywords lambdas, java, scala, c#, f#, query optimization, query languages, declarative

1. Introduction

Java 8 has introduced lambdas with the explicit purpose of enabling streaming abstractions. Such abstractions present an accessible, natural path to multicore parallelism—perhaps the highest value domain in current computing. Other languages, such as Scala, C#, and F#, have supported lambda abstractions and streaming APIs, making them a central theme of their approach to parallelism. Although the specifics of each API differ, there is a core of common features and near-identical best-practices for users of these APIs in different languages.

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. This particular lambda application is *non-capturing*: the bodies of the lambda expressions in lines 3,4 use only their argument values, and no values from the environment.

```
1 public int sumOfSquaresEvenSeq(int[] v) {
2   int sum = IntStream.of(v)
3     .filter(x -> x % 2 == 0)
4     .map(x -> x * x)
5     .sum();
6   return sum;
7 }
```

The above computation can be trivially parallelized with the addition of a `.parallel()` operator before the call to `filter`. This ability showcases the simplicity benefits of streaming abstractions for parallel operations.

In this paper, we perform a comparative study of the lambda+streams APIs of four multi-paradigm, virtual machine-based languages, Java, Scala, C#, and F#, with an emphasis on implementation and performance comparison, across mainstream platforms (JVM on Linux and Windows, .NET CLR for Windows, Mono for Linux). We perform micro-benchmarking[1] and aim to get a high-level understanding of the costs and causes. Our goal is the usual goal of microbenchmarking: to minimize most threats-to-validity by controlling external factors. (The inherent drawback of microbenchmarking, which we do not attempt to address, is the threat that benchmarks are not representative of real uses.) In order to control external factors, we attempt to select equivalent abstractions in all settings, isolate dependencies, employ best-practice benchmarking techniques, and repeatedly consult experts on the different platforms.

Since lambdas+stream operators have arisen independently in so many contexts and have been central in parallel programming strategies, one would expect them to be well-understood: a mainstream, high-value feature is expected to have fairly uniform implementation techniques and trade-offs. Instead we find interesting variation, even in the compilation to intermediate code (per platform, e.g., across Java and Scala, which are both JVM languages). Furthermore, we find that JIT optimization inside the VM does not always interact predictably with the code produced for lambdas. This was a minor surprise, given the maturity of the respective facilities.[2]

A second aspect of declarative streaming operations is that they enable aggressive optimization [7]. Optimization frameworks, such as LinqOptimizer [8] and ScalaBlitz [9, 10], recognize common

[1] Code in https://github.com/biboudis/clashofthelambdas .

[2] Although Java lambdas are standard only as of version 8, their arrival had been forthcoming since at least 2006.

patterns of streaming operations and optimize them, by inlining, performing loop fusion, and more.

In all, we find that Java offers high performance for lambdas and streaming operations, primarily due to optimizing for non-capturing lambdas. At the same time, Java suffers from the lack of an optimizing framework—LinqOptimizer and ScalaBlitz give a significant boost to C#/F# and Scala implementations, respectively, when optimizations are applicable.

2. Implementation Techniques for Lambdas and Streaming

As part of our investigation, we examined current APIs and implementation techniques for lambdas and streaming abstractions in the languages and libraries under study. We detail such elements next, so that we can refer to them directly in our experimental results.

2.1 Programming Languages

We begin with the API and implementation description for the languages of our study: Java, Scala, and C#/F# (the latter are similar enough that are best discussed together, although they exhibit non-negligible performance differences).

2.1.1 Java

Java is probably the best reference point for our study, although it is also the relative newcomer among the lambdas+streaming facilities. We already saw examples of the Java API for streaming in the Introduction. In terms of implementation, the Java language team has chosen a translation scheme for lambdas 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 following expression (serving as a running example in this section) calculates the sum of all values in a double array.

```
1  public double sumOfSquaresSeq(double[] v) {
2    double sum = DoubleStream.of(v)
3    .map(d -> d * d)
4    .sum();
5    return sum;
6  }
```

The code first creates a sequential, ordered `Stream` of `doubles` from an array that holds all values. (`DoubleStream` represents a primitive specialization of `Stream`–one of three specialized `Streams`, together with `IntStream` and `LongStream`.) The calls `map` and `sum` are an intermediate and a terminal operation respectively. The first 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 (non-capturing) lambda function. The second 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 `Spliterators`. The `Spliterator` interface offers an API for traversing and partitioning elements of a source and it can operate either sequentially or in parallel. `Spliterators` are also equiped with more advanced functionality—e.g., they can detect structural interference with the source while processing. The definition of a stream and operations on it are usually described declaratively and the user does not need to invoke operations on a `Spliterator`. However a controlled traversal via a `Spliterator` or even a Java `Iterator` can be effected by obtaining the corresponding instances from the appropriate combinators. The `Spliterator` interface is shown below.

```
1  public interface Spliterator<T> {
2    boolean tryAdvance(Consumer<? super T> action);
3    void forEachRemaining(Consumer<? super T> action);
4    Spliterator<T> trySplit();
5    long estimateSize();
6    long getExactSizeIfKnown();
7    int characteristics();
8    boolean hasCharacteristics(int characteristics);
9    Comparator<? super T> getComparator();
10 }
```

Normally, for the general case of standard stream processing, the implementation of the above interface will have a `forEachRemaining` method that internally calls methods `hasNext` and `next` to traverse 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. In this way, the implementation eliminates two virtual calls per step of iteration and effectively uses internal iteration, instead of external.

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

```
1  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. For instance, the fragment of the `map` implementation below shows the `accept` call (line 7) on the next operator (`sum` in our example). The code also shows the call to `apply`, invoking the passed lambda function.

```
<T, R> Stream<R> map(Stream<T> source,
                     Function<T, R> mapper) {
  return new MapperStream<T, R>(source) {
    Consumer<T> wrap(Consumer<R> consumer) {
      return new Consumer<T>() {
        void accept(T v) {
          consumer.accept(mapper.apply(v));
        }
      };
    }
  };
}
```

Having seen the implementation of streams, we now turn our attention to lambdas. There could be several potential translations for lambdas, such as inner-classes (for both capturing or non-capturing, lambdas), translation based on `MethodHandles`—the dynamic and strongly typed component that was introduced in JSR-292—and more. Each option has some advantages and disadvantages. For the translation of lambdas in Java 8, the compiler incorporates a technique based on JSR-292 [11] and more specifically on the new `invokedynamic` command [4, Chapter 6] and `MethodHandles` [3].

When the compiler encounters a lambda function, it desugars it to a method declaration and emits an `invokedynamic` instruction at that point. For instance, our `sumOfSquaresSeq` example compiles to the bytecode below:

```
1 ...                          // v on the stack
2 invokestatic  #7             // DoubleStream.of
3 invokedynamic #10,  0        // applyAsDouble
4 invokeinterface #11,  2      // DoubleStream.map
5 invokeinterface #8,  1       // DoubleStream.sum
6 dstore_1
7 dload_1
8 dreturn
```

Note the `invokedynamic` instruction on line 3, used to return an object that represents a lambda closure. The method invoked is `LambdaMetafactory.metafactory`—implemented as part of the Java standard library. The fully dynamic nature of the call is due to having a single implementation for retrieving objects for any given method signature. This process involves three phases: *Linkage*, *Capture* and *Invocation*. When `invokedynamic` is met for the first time it must link this site with a method. For the lambda translation case, an instance of `CallSite` is generated whose target knows how to create function objects. This target (`LambdaMetafactory.metafactory`) is a factory for function objects. The Capture phase may involve allocation of a new object that may capture parameters or will always return the same object (if no parameters are captured). The third phase is the actual invocation. The advantage of this translation scheme is that, for lambdas that do not capture any free variables, a single instance for all usages is enough. Furthermore, the call site is linked only once for successive invocations of the lambda and, after that, the JVM inlines the retrieved method's invocation at the dynamic call site. Additionally, there is no performance burden for loading a class from disk, as there would be in the case of a fully static translation.

2.1.2 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. There are two Scala alternatives for our purposes. One is lazy transformations of collections: an approach semantically equivalent to that of other languages, which also avoids the creation of intermediate, allocated results. The other alternative is to use strict collections, which are better supported in the Scala libraries, yet not equivalent to other implementations in our set and suffering from increased memory consumption.

To achieve lazy processing, one has to use the `view` method on a collection.[3] 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. The iteration takes place in the expected, unoptimized order, i.e., each operator has to "request" elements from the one supplying its input, rather than having a "push" pattern, with the producer calling the consumer directly.

The Scala translation is based on synthetic classes that are generated at compile time. For lambdas, Scala generates a class that extends `scala.runtime.AbstractFunction`. For lambdas with free variables (captured from the environment), the generated class includes private member fields that get initialized at instantiation time.

The strict processing of Scala collections is similar to the above lazy idioms from the end-user standpoint: only the `view` call is omitted in our `sumOfSquareSeq` code example. Operators such as `map` are overloaded to also process strict collections.

[3] Scala has more APIs for lazy collections (e.g., "Streams"), but the views API we employed is the exact counterpart, in spirit and functionality, to the machinery in the other languages under study.

2.1.3 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 new major features in C# 3.0 in order to enable a more functional style of programming. These new features, under the umbrella of LINQ [5, 6], can be summarized as support for lambda expressions and function closures, extension methods, anonymous types and special syntax for query comprehensions. All of these new language features 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 have two ways of programming with data streams:

1) as fluent-style method calls

```
nums.Where(x => x % 2 == 0).Select(x => x * x).Sum();
```

2) or with the equivalent query comprehension syntactic sugar

```
(from x in nums
 where x % 2 == 0
 select x * x).Sum();
```

In F#, programming with data is just as simple as a direct pipeline of various combinators.

```
nums  |> Seq.filter (fun x -> x % 2 = 0)
      |> Seq.map (fun x -> x * x)
      |> Seq.sum
```

For the purposes of this discussion, we can consider that both C# and F# have identical operational behaviors and both C# methods (`Select`, `Where`, etc.) and F# combinators (`Seq.map`, `Seq.filter`, etc.) operate on `IEnumerable<T>` objects and return `IEnumerable<T>`.

The `IEnumerable<T>` interface can be thought of as a factory for creating `IEnumerator<T>` objects:

```
interface IEnumerable<T> {
  IEnumerator<T> GetEnumerator();
}
```

and `IEnumerator<T>` is an iterator for an on demand consumption of values:

```
interface IEnumerator<T> {
  // Return current position element
  T Current { get; }
  // Move to next element,
  // returns false if no more elements remain
  bool MoveNext();
}
```

Each of these methods/combinators implement a pair of interfaces called `IEnumerable<T>` / `IEnumerator<T>` and through the composition of these methods a call graph of iterators is chained together. 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. Instead, each operator call is interleaved with each other. As an example we can present an implementation of the `Select` method.

```
static IEnumerable<R> Select<T, R>(IEnumerable<T>
    source, Func<T, R> f) {
  return new SelectEnumerable<T, R>(source, f);
}
```

The `SelectEnumerable` has a simple factory-style implementation:

```
class SelectEnumerable<T, R> : IEnumerable<R> {
  private readonly IEnumerable<T> inner;
  private readonly Func<T, R> func;
  public SelectEnumerable(IEnumerable<T> inner,
                          Func<T, R> func) {
    this.inner = inner;
    this.func = func;
  }
  IEnumerator<R> GetEnumerator() {
    return new SelectEnumerator(inner.GetEnumerator(),
     func);
  }
}
```

`SelectEnumerator` implements the `IEnumerator<R>` interface and delegates the `MoveNext` and `Current` calls to the inner iterator.

```
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); } }
}
```

For programmer convenience, both C# and F# offer support for automatically creating the elaborate scaffolding of the `IEnumerable<T>` / `IEnumerator<T>` interfaces, but for our discussion it is not crucial to understand the mechanisms.

From a performance point of view, it is not difficult to see that there is a lot of 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 generic, unoptimized Java iteration: it is an external iteration, with each consumer asking the producer for the next element.

In terms of lambda translation, C# lambdas are always assigned to delegates, which can be thought of as type-safe function pointers, and in F# lambdas are represented as compiler generated class types that inherit `FSharpFunc`.

```
abstract class FSharpFunc<T, R> {
  abstract R Invoke(T arg);
}
```

In both cases, if a lambda captures free variables, these variables are represented as member fields in a compiler-generated class type.

2.2 Optimizing Frameworks

We next examine two optimizing frameworks for streaming operations: ScalaBlitz and LinqOptimizer.

2.2.1 ScalaBlitz

ScalaBlitz is an open source framework that optimizes Scala collections by applying optimizations for both sequential and parallel computations. It eliminates boxing, performs lambda inlining, loop fusion and specializations to particular data-structures. ScalaBlitz performs optimizations at compile-time based on Scala macros [1].

By enclosing a functional pipeline into an `optimize` block, ScalaBlitz expands in place an optimized version of it:

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

This can be achieved because this library is implemented as a def macro with the following signature:

```
def optimize[T](exp: T): Any = macro optimize_impl[T]
```

The `optimize` block is a function that starts with the additional keyword `macro`. When the compiler encounters an application of the macro `optimize(expression)`, it will expand that application by invoking `optimize_impl`, with the abstract-syntax tree of the functional pipeline expression as argument. The result of the macro implementation is an expanded abstract syntax tree. This tree will be replaced at the call site and will be type-checked.

2.2.2 LinqOptimizer

LinqOptimizer is an open source optimizer for LINQ queries. It compiles declarative queries into fast loop-based imperative code, eliminating virtual calls and temporary heap allocations. LinqOptimizer is a run-time compiler based on LINQ Expression trees, which enable a form of metaprogramming based on type-directed quotations.

In the following example, a lambda expression is assigned to a variable of type `Expression<Func<...>>`.

```
Expression<Func<int, int>> exprf = x => x + 1;
Func<int, int> f = exprf.Compile(); // compile to IL
Console.WriteLine(f(1)); // 2
```

At compile time, the compiler emits code to build an expression tree that represents the lambda expression. LINQ offers library support for runtime manipulation of expression trees (through visitors) and also support for run-time compilation to IL. Using such features, LinqOptimizer lifts queries into the world of expression trees and performs the following optimizations:

1) inlines lambdas and performs loop fusion:

```
var sum = (from num in nums.AsQueryExpr() // lift
           where num % 2 == 0
           select num * num).Sum();
// effectively optimizes to
int sum = 0;
for (int index = 0; index < nums.Length; index++) {
   int num = nums[index];
   if (num % 2 == 0)
      sum += num * num;
}
```

2) for queries with nested structure (`SelectMany`, `flatMap`) applies nested loop generation:

```
var sum = (from num in nums.AsQueryExpr() // lift
           from _num in _nums
           where num % 2 == 0
           select num * _num).Sum();
// effectively optimizes to
int sum = 0;
for (int index = 0; index < nums.Length; index++) {
   for (int _index = 0; _index < _nums.Length;
        _index++) {
      int num = nums[index];
      int _num = _nums[_index];
      if (num % 2 == 0)
         sum += num * _num;
   }
}
```

3. Results

We next discuss our benchmarks and experimental results.

3.1 Microbenchmarks

In this work, we use 4 main microbenchmarks. We focus our efforts on measuring iteration throughput and lambda invocation costs. In all of our benchmarks we produce scalar values as the result of a terminal operation (e.g., instead of producing a transformed list of values), as we do not want to cause memory management effects (e.g., garbage collection). Furthermore, we did not employ sorting or grouping operators, in order to avoid interfering with algorithmic details of library implementations (e.g., mergesort vs quicksort, hash tables vs balanced trees, etc.).

We measure the performance of:

- **sum** iteration speed with no lambdas, just a single iteration.
- **sumOfSquares** a small pipeline with one map operation (i.e., one lambda).
- **sumOfSquaresEven** a bigger pipeline with a filter and map chain of two lambdas.
- **cart** a nested pipeline with a `flatMap` and an inner operation, again with a `flatMap` (capturing a variable), to encode a Cartesian product.

We developed this set for all four languages, Java, Scala, C# and F#, for both sequential and parallel execution. For the latter three we have also included optimized versions using ScalaBlitz and LinqOptimizer. For Scala we also include alternate implementations, which employ more idiomatic strict collections (without the views API). Arguably this approach is better supported in the Scala libraries. Therefore we present separate measurements for Scala-views and Scala-strict tests. In our following analysis, when we do not refer to a Scala-strict test explicitly, Scala-views are implied. Additionally, we include a baseline suite of benchmarks for the sequential cases.

We have run these benchmarks on both Windows and Linux, although Windows is the more universal reference platform for our comparison: it allows us to perform the C#/F# tests on the

industrial-strength implementation of the Microsoft CLR virtual machine.

The purpose of baseline benchmarks is to assess the performance difference between functional pipelines and the corresponding imperative, hand-written equivalents. The imperative examples make use of indexed-based loop iterations in the form of `for`-loops (except for the Scala case in which the while-loop is the analogue of imperative iteration).

Input: All tests were run with the same input set. For the **sum**, **sumOfSquares** and **sumOfSquaresEven** we used an array of $N = 10,000,000$ long integers, produced by N integers with a `range` function. The **cart** test iterates over two arrays. An outer one of $1,000,000$ long integers and an inner one of 10.

The Scala, C# and F# tests were compiled with optimization flags enabled and for Java/Scala tiered compilation was left disabled (C2 JIT compiler only). Additionally, we fixed the heap size to 3GB for the JVM to avoid heap resizing effects during execution.

3.2 Experimental Setup

	Windows	Ubuntu Linux
Version	8.1	13.10/3.11.0-20
Architecture	x64	x64
CPU	Intel Core i5-3360M vPro 2.8GHz	
Cores	2 physical x 2 logical	
Memory	4GB	

Systems: We performed both Linux (see Figure 2) and Windows (see Figure 1) tests natively on the same system via a dual-boot installation.

	Windows	Ubuntu Linux
Java	Java 8 (b132)/JVM 1.8	
Scala	2.10.4/JVM 1.8	
C#	C#5 /CLR v4.0	C# mono 3.4.0.0/mono 3.4.0
F#	F#3.1/CLR v4.0	F# open-source 3.0/mono 3.4.0

Microbenchmarking automation: For Java and Scala benchmarks we used the Java Microbenchmark Harness (JMH) [13] tool: 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. Its goal is to produce as objective results as possible. The JVM performs JIT compilation (we use the C2 JIT compiler) 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. Regarding the CLR, warm-up effects take an infinitesimal amount of time compared to the JVM [14]. The CLR JIT compiler compiles methods exactly once and subsequent method calls invoke directly the JITted version. Code is never recompiled (nor interpreted at any point). For the purpose of benchmarking C#/F# programs, as there is not any widely-used, state-of-the-art tool for microbenchmarking, we created the *LambdaMicrobenchmarking* utility [4] written in C#, according to the common microbenchmarking practices described in [12]. It calculates the average execution of method invocations using the `TimeSpan.TotalMilliseconds` property of the `TimeSpan` structure that converts ticks to whole and fractional milliseconds. Our utility uses the Student-T distribution for statistical inference; mean error and standard deviation. The same distribution is employed in JMH as well. Our utility forces garbage collection between runs. For all tests, we do not measure the time needed to initialize data-structures (filling arrays), and neither the run-time compilation cost of the optimized queries in the LinqOptimizer case nor the compile-time overhead of macro expansion in the ScalaBlitz case.

[4] https://github.com/biboudis/LambdaMicrobenchmarking .

3.3 Performance Evaluation

Languages: Among the languages[5] of our study, Java exhibits by far the best performance, in both sequential and parallel tests, due to its advanced translation scheme. Notably, Java results show not only that three out of four of our tests are very close to baseline measurements but also that the parallel versions scale well. Regarding the parallel versions, all microbenchmarks reveal that even in cases where Java was very close to the baseline, performance increases further achieving parallel speedups of 1.1x-1.6x. For the **cart** benchmark, although Java has the best performance among all streaming implementations, it still pays a considerable cost for inner closures, as can be seen in comparison to the baseline benchmark for the sequential case. During the execution of **cart** the garbage collector was invoked 3 times (per iteration) for the sequential version and 4 times for the parallel version, indicating significant memory management activity. Scala Parallel Collections using the lazy, `view`, API seem to suffer in the parallel tests quite significantly over all other implementations (note that the Y-axis is truncated) due to boxing/unboxing, iterator, and function object abstraction penalties. (For a more detailed analysis, see Section 4.) The strict Scala API (which, although non-equivalent to other implementations is arguably more idiomatic) performed significantly better. Although we present results for a 3GB heap space, we have also conducted the same tests under various constrained heap spaces. In practice, Scala-strict benchmarks ran with about 4x more heap space than their Java counterparts, which is unsurprising given that all strict operators need to generate and process intermediate collections. Still, the parallel Scala/Scala-strict benchmarks were almost always the slowest among all implementations on both Windows and Linux.

In the sequential tests of C# and F# we observe a constant difference in favor of C# for **sumOfSquares**, **sumOfSquaresEven** and a significant difference of 2.7x for the **cart** benchmark on Windows. As `seq<'T>` is a type alias for .NET's `IEnumerable<T>` we conclude that the difference is attributed to different implementations of operators. In the parallel benchmarks, as F# relies on the standard library for .NET, it is driven by its performance. Thus, all parallel benchmarks (Windows and Linux) show these two languages at the same level.

In all cases, the parallel benchmarks of LINQ on mono scaled poorly, revealing poor scaling decisions in the implementation. Additionally, comparing the Windows and Linux charts for the respective baseline benchmarks, mono seems to have generated slower code for the **sumOfSquaresEven** benchmark, in which the modulo operation is applied. This indicates that JIT compilation optimizations can be improved significantly, especially in cases such as the handwritten fused loop-if operation of the **sumOfSquaresEven** situation.

[5] Although it is easy to categorize benchmarks per language, and we refer to languages throughout, it is important to keep in mind that the comparison concerns primarily the standard libraries of these languages and only secondarily the language translation techniques for lambdas.

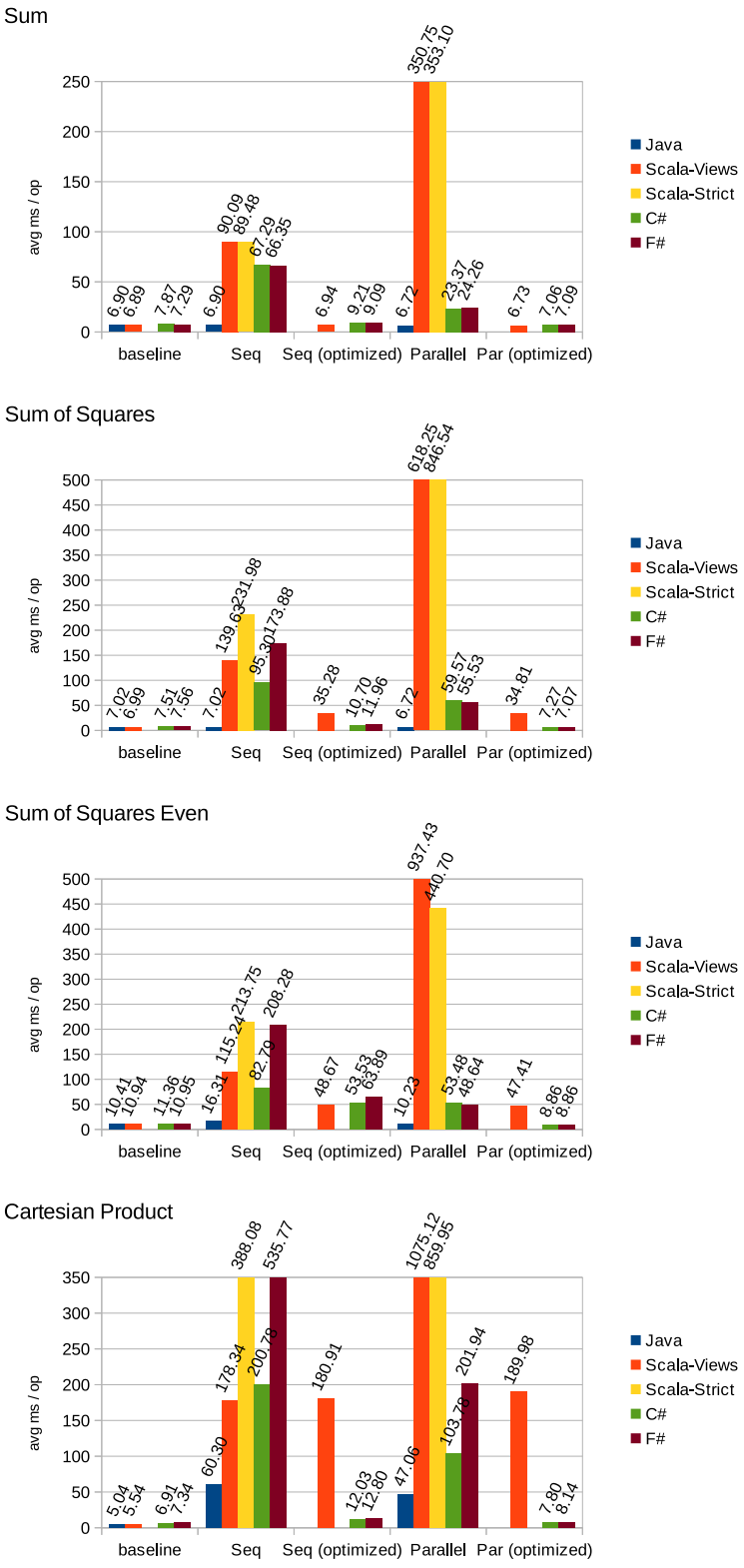


Figure 1: Microbenchmark Results on Windows (CLR/JVM) in milliseconds / iteration (average of 10). Y-axis truncated for readability.

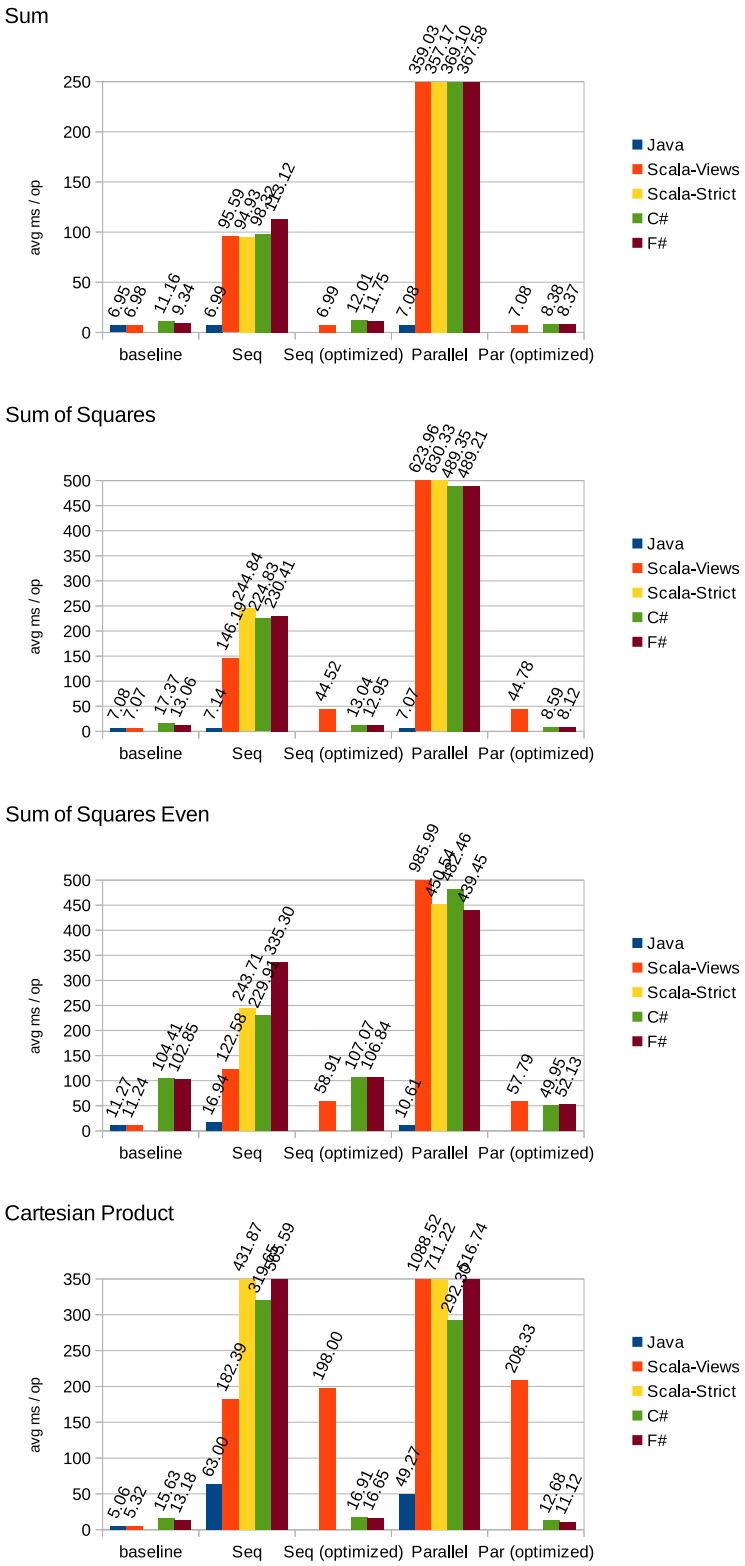


Figure 2: Microbenchmark Results on Linux (mono/JVM) in milliseconds / iteration (average of 10). Y-axis truncated for readability.

	Windows					Linux				
Benchmark	Java	Scala	Scala-Strict	C#	F#	Java	Scala	Scala-Strict	C#	F#
sumBaseline	0.011	0.015		1.214	0.168	0.054	0.011		0.552	0.818
sumSeq	0.015	0.607	0.277	2.407	0.525	0.014	0.449	0.475	0.359	1.015
sumSeqOpt		0.010		0.536	0.212		0.022		0.248	0.730
sumPar	0.035	2.348	2.622	0.895	4.371	0.009	3.653	1.827	106.800	117.358
sumParOpt		0.017		0.075	0.196		0.026		1.400	2.010
sumOfSquaresBaseline	0.008	0.016		0.129	0.202	0.023	0.013		0.799	1.072
sumOfSquaresSeq	0.009	1.049	2.052	0.763	3.755	0.019	1.331	0.895	1.193	1.116
sumOfSquaresSeqOpt		1.104		0.215	0.292		0.238		0.583	0.171
sumOfSquaresPar	0.008	3.691	9.355	2.745	0.162	0.017	2.807	6.347	23.856	40.342
sumOfSquaresParOpt		0.036		0.433	0.094		0.136		0.782	0.485
sumOfSquaresEvenBaseline	0.044	0.085		0.204	0.393	0.059	0.035		0.906	1.270
sumOfSquaresEvenSeq	0.121	1.157	1.510	3.789	4.838	0.096	1.159	1.042	0.895	1.680
sumOfSquaresEvenSeqOpt		0.550		2.052	5.351		0.162		0.847	0.522
sumOfSquaresEvenPar	0.025	5.184	8.207	5.943	2.556	0.027	4.905	16.252	46.739	21.465
sumOfSquaresEvenParOpt		0.502		0.115	0.128		0.483		1.737	4.390
cartBaseline	0.060	0.041		0.015	1.007	0.010	0.010		0.040	0.113
cartSeq	0.749	6.195	3.939	4.284	5.840	0.510	2.437	5.486	0.954	2.791
cartSeqOpt		0.666		0.148	0.232		0.763		0.751	0.307
cartPar	0.131	13.167	13.165	4.954	7.855	0.243	7.641	7.484	10.963	7.546
cartParOpt		2.694		0.904	1.371		2.642		1.810	1.310
refBaseline	0.069	0.259		0.159	0.360	0.152	0.288		1.740	1.566
refSeq	0.221	1.077	0.719	1.267	3.415	0.237	0.438	0.353	1.269	0.639
refSeqOpt		0.284		2.082	1.437		0.235		2.409	1.643
refPar	0.119	5.123	0.853	8.548	2.556	0.271	6.904	0.765	44.879	27.644
refParOpt		0.247		0.782	0.187		0.112		1.445	2.592

Table 1: Standard deviations for 10 runs of each benchmark.

Among all standard parallel libraries, F# achieved the best scaling of 2.6x-4.3x.

Optimizing frameworks: When streaming pipelines are amenable to optimization, the improvement can be dramatic.

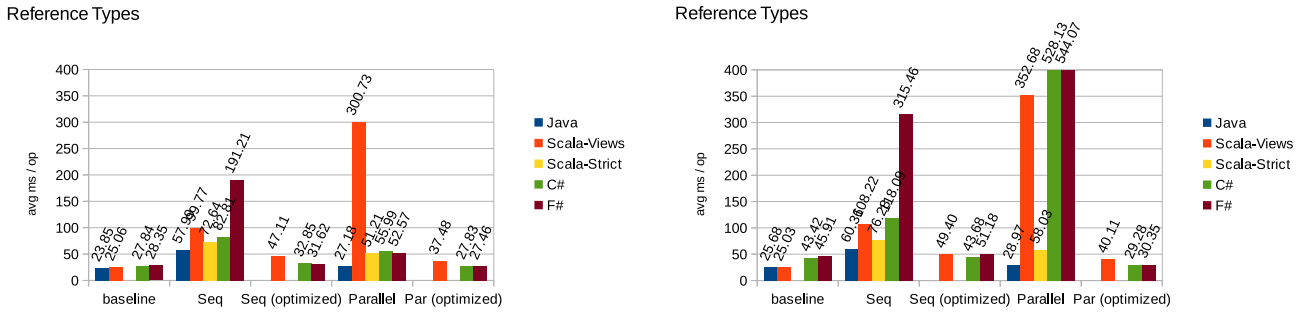
ScalaBlitz improved Scala in virtually all cases. Especially in the **sum** benchmark, Scala was significantly improved, achieving an execution time close to that of the Java/Scala baseline tests. Notable are the 52x speed-up in relation to Scala Parallel Collections for the **sum** benchmark on Windows, as well as 50x on Linux. Additionally, ScalaBlitz achieved a 17x improvement for **sumOfSquares** and 19x for **sumOfSquaresEven** (again for the parallel benchmarks) on Windows. ScalaBlitz did not demonstrate improved performance in the case of nested loops (sequential **cart**) but presented a 5.7x speedup in the parallel version on Windows (and 5.2x on Linux). Apart from the elimination of abstraction penalties, ScalaBlitz offers additional performance improvement in the parallel optimized versions due to its iterators that allow fine-grained and efficient work-stealing [10].

LinqOptimizer improved in all cases the performance of the C# and F# benchmarks. The result of LinqOptimizer universally demonstrates the smallest performance gap with the baseline benchmarks, in absolute values. Especially in the **cart** benchmark, LinqOptimizer achieved a speed-up of 17x(sequential) and 13x(parallel) for C# and 42x and 25x respectively for F#. Among the two .NET languages, F# is the one that benefits more by LinqOptimizer in the sequential **sumOfSquares** and **sumOfSquaresEven** benchmarks. F# gets 14x and 3x improvements for these benchmarks, respectively, while C# gets 9x and 1.5x for the sequential tests. In the case of **cart**, LinqOptimizer has employed the nested loop optimization, which brings execution near the baseline level.

In table 1 we present the standard deviation of all microbenchmarks. Among all measurements, the parallel collections of Scala and C#/ F# on mono/Linux presented the highest deviations. Java demonstrates the highest stability. The strict version of Scala for the parallel **sumOfSquares** benchmark exhibit a relatively higher standard deviation, possibly because of memory effects. (Recall that the strict implementations do not use a fixed heap size.)

4. Discussion

Our microbenchmarks paint a fairly clear picture of the current status of lambdas+streaming implementations, as well as their future improvement prospects. Java employs the most aggressive implementation technique that does not perform invasive optimization. Other languages could benefit from the same translation approach. At the same time, Java does not have an optimization framework along the lines of ScalaBlitz or LinqOptimizer. The **cart** microbenchmark showcases the need for such optimizations: C#/F# are 7x faster in parallel performance than Java. For more realistic programs, such benefits may arise more often. Hence, identifying cases in which Java can benefit from a Stream API optimizing framework (as in the closed-over variables of **cart**) is a promising direction. Scala is an outlier in most of our measurements. We found that its performance, in both the strict and the non-strict case, is subject to memory management effects. We first examined whether such effects can be alleviated with the use of VM flags, without intrusive changes to the benchmarks' source code. Our microbenchmark runs employ the default JVM setup of a parallel garbage collector (GC) with *GC ergonomics* enabled by default. GC ergonomics is an adaptive mechanism that tries to meet (in order) three goals: 1) minimize pause time, 2) maximize throughput, 3) minimize footprint. Leaving GC ergonomics enabled is not always beneficial for Scala. We conducted the same tests without the use of adaptive sizing (-XX:-UseAdaptiveSizePolicy) and no explicit sizing of generations (on Linux). For both strict and non-strict (not optimized) parallel tests, we observed an improvement of 1.1x-2.9x, with the parallel version of **sumOfSquaresPar** exhibiting the maximum increase. However, removing adaptive sizing of the heap also causes a performance degradation of about 10%-15% in the majority of sequential tests. In limited exploration (also based on



(a) Windows tests

(b) Linux tests

Figure 3: Microbenchmark with manual boxing. Y-axis in milliseconds / iteration (average of 10), truncated for readability.

suggestions by Scala experts) we found no other flag setup that significantly affects performance.

The main problem with Scala performance is that the Scala Collections are not specialized for primitive types. Therefore, Scala suffers significant boxing and unboxing overheads for primitive values, as well as memory pressure due to the creation of intermediate (boxed) objects. Prokopec et al. [10] explain such issues, along with the effects of indirections and iterator performance. Method-level specialization for primitive types can currently be effected in two ways. One is the Scala `@specialized` annotation, which specializes chains of annotated generic call sites [2], while the other is *Miniboxing* [15]. Use of the `@specialized` annotation causes the injection of specialized method calls while preserving compatibility with generic code. The use of `@specialized` preserves separate compilation by generating all variants of specialized methods, hence leading to bytecode explosion. Partly due to such considerations, Scala Collections do not employ the `@specialized` annotation. Miniboxing presents a promising alternative that minimizes bytecode size and defers transformations to load time. Currently Miniboxing is offered as a Scala compiler plugin. Having specialized collections in the Scala standard library could greatly improve performance in our benchmarks.

To demonstrate the above points, in Figure 3 we present an additional benchmark (**refs**), which executes a pipeline with reference types and avoids automatic boxing of our input data. The benchmark operates on an array of $10,000,000$ instances of a class, `Ref`, employs two filter combinators, and finally returns the size of the resulting collection. This benchmark effectively performs boxing manually, for all languages. In this benchmark, Java outperforms other streaming libraries but the difference is quite small. Scala is now directly comparable to all other implementations, since it performs no extraneous boxing compared to other languages. Both sequential and parallel tests for Java didn't invoke the GC. However, Scala in the `Filtered` trait, which is defined in the `GenSeqViewLike` implementation trait, causes internal boxing for the size operator. The `length` definition in `Filtered`, which delegates to the lazy value of `index`, and the array allocation inside that lazy value are responsible for this effect. In the Scala-strict parallel test, nearly 100% of the allocated memory (originating both from the main thread and from the Fork/Join workers) comes from the intermediate arrays, but the ample heap space combined with the almost perfect inlining of the main internal transformer (`ParArrayIterator.filter2combiner_quick`) makes the Scala version highly competitive.

Figure 3 exhibits a desirable property: if we consider the implementations that remove the incidental overheads that we identified (and which otherwise dominate computation costs), all language versions exhibit parallel scaling. Observe the parallel speedups in the case of Java, Scala-strict, F#, and C# on Windows.

One final remark is on the choice of using the C2 JIT compiler only (by using the -XX:-TieredCompilation flag). In both Scala and Java tests, using tiered compilation degraded the performance in the majority of our benchmarks. Concretely, for the Scala tests, tiered compilation had only a minor positive effect on the **sum** tests and an approximately 10% performance degradation in all other cases. Regarding the Java cases, all tests, apart from the sequential and parallel versions of the **refs** benchmark, presented performance degradation.

5. Future Work

Several possibilities for further work arise. Our benchmark suite can be enhanced with more complex microbenchmarks to capture the case of streams that include a variable number of successive combinators, such as `filters`. Additionally, an interesting followup would be to examine how measurements are affected as a function of the number of processors. Regarding standard stream APIs, C#, F# and Scala seem to use external iteration while Java uses internal iteration. Thus an interesting direction is to implement internal iterator-based streaming APIs for the aforementioned languages. Finally, LinqOptimizer demonstrated how, by leveraging the LINQ Expression tree API, optimized queries can be obtained, while ScalaBlitz employed macros for compile-time optimizations. Java can benefit from an optimizing framework. As Java can have access to the internal compiler API, a very promising direction to explore is the design and development of an optimizing framework, designed as a `javac` plugin.

6. Conclusions

In this work, we evaluated the combined cost of lambdas and stream APIs in four different multiparadigm languages running on two different runtime platforms. We used benchmarks expressed with the closest comparable datatypes that each language offers in order to preserve semantic equivalence. Our benchmarks constitute a fine grained set. Each benchmark builds upon the previous one in terms of complexity. Additionally we run all benchmarks on both Windows and Linux. Our results clearly show the benefit of advanced implementation techniques in Java, but also the performance advantage of optimizing frameworks that can radically transform streaming pipelines.

Acknowledgments

We would like to thank Aleksey Shipilev, Paul Sandoz, Brian Goetz, Alex Buckley, Doug Lea, and the ScalaBlitz developers, Aleksandar Prokopec and Dmitry Petrashko, for their valuable comments that helped strengthen this study. We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant (PADECL) and a European Research Council Starting/Consolidator grant (SPADE); and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award (MORPH-PL.)

References

[1] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proc. of the 4th Workshop on Scala*, page 3, Montpellier, France, 2013. ACM.

[2] I. Dragos. *Compiling Scala for Performance*. PhD thesis, IC, Lausanne, 2010.

[3] B. Goetz. Translation of Lambda Expressions, Apr. 2012. URL `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`.

[4] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java® Virtual Machine Specification : Java SE 8 Edition, Mar. 2014. URL `http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

[5] E. Meijer. The World According to LINQ. *Queue*, 9(8):60:6060:72, Aug. 2011. ISSN 1542-7730.

[6] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[7] D. G. Murray, M. Isard, and Y. Yu. Steno: Automatic Optimization of Declarative Queries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 121–131, New York, NY, USA, 2011. ACM.

[8] N. Palladinos and K. Rontogiannis. LinqOptimizer: an automatic query optimizer for LINQ to objects and PLINQ., 2013. URL `http://nessos.github.io/LinqOptimizer/`.

[9] A. Prokopec and D. Petrashko. ScalaBlitz: Lightning-Fast Scala collections framework, 2013. URL `http://scala-blitz.github.io/`.

[10] A. Prokopec, D. Petrashko, and M. Odersky. On Lock-Free Work-stealing Iterators for Parallel Data Structures. Technical report, 2014.

[11] J. Rose, D. Coward, O. Bini, W. R. Cook, S. Pedroni, and J. Theodorou. JSR 292: Supporting dynamically typed languages on the java platform, 2011. URL `https://jcp.org/en/jsr/detail?id=292`.

[12] P. Sestoft. Microbenchmarks in Java and C#. 2013.

[13] A. Shipilev, S. Kuksenko, A. Astrand, S. Friberg, and H. Loef. OpenJDK: jmh. URL `http://openjdk.java.net/projects/code-tools/jmh/`.

[14] J. Singer. JVM versus CLR: a comparative study. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 167–169. Computer Science Press, Inc., 2003.

[15] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *Proc. of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 73–92, New York, NY, USA, 2013. ACM.