Streams - CPS

efficient functional style

pipelines on streams of data

{no only in F#}

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism."

— Edward A. Lee

(The Problem with Threads, Berkeley 2006)

AGENDA

- Different Streams Implementation, *optimization* and benchmarking
- Lighting fast Java 8 Streams – How is it possible?
- Continuation Passing Style (CPS)
- Create your own Fast Streams
- Demo and performance measurements

SOMETHING ABOUT ME
RICCARDO TERRELL

- Originally from Italy, currently living in USA ~10 years
  - Living/working in Washington DC
- +/- 18 years in professional programming
  - C++/VB → Java → .Net C# → Haskell → C# & F# → Scala→ ??
- Organizer of the DC F# User Group
- Software Architect @ Microsoft
- Believer in polyglot programming as a mechanism in finding the right tool for the job

@trikace rickyterrell.com

GOALS - PLAN OF THE TALK

Streams are lightweight and powerful

CPS is a great, useful but miss-used tool

- to extend functionality

There is always room to improve performance


DB
218 387-6
104
2
7

Streams

do not provide a means to directly access or manipulate their elements

```
interface Iterable<T> {
   Iterator<T> iterator()
}

interface Iterator<T> {
   boolean hasNext()
   T next()
}
```

Ernst Friedrich Ferdinand Zermelo (1871–1953)

http://en.wikipedia.org/wiki/Ernst_Zermelo#mediaviewer/File:Ernst_Zermelo.jpeg

Typical Pipeline Pattern

***list comprehension* is a syntactic construct for creating a list based on existing lists**

It follows the form of the mathematical "set comprehension" to use HOF as map / filter / reduce …

Typical Pipeline Pattern

```
// source |>
// (('a -> bool) -> 'a list -> 'a list)
// (('a -> 'b) -> 'a list -> 'b list)
// (('a -> 'a -> 'a) -> 'a list -> 'a)
let source' = source
       |> List.filter f |> List.map m |> List.reduce(+)
```

- inter : intermediate lazy operations (map, filter…)
- terminal : produces result or side-effects (reduce…)

JAVA 8 -> STREAM

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

JAVA 8 -> STREAMS -> PARALLEL

```
public double sumOfSquaresSeq(double[] v) {
        double sum = DoubleStream.of(v)
                .parallel()
                  .map(d -> d * d)
                  .sum(); // (x,y) -> x + y
        return sum;
}
```

JAVA 8 -> STREAMS -> COMPILER

```
<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));
               }
            };
        }
```

JAVA 8 -> STREAMS -> COMPILER

```
<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));
               }
            };
        }
```

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

C# -> LINQ

```
public int sumOfSquaresEvenSeq(int[] v) {
        int sum =
                v.Where(x => x % 2 == 0)
                 .Select(x => x * x)
                 .Sum();
        return sum;
}
```

C# -> LINQ

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


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

C# -> LINQ -> PARALLEL

```
public int sumOfSquaresEvenSeq(int[] v) {

        int sum =
                v.AsParallel()
                 .Where(x => x % 2 == 0)
                 .Select(x => x * x)
                 .Sum();
        return sum;
}
```

F# -> Seq

```
let data = [|1..1000000|] |> Array.map int64

data
|> Seq.filter(fun i -> i % 2L = 0L) // lazy
|> Seq.map(fun i -> i + 1L) // lazy
|> Seq.sum // eager force evaluation
```

```
|>  ('a -> ('a -> 'b) -> 'b)
```

F# -> Seq -> Parallel

```
let data = [|1..1000000|] |> Array.map int64

data
|> PSeq.filter(fun i -> i % 2L = 0L) // lazy
|> PSeq.map(fun i -> i + 1L) // lazy
|> PSeq.sum // eager force evaluation
```

C# & F# -> LINQ & SEQ -> COMPILER

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

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

C# & F# -> LINQ & Seq -> Compiler

```
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.func = func;

        this.inner = inner;

}

MoveNext() { return inner.MoveNext(); }

Current { get { return func(inner.Current); } }

}
```

C# & F# -> LINQ & Seq -> Compiler

```
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.func = func;

        this.inner = inner;

}

MoveNext() { return inner.MoveNext(); }

Current { get { return func(inner.Current); } }

}
```

SCALA -> SEQVIEW

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

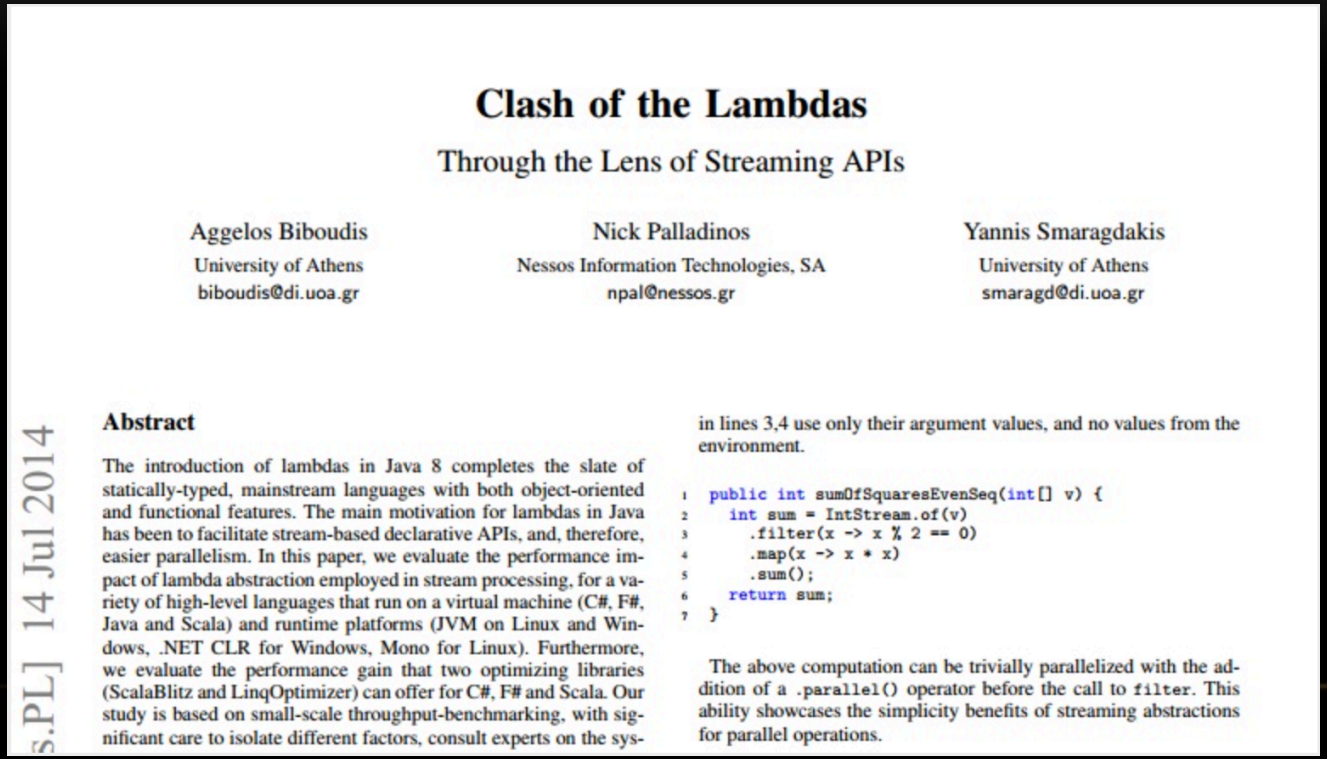
SCALA -> SEQVIEW -> COMPILER

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

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

SCALA -> SEQVIEW -> COMPILER

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

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

Clash of the Lambdas



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 sys-

in lines 3,4 use only their argument values, and no values from the environment.

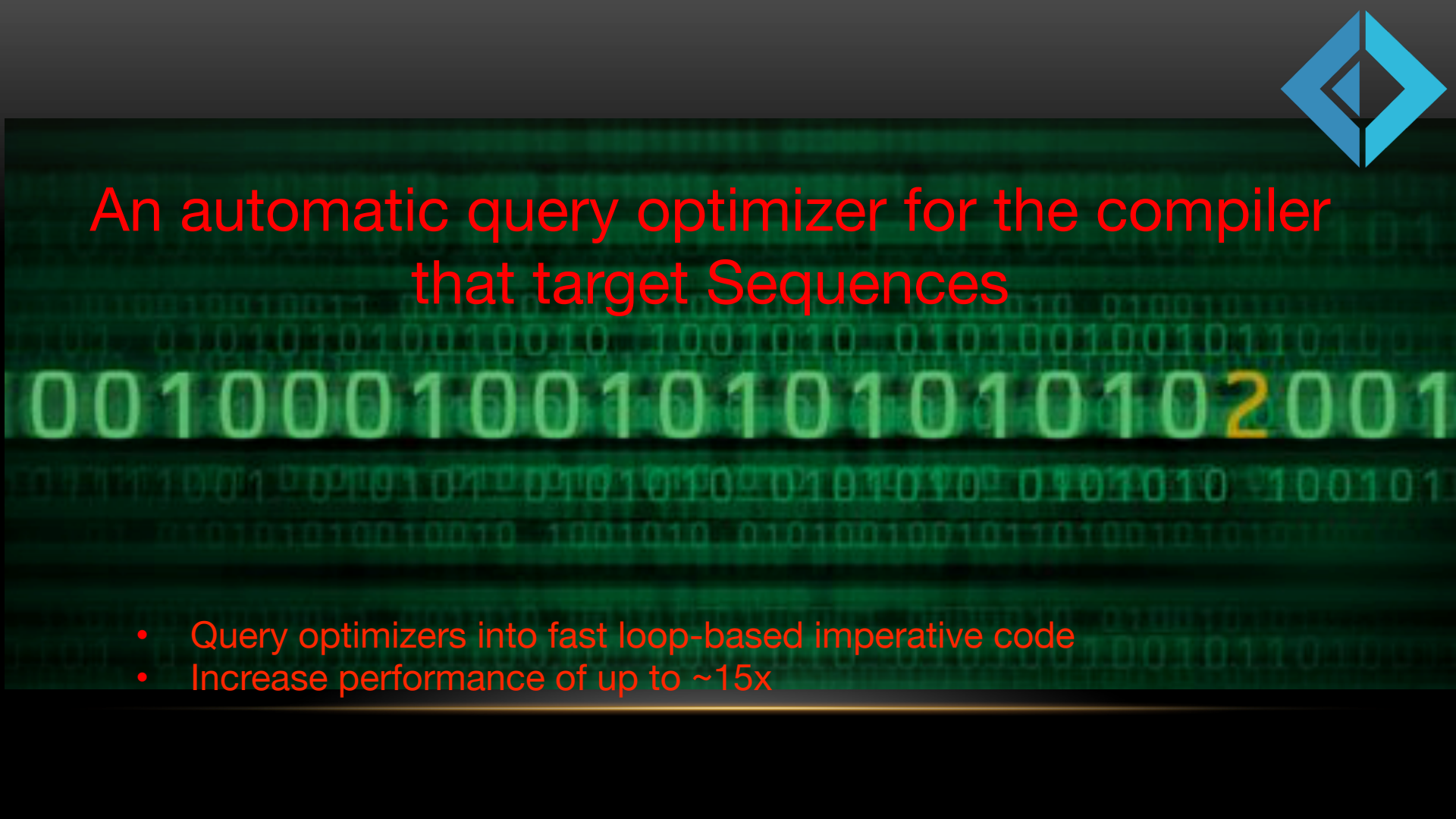
```
1 public int sumOfSquaresEvenSeq(int[] v) {
2   int sum = IntStream.of(v)
3     .filter(x -> x % 2 == 0)
4     .map(x -> x * x)
5     .sum();
6   return sum;
7 }
```

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.

http://arxiv.org/abs/1406.6631


Make functional data query pipelines FAST

- LinqOptmizer
- Steno
- ScalaBlitz
- Stream Fusion



An automatic query optimizer for the compiler that target Sequences

- Query optimizers into fast loop-based imperative code
- Increase performance of up to ~15x

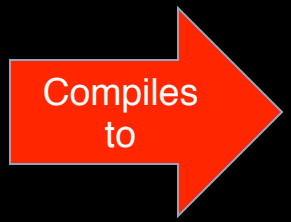
Linq-Optimizer

```
var query = (from num in nums.AsQueryable ()
             where num % 2 == 0
             select num * num).Sum ();
```

https://github.com/nessos/LinqOptimizer

Linq-Optimizer

```
var query = (from num in nums.AsQueryable ()
             where num % 2 == 0
             select num * num).Sum ();
```




```
int sum = 0;
for (int index = 0; index < nums.Count; index++) {
     int num = nums [index];
     if (num % 2 == 0)
         sum += num * num;
            }
```

Loop Fusion

```
int i, a[100], b[100];
 for (i = 0; i < 100; i++)
   a[i] = 1;
 for (i = 0; i < 100; i++)
   b[i] = 2;
```

Is equivalent to

```
int i, a[100], b[100];
  for (i = 0; i < 100; i++)
  {
    a[i] = 1;
    b[i] = 2;
  }
```

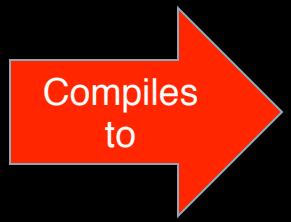
Linq-Optimizer

```
var query = (from num in nums.AsParallelQueryExpr()
             where num % 2 == 0
             select num * num).Sum ();
```

https://github.com/nessos/LinqOptimizer

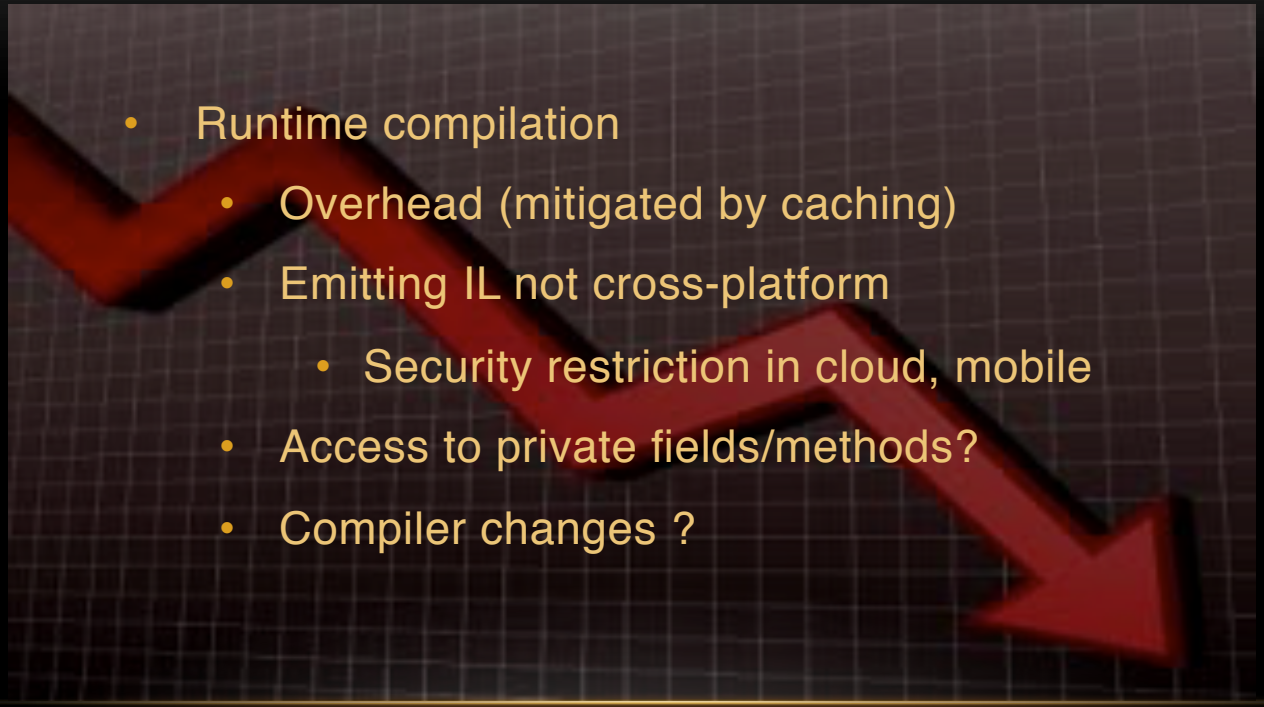
Linq-Optimizer

```
var query = (from num in nums. nums.AsParallelQueryExpr()
             where num % 2 == 0
             select num * num).Sum ();
```




```
ParallelQuery.ReduceCombine (nums, 0,
          (acc, num) => {
            if (num % 2 == 0)
                return acc + num * num;
            else
                return acc;
             }, (left, right) => left + right);
```

Linq-Optimizer disadvantages



- Runtime compilation
  - Overhead (mitigated by caching)
  - Emitting IL not cross-platform
    - Security restriction in cloud, mobile
  - Access to private fields/methods?
  - Compiler changes ?

STENO Automatic Optimization of Declarative Queries

Steno: Automatic Optimization of Declarative Queries

Derek G. Murray
University of Cambridge Computer Laboratory
Derek.Murray@cl.cam.ac.uk

Michael Isard Yuan Yu
Microsoft Research Silicon Valley
{misard, yuanbyu}@microsoft.com

Abstract

Declarative queries enable programmers to write data manipulation code without being aware of the underlying data structure implementation. By increasing the level of abstraction over imperative code, they improve program readability and, crucially, create opportunities for automatic parallelization and optimization. For example, the Language Integrated Query (LINQ) extensions to C# allow the same declarative query to process in-memory collections, and datasets that are distributed across a compute cluster. However, our experiments show that the *serial* performance of declarative code is several times slower than the equivalent hand-optimized code, because it is implemented using run-time abstractions—such as iterators—that incur overhead due to virtual function calls and

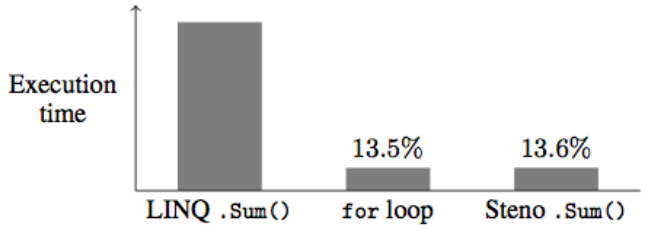


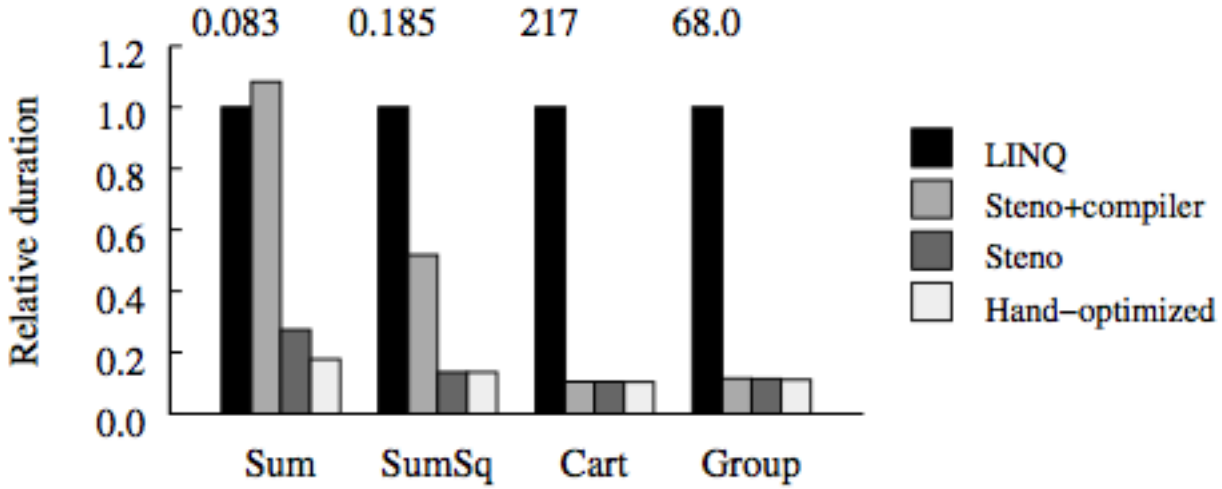
Figure 1. Relative execution time for computing the sum of squares of 10^7 `doubles` using LINQ, an imperative loop, and a Steno-optimized query. Steno achieves a 7.4× speedup over LINQ.

http://research.microsoft.com/pubs/173946/paper-pldi.pdf

C# & F# LINQ -> STENO

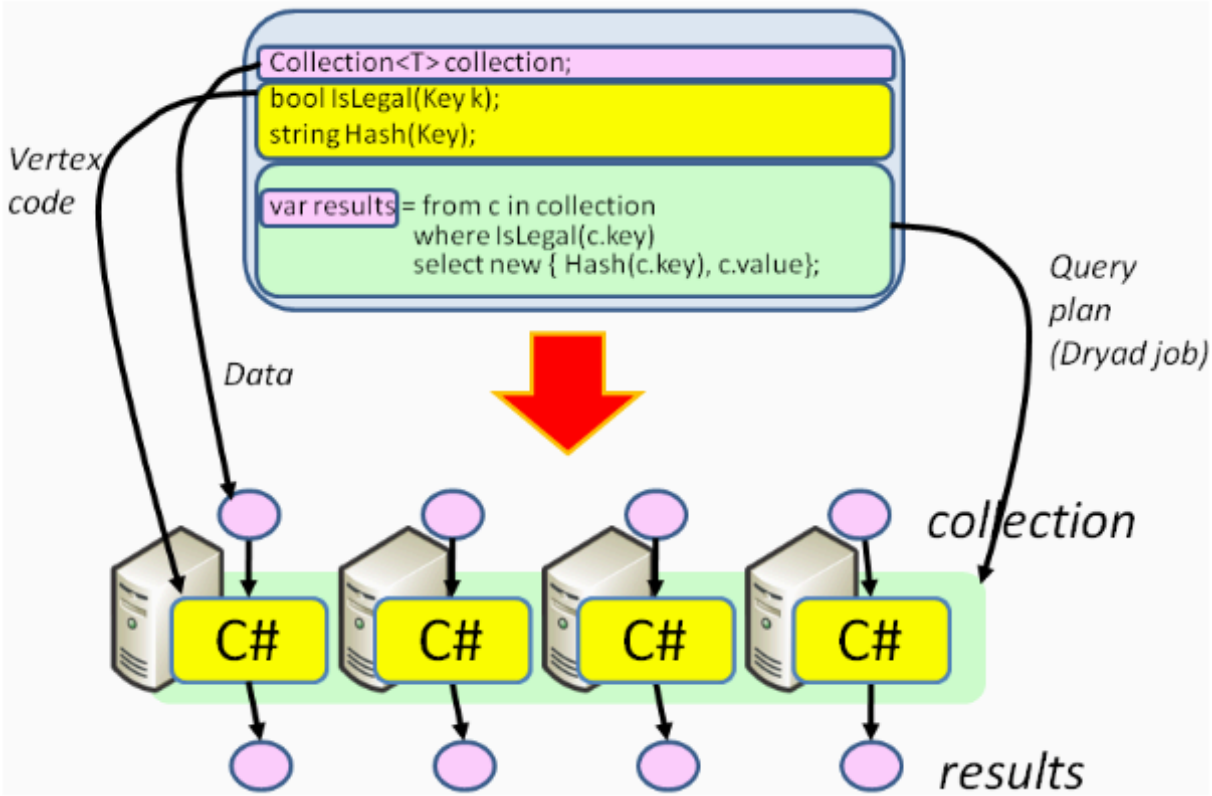
- LINQ are lazy evaluated by using iterator, which is the cause of two virtual function calls per element per query operator
- Nested LINQ query increase overhead previous point
- Steno converts queries into efficient, loop-based imperative code

C# & F# LINQ -> STENO



Relative performance of LINQ, Steno-optimized and hand-optimized queries for sequential micro benchmarks. Lower values are better. Each query is annotated with the absolute LINQ execution time, in seconds.

C# & F# LINQ -> STENO -> DRYADLINQ



HASKELL -> STREAM FUSION

Stream Fusion

From Lists to Streams to Nothing at All

Duncan Coutts[1] Roman Leshchinskiy[2] Don Stewart[2]

[1] Programming Tools Group
Oxford University Computing Laboratory
duncan.coutts@comlab.ox.ac.uk

[2] Computer Science & Engineering
University of New South Wales
{rl,dons}@cse.unsw.edu.au

Abstract

This paper presents an automatic deforestation system, *stream fusion*, based on equational transformations, that fuses a wider range of functions than existing short-cut fusion systems. In particular, stream fusion is able to fuse zips, left folds and functions over nested lists, including list comprehensions. A distinguishing feature of the framework is its simplicity: by transforming list functions to expose their structure, intermediate values are eliminated by general purpose compiler optimisations.

No previously implemented short-cut fusion system eliminates all the lists in this example. The fusion system presented in this paper does. With this system, the Glasgow Haskell Compiler (The GHC Team 2007) applies all the fusion transformations and is able to generate an efficient "worker" function f' that uses only unboxed integers ($Int\#$) and runs in constant space:

$$f' :: Int\# \to Int\#$$
$$f'\ n =$$
$$\quad \textbf{let}\ go :: Int\# \to Int\# \to Int\#$$

http://metagraph.org/papers/stream_fusion.pdf

HASKELL -> STREAM FUSION

Stream Fusion

From Lists to Streams to Nothing at All

Duncan Coutts[1] Roman Leshchinskiy[2] Don Stewart[2]

[1] Programming Tools Group
Oxford University Computing Laboratory
duncan.coutts@comlab.ox.ac.uk

[2] Computer Science & Engineering
University of New South Wales
{rl,dons}@cse.unsw.edu.au

The key trick is that all producers are non-recursive

Abstract

This paper presents an automatic deforestation system, *stream fusion*, based on equational transformations, that fuses a wider range of functions than existing short-cut fusion systems. In particular, stream fusion is able to fuse zips, left folds and functions over nested lists, including list comprehensions. A distinguishing feature of the framework is its simplicity: by transforming list functions to expose their structure, intermediate values are eliminated by general purpose compiler optimisations.

No previously implemented short-cut fusion system eliminates all the lists in this example. The fusion system presented in this paper does. With this system, the Glasgow Haskell Compiler (The GHC Team 2007) applies all the fusion transformations and is able to generate an efficient "worker" function f' that uses only unboxed integers ($Int\#$) and runs in constant space:

$$
\begin{aligned}
&f' :: Int\# \to Int\# \\
&f'\ n\ = \\
&\quad \text{let } go :: Int\# \to Int\# \to Int\#
\end{aligned}
$$

http://metagraph.org/papers/stream_fusion.pdf

HASKELL -> STREAM FUSION

```
data Stream a = E s . Stream (s → Step a s) s

maps :: (a → b) → Stream a → Stream b
maps f (Stream next0 s0) = Stream next s0
    where
        next s = case next0 s of
            Done → Done
            Skip s' → Skip s'
            Yield x s' → Yield (f x) s'
```

http://metagraph.org/papers/stream_fusion.pdf

HASKELL -> STREAM FUSION

Stream fusion three step approach:

1. Convert recursive structures into non-recursive *co-structures*

2. Eliminate superfluous conversions between structures and co-Structures

1. Use general optimizations to fuse the co-structure code.

HASKELL -> STREAM FUSION

Haskell Beats C Using Generalized Stream Fusion

Geoffrey Mainland
Microsoft Research Ltd
Cambridge, England
gmainlan@microsoft.com

Roman Leshchinskiy
rl@cse.unsw.edu.au

Simon Peyton Jones
Microsoft Research Ltd
Cambridge, England
simonpj@microsoft.com

Abstract

Stream fusion [6] is a powerful technique for automatically transforming high-level sequence-processing functions into efficient implementations. It has been used to great effect in Haskell libraries for manipulating byte arrays, Unicode text, and unboxed vectors. However, some operations, like vector append, still do not perform well within the standard stream fusion framework. Others, like SIMD computation using the SSE and AVX instructions available on modern x86 chips, do not seem to fit in the framework at all.

which fusion is relatively straightforward, was the first truly general solution.

Instead of working directly with lists or vectors, stream fusion works by re-expressing these functions as operations over streams, each represented as a state and a step function that transforms the state while potentially yielding a single value. Alas, different operations need different stream representations, and no single representation works well for all operations (§2.3). Furthermore, for many operations it is not obvious what the choice of representation should be.

http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/haskell-beats-C.pdf

HASKELL -> STREAM FUSION

Haskell Beats C Using Generalized Stream Fusion

Geoffrey Mainland
Microsoft Research Ltd
Cambridge, England
gmainlan@microsoft.com

Roman Leshchinskiy
rl@cse.unsw.edu.au

Simon Peyton Jones
Microsoft Research Ltd
Cambridge, England
simonpj@microsoft.com

Haskell – Stream Fusion faster than C

Abstract

Stream fusion [6] is a powerful technique for automatically transforming high-level sequence-processing functions into efficient implementations. It has been used to great effect in Haskell libraries for manipulating byte arrays, Unicode text, and unboxed vectors. However, some operations, like vector append, still do not perform well within the standard stream fusion framework. Others, like SIMD computation using the SSE and AVX instructions available on modern x86 chips, do not seem to fit in the framework at all.

which fusion is relatively straightforward, was the first truly general solution.

Instead of working directly with lists or vectors, stream fusion works by re-expressing these functions as operations over streams, each represented as a state and a step function that transforms the state while potentially yielding a single value. Alas, different operations need different stream representations, and no single representation works well for all operations (§2.3). Furthermore, for many operations it is not obvious what the choice of representation should be.

http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/haskell-beats-C.pdf

ScalaBlitz

Lightning-fast Scala collections framework

SCALABLITZ

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

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

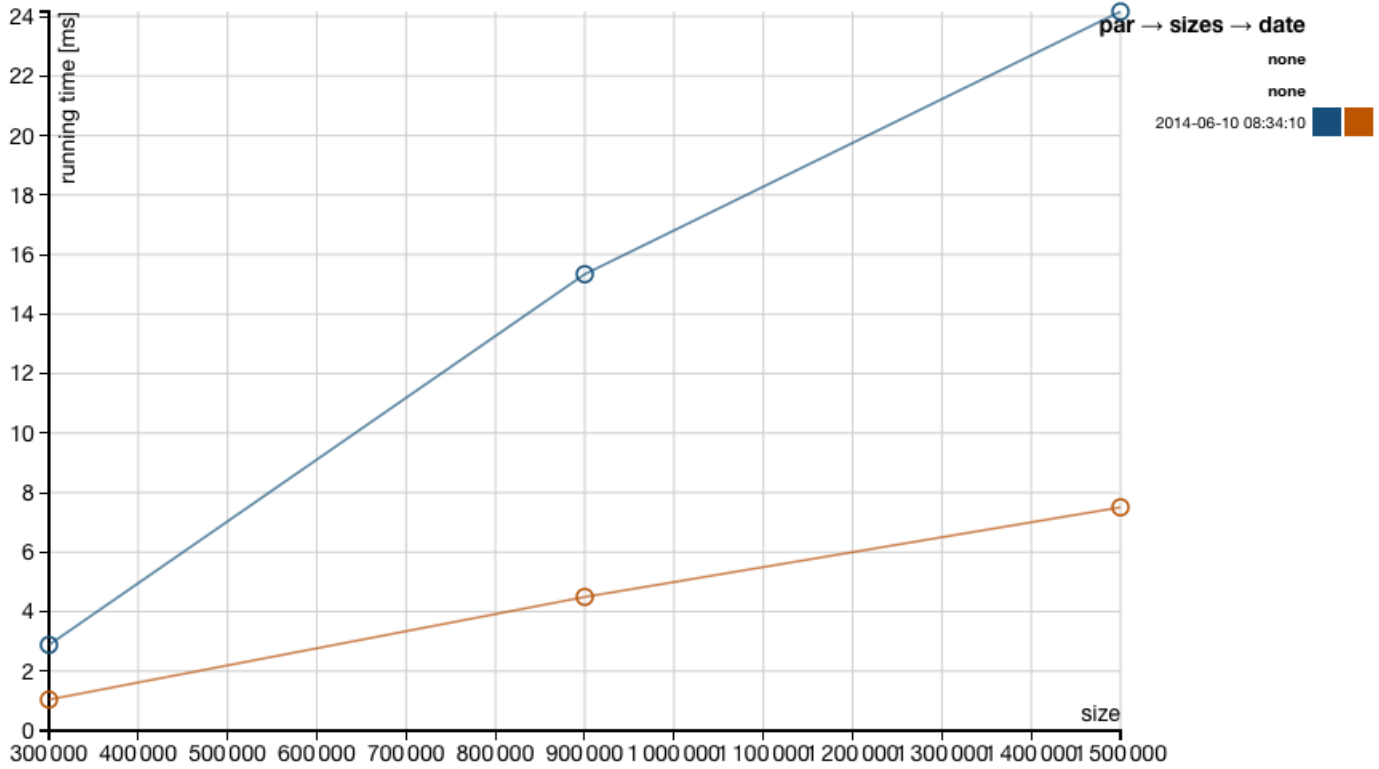
SCALABLITZ

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

- Eliminates boxing
- Performs Lambda inlining
- Loop fusion

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

SCALABLITZ - REDUCE



Micro-benchmark results
Java *vs* Scala *vs* C# & F#
Sequential & Parallel

- Sum iteration speed with no lambdas, just a single iteration.

- Sum Of Squares a small pipeline with one map operation

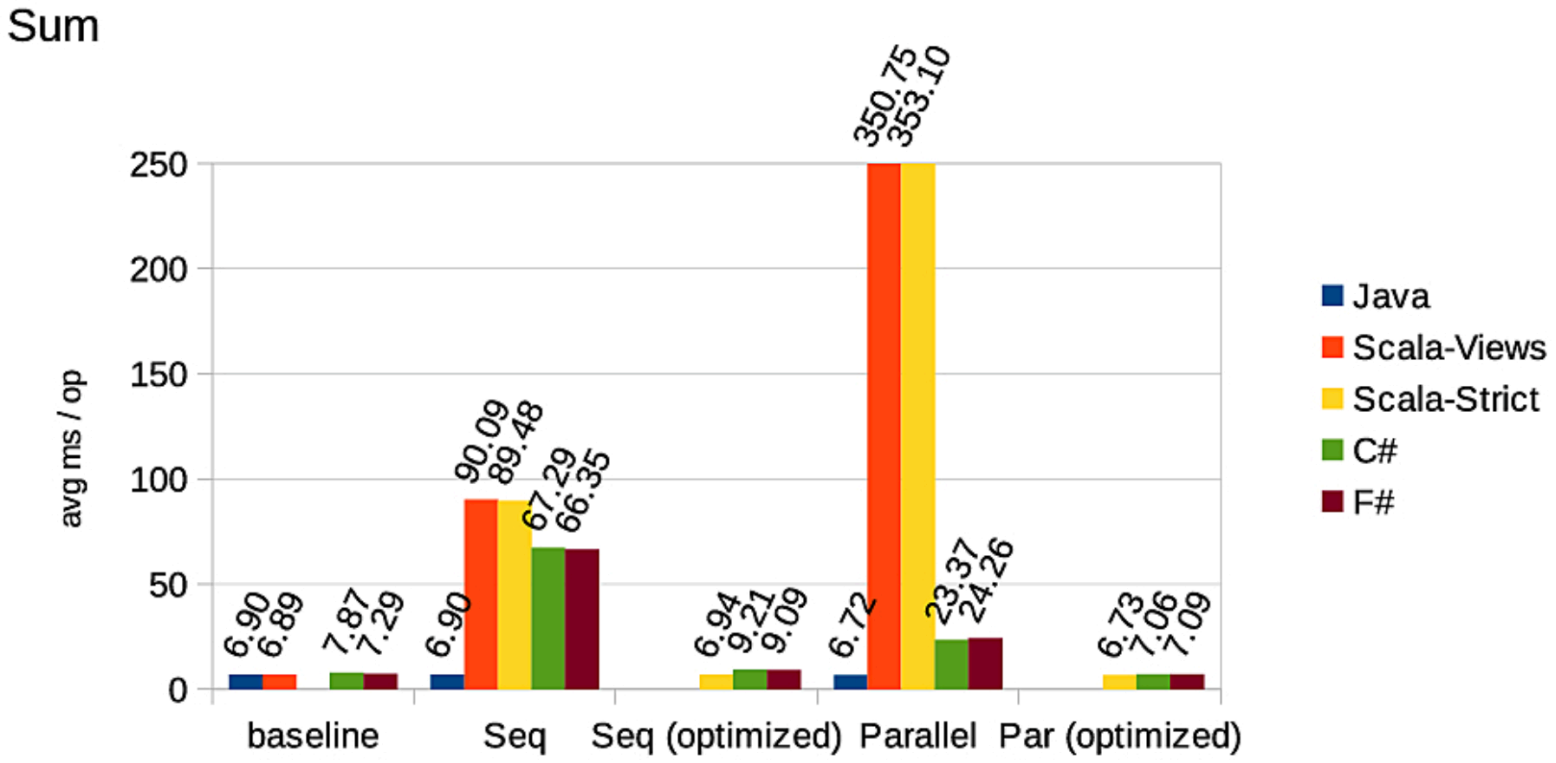
- Cartesian product a nested pipeline with a flatMap and an inner operation, again with a flatMap (capturing a variable), to encode a Cartesian product.

Settings & Setup

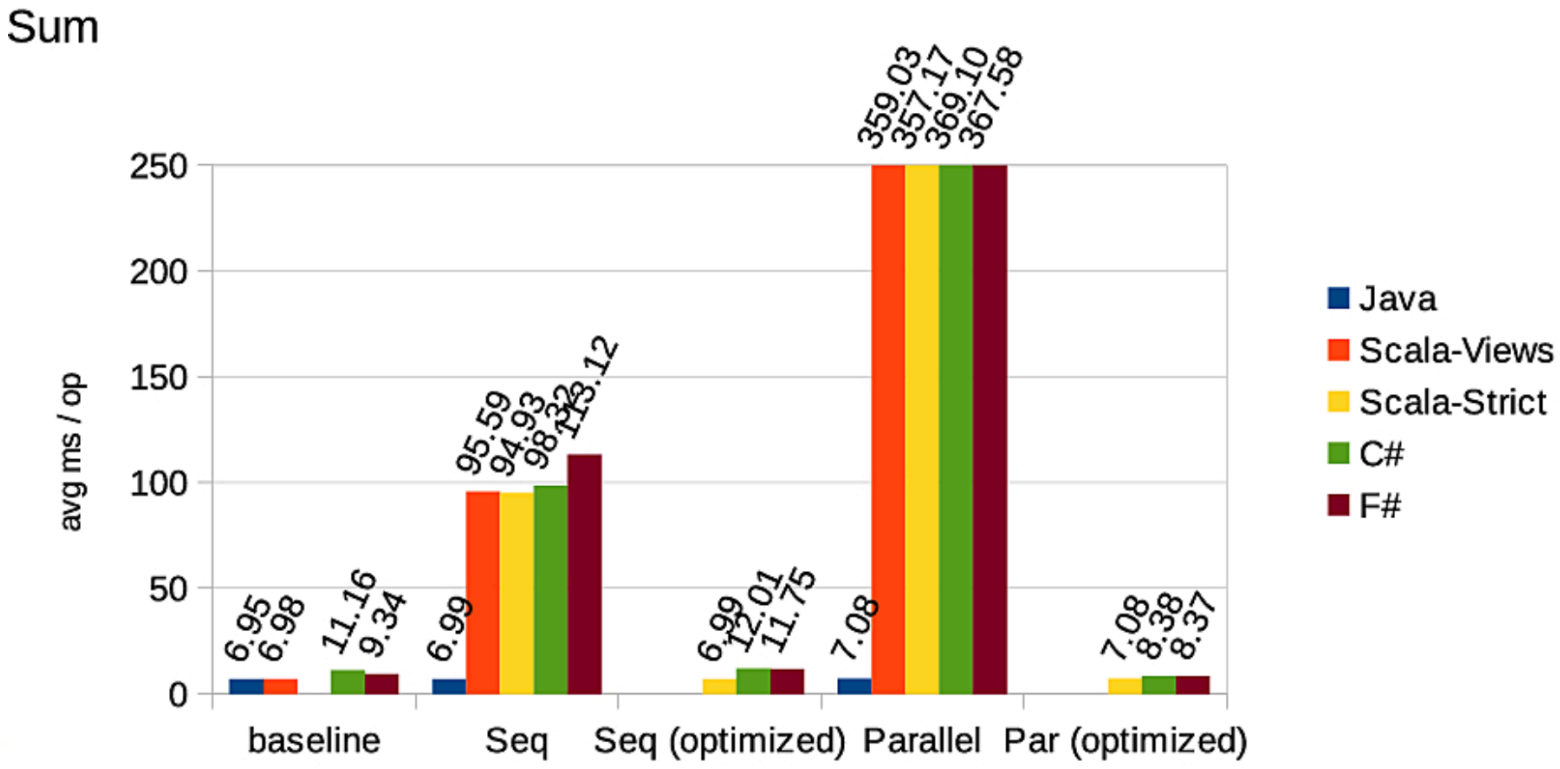
	Windows	Ubuntu Linux
Version	8.1	13.10/3.11.0-20
Architecture	x64	x64
CPU	Intel Core i5-3360M vPro 2.8GHz	
Cores	2 physical x 2 logical	
Memory	4GB	

	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

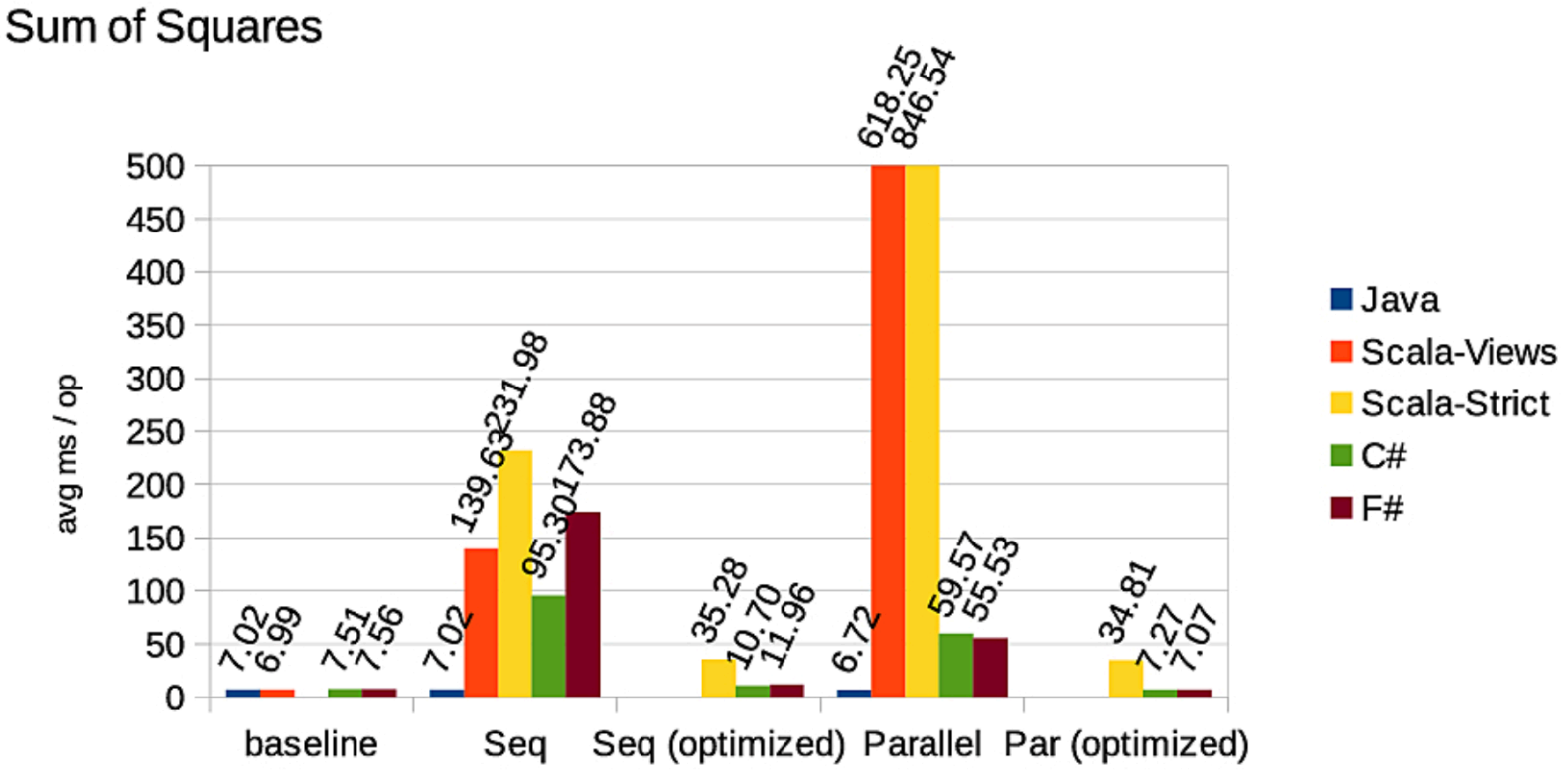
Windows


Sum
avg ms / op
250
200
150
100
50
0
6.90
6.89
7.87
7.29
6.90
90.09
89.48
67.29
66.35
6.94
9.21
9.09
6.72
350.75
353.10
23.37
24.26
6.73
7.06
7.09
baseline
Seq
Seq (optimized)
Parallel
Par (optimized)
Java
Scala-Views
Scala-Strict
C#
F#

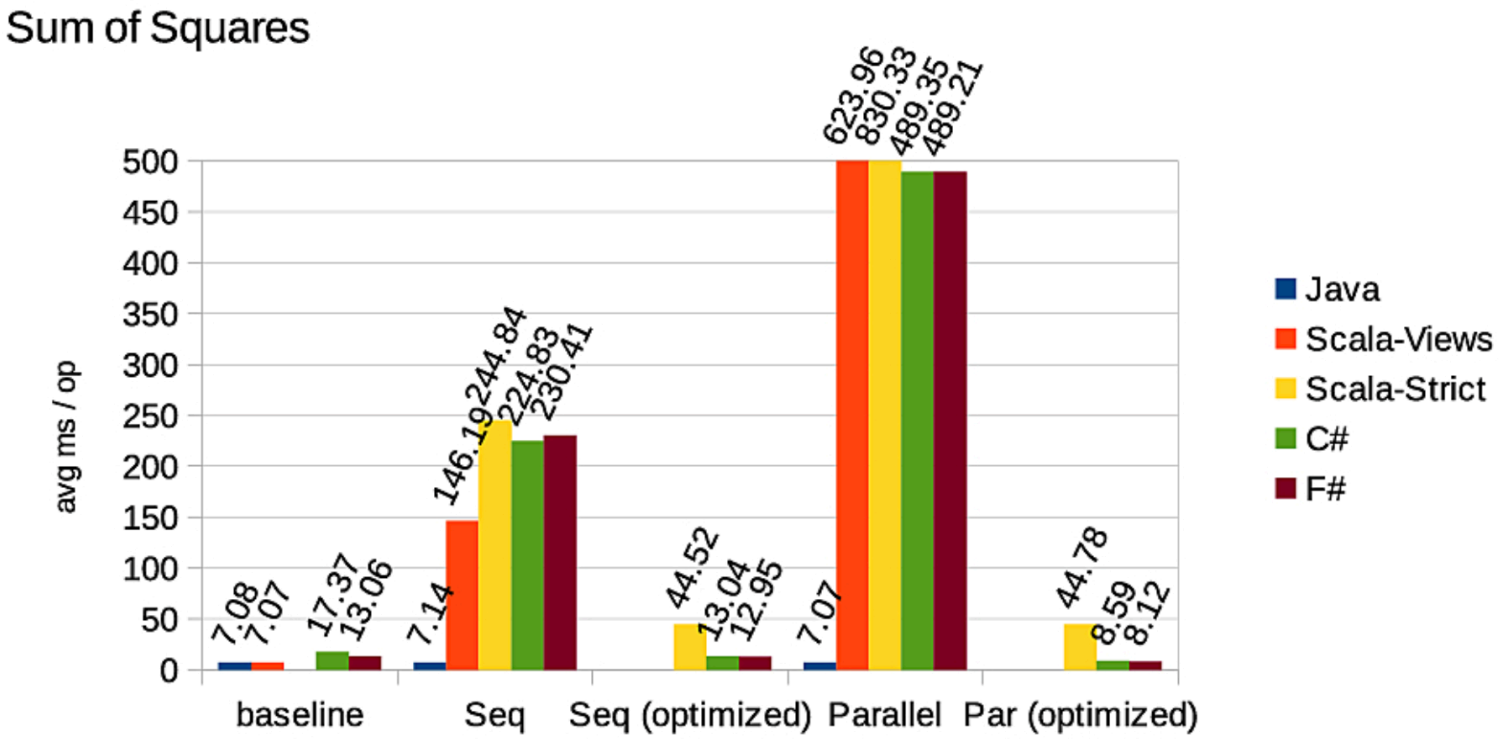
Linux


Sum
avg ms / op
250
200
150
100
50
0
6.95
6.98
11.16
9.34
6.99
95.59
94.93
98.32
113.12
6.99
12.01
11.75
7.08
359.03
357.17
369.10
367.58
7.08
8.38
8.37
baseline
Seq
Seq (optimized)
Parallel
Par (optimized)
Java
Scala-Views
Scala-Strict
C#
F#

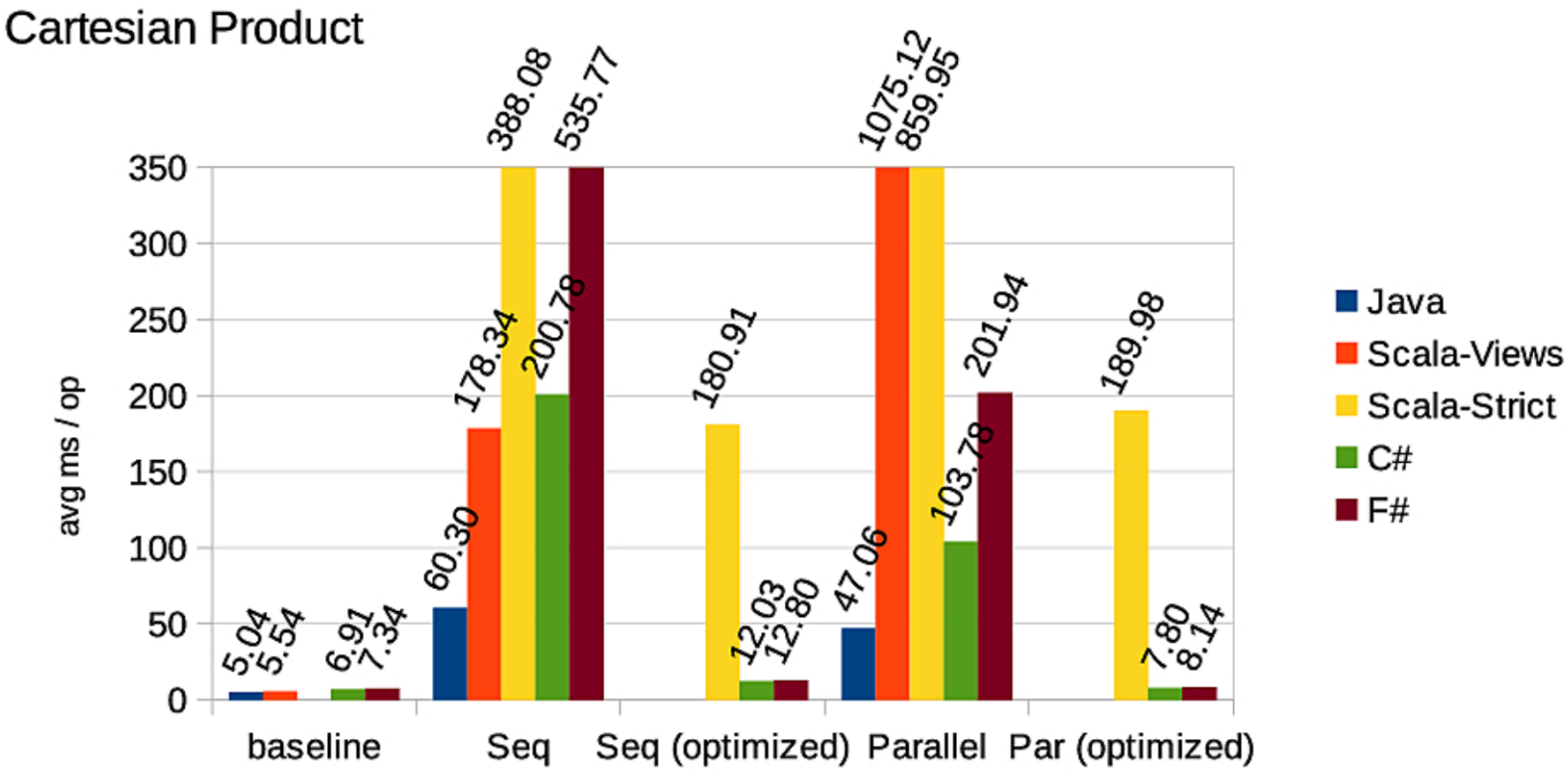
Windows


Sum of Squares
avg ms / op
500
450
400
350
300
250
200
150
100
50
0
7.02
6.99
7.51
7.56
7.02
139.63
231.98
95.30
173.88
35.28
10.70
11.96
6.72
618.25
846.54
59.57
55.53
34.81
7.27
7.07
baseline
Seq
Seq (optimized)
Parallel
Par (optimized)
Java
Scala-Views
Scala-Strict
C#
F#

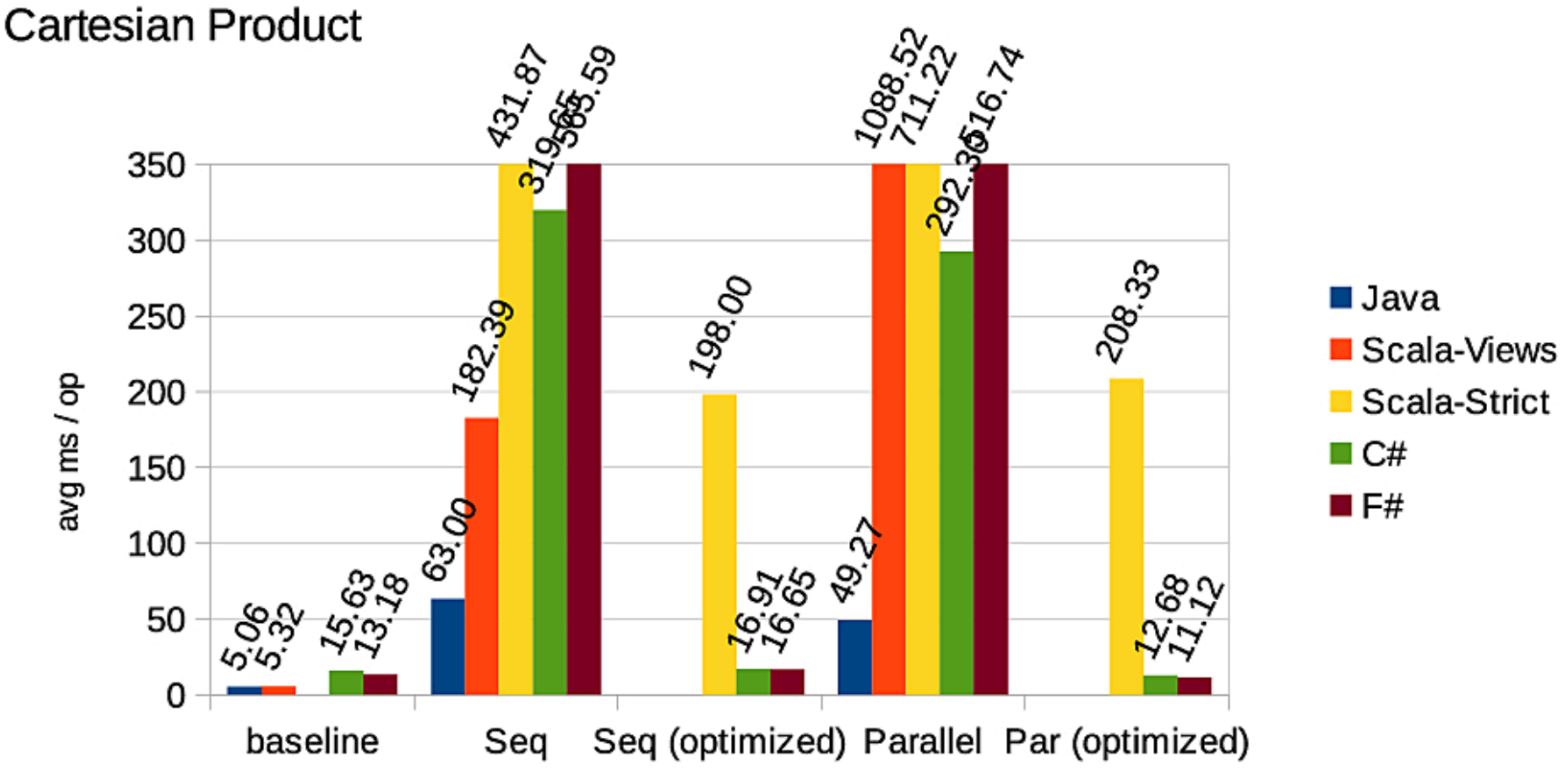
Linux


Sum of Squares
avg ms / op
500
450
400
350
300
250
200
150
100
50
0
7.08
7.07
17.37
13.06
7.14
146.19
244.84
224.83
230.41
44.52
13.04
12.95
7.07
623.96
830.33
489.35
489.21
44.78
8.59
8.12
baseline
Seq
Seq (optimized)
Parallel
Par (optimized)
Java
Scala-Views
Scala-Strict
C#
F#

Windows


Cartesian Product
avg ms / op
350
300
250
200
150
100
50
0
5.04
5.54
6.91
7.34
60.30
178.34
388.08
200.78
535.77
180.91
12.03
12.80
47.06
1075.12
859.95
103.78
201.94
189.98
7.80
8.14
baseline
Seq
Seq (optimized)
Parallel
Par (optimized)
Java
Scala-Views
Scala-Strict
C#
F#

Linux


Cartesian Product
avg ms / op
350
300
250
200
150
100
50
0
5.06
5.32
15.63
13.18
63.00
182.39
431.87
319.65
365.59
198.00
16.91
16.65
49.27
1088.52
711.22
292.30
416.74
208.33
12.68
11.12
baseline
Seq
Seq (optimized)
Parallel
Par (optimized)
Java
Scala-Views
Scala-Strict
C#
F#


JAVA 8 IS VERY FAST


WHAT MAKES JAVA 8 FASTER
?

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349

December 1975

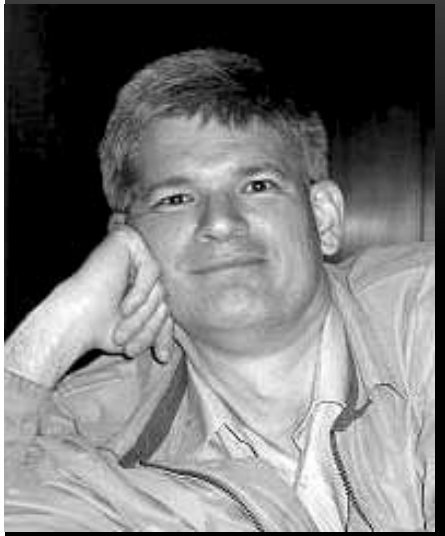
SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

by

Gerald Jay Sussman and Guy Lewis Steele Jr.

Abstract:
Inspired by ACTORS [Greif and Hewitt] [Smith and Hewitt], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [Church], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:



Another Way To Do Recursion

Now consider the following alternative definition of FACT. It has an extra argument, the continuation [Reynolds], which is a function to call with the answer, when we have it, rather than return a value; that is, rather than ultimately reducing to the desired value, it reduces to a combination which is the application of the continuation to the desired value.

http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf

CONTINUATION PASSING STYLE (CPS)

A function written in continuation-passing style takes an extra argument: an explicit "continuation" i.e. a function of one argument. When the CPS function has computed its result value, it "returns" it by calling the continuation function with this value as the argument. That means that when invoking a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's "return" value. Expressing code in this form makes a number of things explicit which are implicit in direct style. These include: procedure returns, which become apparent as calls to a continuation; intermediate values, which are all given names; order of argument evaluation, which is made explicit; and tail calls, which simply call a procedure with the same continuation, unmodified, that was passed to the caller.

Programs can be automatically transformed from direct style to CPS.

```
(A -> B) -> B

(A, B -> unit) -> unit
```

A little more about CPS

- **Converting to CPS**

```
static int Max(int n, int m){
       return n > m ? n : m;
}
```

A little more about CPS

- **Converting to CPS**
  - Change the return type to void
  - Add an extra argument of type Action<T> where T is the original return type
  - Replace all return statements with invocations of the new continuation argument passing the expression used in the return statement

```
static int Max(int n, int m){
       return n > m ? n : m;
}
```

```
static void Max(int n, int m, Action<int> k){
       var max = n > m ? n : m;
        k (max);
}
```

Recursive CPS

```
static void Factorial(int n, Action<int> k){
       if (n == 0)
              k (1);
       else
              Factorial (n - 1, x => k (n * x));
}

Factorial(5, x => Console.WriteLine(x));
```

A little more about CPS

- **Returning Values with Continuations**

```
// ((T -> T) -> T) -> T
T CallCC<T>(Func<Func<T, T>, T> f)

static int Foo(int n){
    return CallCC<int>(Return => {
        Return(n);
        return n + 1;
    });
}

Foo(42);
```

Why CPS?

What exactly can a programmer do with CPS besides showoff at parties?

- Compilers use a more thorough CPS transformation to produce an intermediate form amenable to many analysis.
- UI frameworks use CPS to keep the UI responsive while allowing nonlinear program interaction.
- Web servers use CPS to allow computation to flow asynchronously across pages (Callbacks)

Java 8 Streams use CPS -> are pushing

```
Stream.OfArray(data)
      .filter(i -> i % 2 == 0)
      .map(i -> i * i)
      .sum();
```

The source is pushing data down the pipeline

C# & F# -> LINQ & Seq are pulling

```fsharp
let data = [|1..1000000|] |> Array.map int64

data
|> Seq.find(fun i -> i % 2L = 0L) // lazy
|> Seq.map(fun i -> i + 1L) // lazy
|> Seq.sum // eager force evaluation
```

The terminal function Sum-Aggregate is pulling data from the pipeline via IEnumerator.Current and IEnumerator.MoveNext()

How does it work applied to Sequences?

How does it work applied to Sequences?

Can we use the same approach?

LET'S MAKE A SIMPLE STREAMS

Starting from Seq.iter

```
// iter : ('T -> unit) -> seq<'T> -> unit
let iter f values =
        for value in values do
            f value
```

Starting from Seq.iter

```
// iter : ('T -> unit) -> seq<'T> -> unit
let iter f values =
        for value in values do
            f value
```

FLIP THE ARGS

```
seq<'T> -> ('T -> unit) -> unit
```

Streams !

```
type Stream<'T> = ('T -> unit) -> unit

// iter : seq<'T> -> Stream<'T>
let iter values f =
        for value in values do
            f value
```

Simple Streams

```
type Stream = ('T -> unit) -> unit

// map : ('T -> 'R) -> Stream<'T> -> Stream<'R>
let map f stream =
    fun k -> stream (fun v -> k (f v))

// filter : ('T -> bool) -> Stream<'T> -> Stream<'T>
let filter f stream =
    fun k -> stream (fun v -> if f v then k v else ())

// length : Stream<'T> -> int
let length stream =
    let counter = ref 0
    stream (fun _ -> counter := counter.Value + 1)
    counter.Value
```

Streams are pushing

```
let data = [|1..1000000|] |> Array.map int64

let stream =
        Stream.ofArray data // source
        |> Stream.filter(fun i -> i % 2L = 0)
        |> Stream.map(fun i -> i + 1L)
        |> Stream.sum
```

The source is pushing data down the pipeline

When to stop pushing ?

```
type Stream<'T> = ('T -> unit) -> unit
```

Stopping push required ->

```
Stream.takeWhile : ('T -> bool) ->
                   Stream<‘T> -> Stream<‘T>
```

Stopping push

Change

```
type Stream<'T> = ('T -> unit) -> unit
```

to

```
type Stream<'T> = ('T -> bool) -> unit
```

```
// takeWhile : ('T -> bool) -> Stream<'T> -> Stream<'T>
let takeWhile f stream =
    fun k -> stream (fun v -> if f v then k v else false)
```

Streams can push and pull

```
// Provides functions for iteration
type Iterable<'T> = {
     Bulk : unit -> T
     TryAdvance : unit -> bool }


// Represents a Stream of Values
type Strean<'T> = Stream of (('T -> bool)
                            -> Iterable<'T>)
```

ofArray

```
val ofArray : 'T[] -> Stream<'T>
let ofArray values =
    fun k ->
        let bulk () =  for value in values do
                           k value
        let index = ref -1
        let tryAdvance () =
            incr index;
            if !index < Array.length values then
                (k values.[!index])
                true
            else
                false
        { Builk = bulk; TryAdvance = tryAdvance  }
```

Parallel Streams

```
let data = [| 1..1000000 |] |> Array.map int

data
|> ParStream.ofArray
|> ParStream.filter (fun x -> x % 2L = 0L)
|> ParStream.map (fun x -> x * x)
|> ParStream.sum
```

ParStream - Parallel Streams

```
type ParStream<'T> = (unit -> ('T -> unit)) -> unit

val ofArray : 'T[] -> ParStream<'T>
let ofArray values =
    fun thunk ->
        let forks =
            values
            |> partitions
            |> Array.map (fun p -> (p, thunk ()))
            |> Array.map (fun (p, k) -> fork p k)
        join forks
```

ParStream - Parallel Streams

```
type ParStream<'T> = (unit -> ('T -> unit)) -> unit

val map : ('T -> 'R) -> ParStream<'T> -> ParStream<'R>
let map f stream =
    fun thunk ->
        stream (fun () ->
                    let k = thunk ()
                    (fun v -> k (f v)))
```

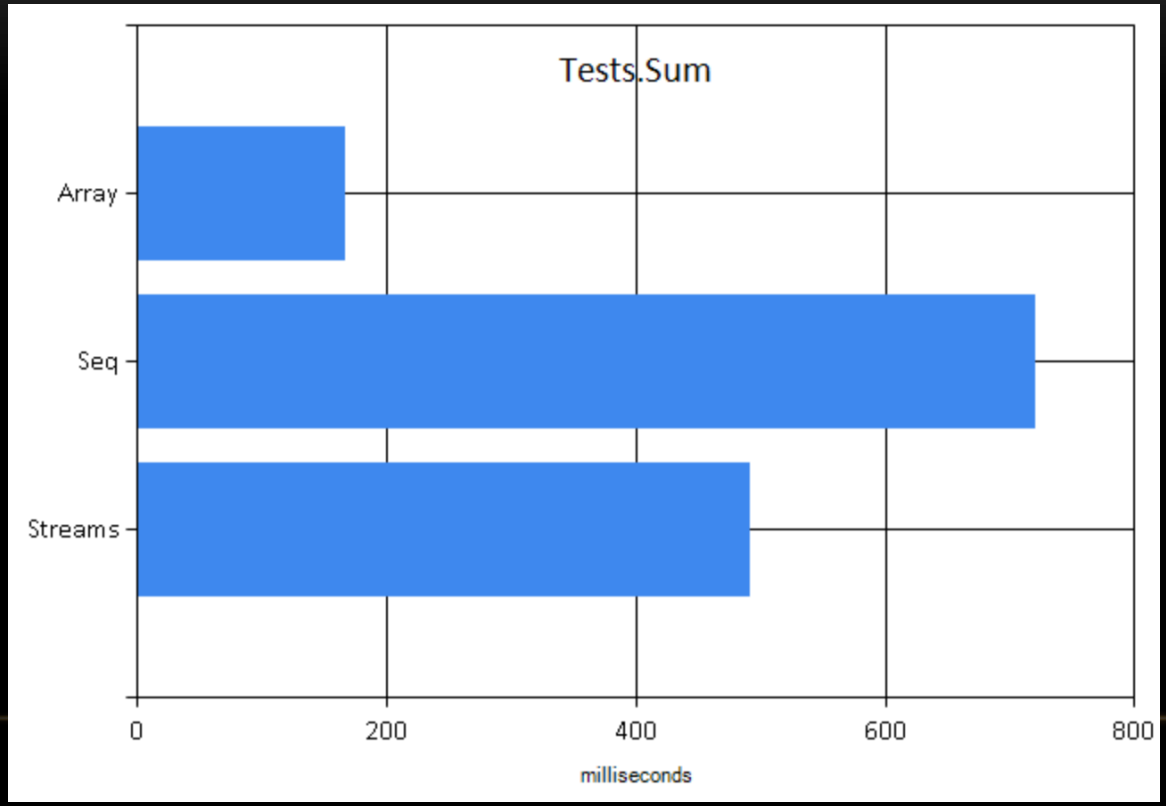
ParStream - Parallel Streams

```
type ParStream<'T> = (unit -> ('T -> unit)) -> unit

val sum : ParStream<'T> -> int
let sum stream =
    let array = new ResizeArray<int ref>()
    stream (fun () ->
                let sum = ref 0
                array.Add(sum)
                (fun v -> sum := sum + v)
            )
    array |> Array.map (fun sum -> !sum) |> Array.sum
```

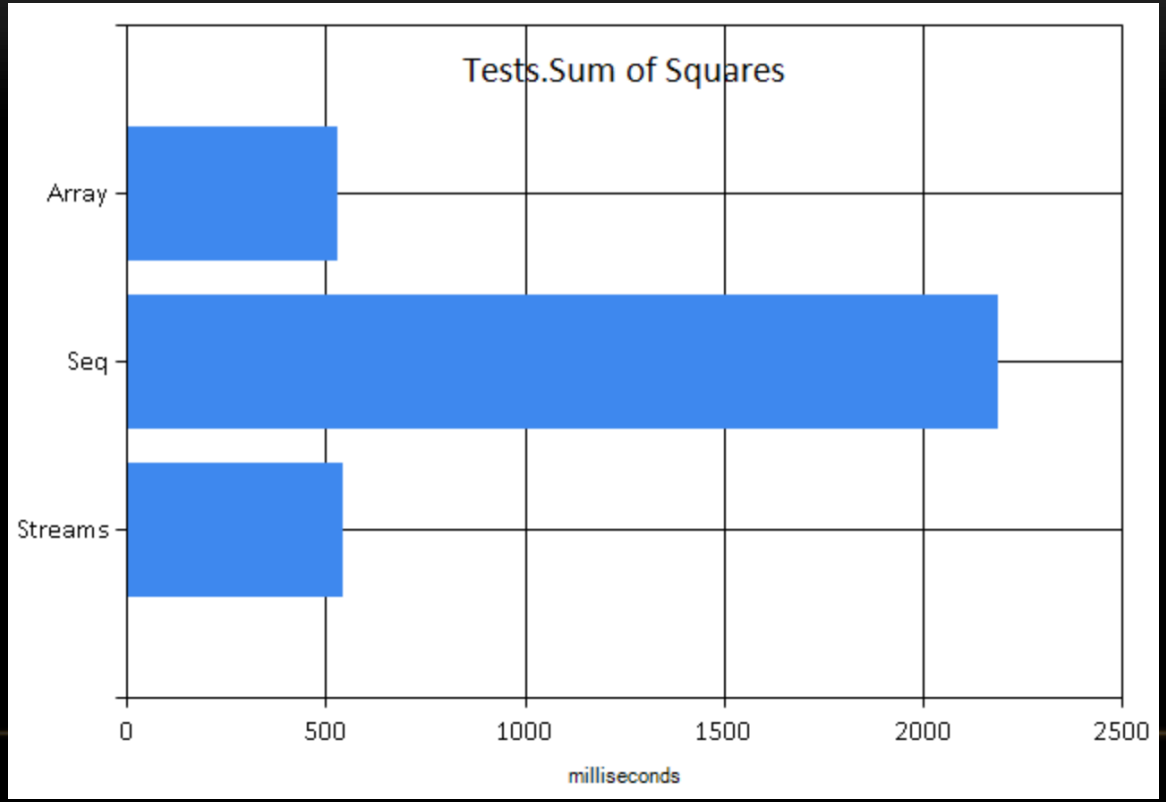
Benchmarks

i7 x 3.7 Ghz (4 physicall), 6 GB RAM

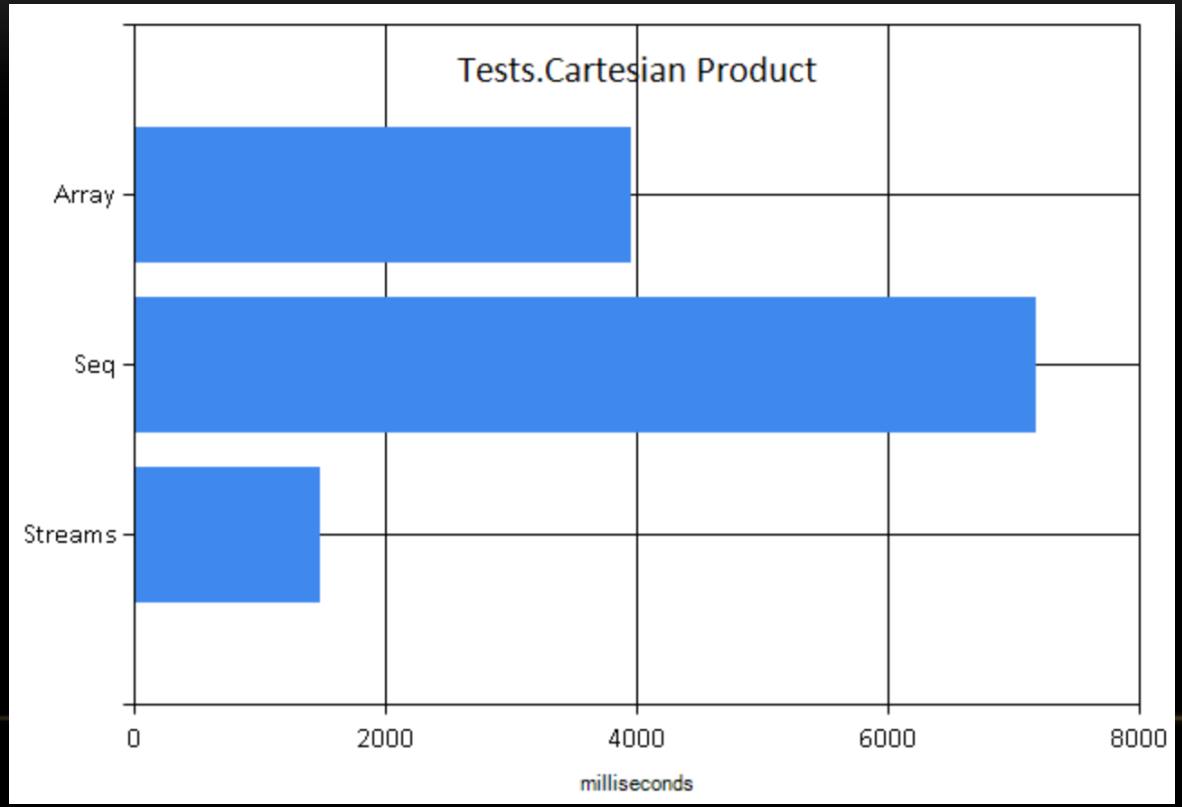
Sum



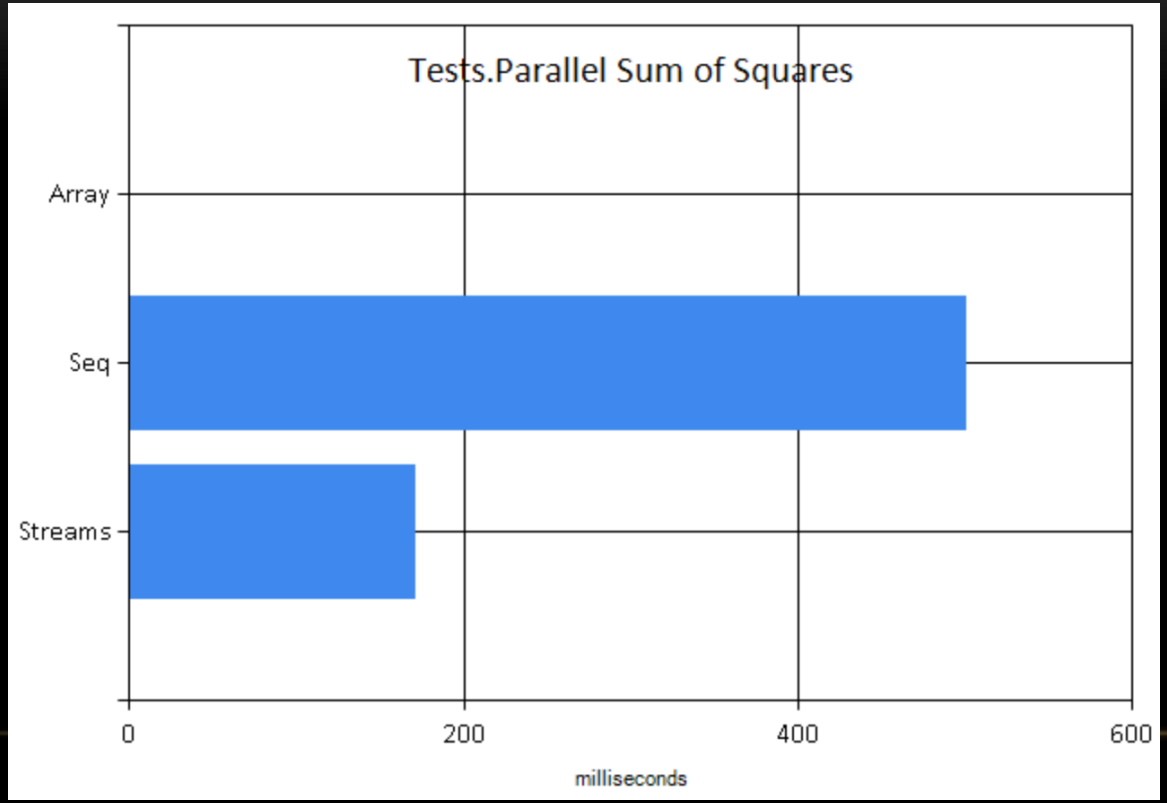
Sum of squares



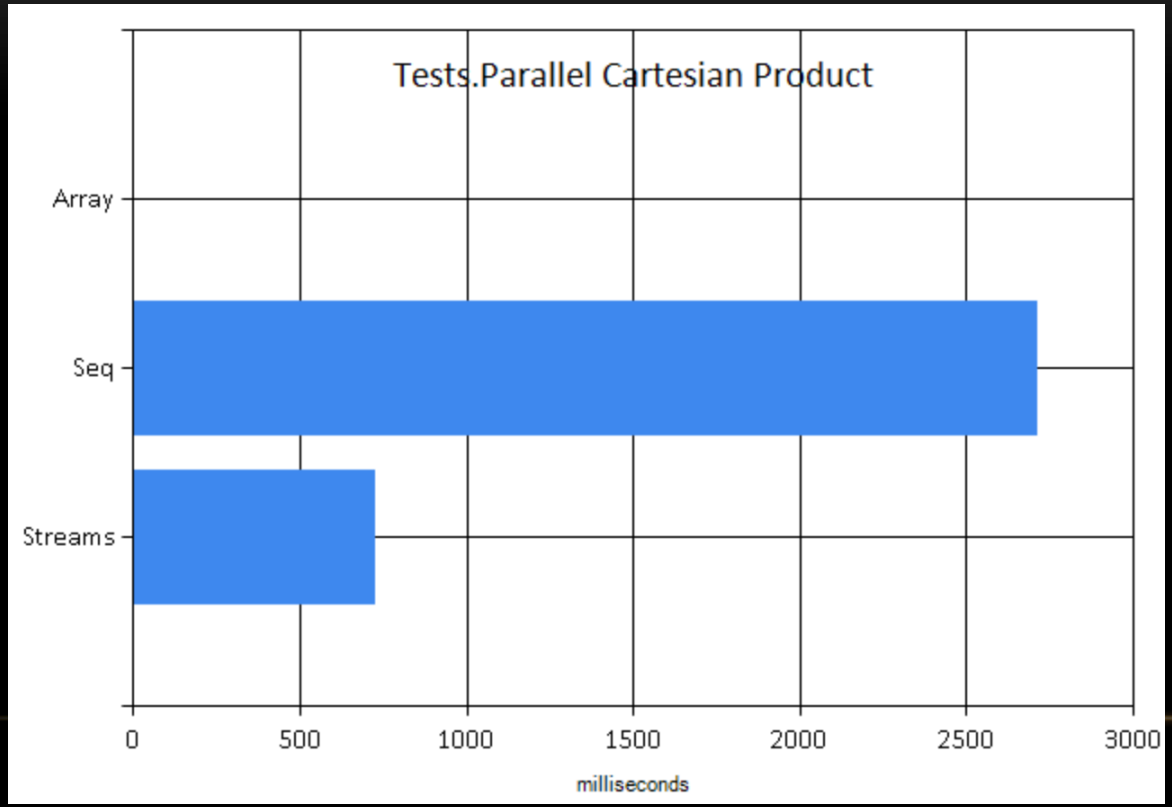
Cartesian Product



Parallel Sum of Squares



Parallel Cartesian Product




DEMO

Streams are lightweight and powerful

Sequential & Parallel

Beauty and the Beast

Write beautiful functional code which performs like imperative code

Almost…

Depends on the compiler’s ability to inline.

Can we make compilers smarter?

GOALS - CONCLUSION

Streams are lightweight and powerful

CPS is a great, useful but miss-used tool

- to extend functionality

There is always room to improve performance

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

REFERENCES

- https://github.com/nessos/Streams
- https://github.com/biboudis/sml-streams
- http://code.haskell.org/~dons/papers/icfp088-coutts.pdf
- https://github.com/nessos/LinqOptimizer
- http://arxiv.org/abs/1406.6631

HOW TO REACH ME

github.com/rikace/Presentations/FastStreams

@DCFsharp @TRikace

rterrell@microsoft.com


That's all Folks!