Expressive and Efficient Streaming Libraries

PhD Candidate: Aggelos Biboudis
PhD Advisor: Professor Yannis Smaragdakis

March 7, 2017
University of Athens
Processing sequences of tweets

tweetsDataset
  ▶ filter(t => t.contains("#phdlife"))
  ▶ filter(t => Sentiment.detectSentiment(t) == POSITIVE)
  ▶ map(t => t.User)
  ▶ take 15
  ▶ any(u => u.Followers > 1000)

1. pipe operator ✓
2. functionally-inspired ✓
3. demand-driven (lazy) ✓
4. possibly infinite ✓
   • is performance equivalent to for-loops?
Basics of a Streaming API

```plaintext
type α stream

Producers
val of_arr : α array → α stream
val unfold : (ζ → (α * ζ) option) → ζ → α stream

Transformers
val map : (α → β) → α stream → β stream
val filter : (α → bool) → α stream → α stream
val take : int → α stream → α stream
val flat_map : (α → β stream) → α stream → β stream
val zip_with : (α → β → γ) → (α stream → β stream → γ stream)

Consumer
val fold : (ζ → α → ζ) → ζ → α stream → ζ
```
Stream Origins

- Melvin Conway, 1963: Coroutines
  “separable programs”

- Douglas McIlroy, 1964: Unix Pipes
  `pipe()` implemented by Ken Thompson in v3, 1973
  ‘I’ leads to a “pipeline revolution” in v4

- Peter Landin, 1965: Streams
  “functional analogue of coroutines”
Fast-Forward 52 years

• iterators (‘yield’), generators as in Python, …

• LINQ, Java 8 Streams, …

• Lucid, LUSTRE, …

• Naiad, Flink, DryadLINQ, Spark Streaming, …

• Rx, Elm, …

• SIMD, …

• StreamIt, …

• Ziria, …
What we observe?

*same* pipeline on *different* languages has *different* performance characteristics

(part I)
Can we enhance streams for extensibility and performance?

1. Modularize the design of streams
   - On the library level (part II)
   - On the language level (part III)

2. Separate optimizations from the compiler
   - Stream fusion to completeness, as a library (part IV)
I. Assess performance

• Mainstream, VM-based, multi-paradigm PLs

• Scala, C#, F# share many similarities
  ◦ similar translation of lambdas
  ◦ similar design for streams

• While Java 8 took a different turn
\[ \sum_{i=0}^{n} a_i^2 \] pipelines

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

**Java**
```
public double sumOfSquaresSeq(double[] a) {
  double sum = DoubleStream.of(a)
    .map(a_i -> a_i * a_i)
    .sum();
  return sum;
}
```
Both styles conceptually

```java
Push<T> source(T[] arr) {
    return k -> {
        for (int i = 0; i < arr.length; i++)
            k(arr[i]);
    }
}
```

```java
Pull<T> source(T[] arr) {
    return new Pull<T>() {
        boolean hasNext() {..}
        T next() {..}
    };
}
```

```scala
Pull<Integer> sIt = source(v).map(i->i*i);
while (sIt.hasNext()) {
    el = sIt.next();
    /* consume el */
}
```

```java
Push<Integer> sFn = source(v).map(i->i*i);
sFn(el -> /* consume el */);
```
Benchmark: \[ \sum_{i=0}^{n} a_i^2 \]

**Sum of Squares**

(more sets in the dissertation)
But, push to pull in Java 8
(related to JDK-8075939 on bugs.openjdk.java.net)

```
Iterator<Long> iterator = Stream
    .of(v)
    .flatMap(x -> Stream.iterate(0L, i -> i + 2)
        .map(y -> x * y))
    .iterator();

iterator.hasNext();  // Out-of-memory :-(
```
And, pull/push perspectives
(on hotspot-compiler-dev mailing list)

perspectives on streams performance

John Rose john.r.rose at oracle.com
Fri Mar 6 01:01:20 UTC 2015

- Previous message: RFR(S) 8074010: followup to 8072383
- Next message: perspectives on streams performance
- Messages sorted by: [date] [thread] [subject] [author]

In order to get the full benefit from JDK 8 streams we will need to make the
I think of streams as a more concise and orderly replacement of classic "for"
A classic "for" loop is a external iterator notation: The iteration machinery
External iterators are easier to optimize, because their crucial iteration
HotSpot are less good at internal iterators. If the original point of the
II. Library-Level Extensibility

• **StreamAlg**: a library-design for streams

• “à la carte” behaviors to **control** the performance

• Also “mix” behaviors:
  • e.g., log a push, fuse a pull

  + Add new combinators

  + Development **without recompiling** the library
Object Algebras*

• Visitor is not sufficient
  ☑ adding new behaviors (semantics) ✓
  ☑ adding new variants (combinators) ✗

• e.g., expression (1 + (2 + 3)) using Object Algebras

```java
<Exp> Exp mkAnExp(ExpFactory<Exp> f) {
    return f.add(f.lit(1),
               f.add(f.lit(2), f.lit(3)));
}
```

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

interface ExecStreamAlg<E, C> extends StreamAlg<C> {
    <T> E<Long> count(C<T> stream);
    <T> E<T> fold(T identity,
                 BinaryOperator<T> accumulator,
                 C<T> stream);
}

class PushFactory implements StreamAlg<Push>
Create Pipelines

```java
E<-> s(ExecStreamAlg<E, C> alg) {
    return alg.sum(
        alg.map(x -> x * x),
        alg.source(v)));
}

s(new ExecPushFactory());

s(new ExecPullFactory());

s(new LogFactory<>()(new ExecFusedPullFactory()));

s(new LogFactory<>()(new ExecFusedPushFactory()));

s(new ExecFutureFactory<>()(new ExecPushFactory()).get());

s(new ExecFutureFactory<>()(new ExecPullFactory()).get());
```
Benchmarks

a) Abstraction does not interfere

b) Fusion is now pluggable

c) Pure pull-based vs push-to-pull in Java

d) Our pathological case from earlier

Part II
II. Language-Level Extensibility

- A **lightweight tool** to create Java **dialects**

- Extensions
  - Syntactic
  - Semantics

- e.g. implement a streaming library in Java, with **yield**
What the programmer writes
(1/3)

recaf Iter<Integer> alg = new Iter<Integer>();

recaf Iterable<Integer> filter(Iterable<Integer> iter,
                                Predicate<Integer> pred) {
    for (Integer t: iter) {
        if (pred.test(t)) {
            yield! t;
        }
    }
}

declaring the new semantics

using the new construct
What Recaf translates (2/3)

Iter<Integer> alg = new Iter<Integer>();

Iterable<Integer> filter(Iterable<Integer> iter,
    Predicate<Integer> pred) {
    return alg.Method(
        alg.ForEach(() -> iter,
            (t) -> alg.If(() -> pred.test(t),
                alg.Yield(() -> t))));
}

code is transformed into calls to methods on the semantics object
Where is Yield defined? (3/3)

```java
public class Iter<R> implements EvalJavaStmt<R>, JavaMethodAlg<Iterable<R>, SD<R>> {
    public <U> SD<R> Yield(ISupply<U> exp) {
        return (label, rho, sigma, brk, contin, err) -> {
            get(exp).accept(v -> {
                YIELD.value = v;
                YIELD.k = sigma;
                throw YIELD;
            }, err);
        };
    }

    ...
}
```
**IV. Stream Fusion, to Completeness**

- **Strymonas**: a **library** for **fused streams** …

- … that supports a wide range and **complex** combinations of operators …

- … and generates **loop-based, fused code with zero allocations**.
Staging Stream Fusion

```
of_arr .{arr}.
  ▶ map (fun x → .(~x * ~x).)
  ▶ sum

let s_1 = ref 0 in
let arr_2 = arr in
  for i_3 = 0 to Array.length arr_2 - 1 do
    let el_4 = arr_2.(i_3) in
    let t_5 = el_4 * el_4 in
    s_1 := t_5 + !s_1
  done;
!s_1
```
Staging Stream Fusion

and much more complex...

```plaintext
zip_with (fun e1 e2 → .{(~e1,~e2)}.)
(of_arr .{arr1}. (* 1st stream *))
  ▶ map (fun x → .{~x * ~x}.)
  ▶ take .{12}.
  ▶ filter (fun x → .{~x mod 2 = 0}.)
  ▶ map (fun x → .{~x * ~x}.)

(iota .{1}). (* 2nd stream *)
  ▶ flat_map (fun x → iota .{~x+1}. ▶ take .{3}.)
  ▶ filter (fun x → .{~x mod 2 = 0}.)
  ▶ fold (fun z a → .{~a :: ~z}.) .{[]}.
```
Benchmarks

OCaml/BER MetaOCaml

Part IV
Part IV
Multi-Stage Programming

- manipulate code templates
- brackets to create well-formed, scoped, typed templates
  
  ```
  let c = .< 1 + 2 >.
  ```

- create holes
  
  ```
  let cf x = .< .~x + .~x >.
  ```

- synthesize code at staging-time (runtime)
  
  ```
  cf c ~> .< (1 + 2) + (1 + 2) >.
  ```
Naive Staging

\[
\text{type } \alpha \text{ stream } = \exists \sigma. \sigma * (\sigma \rightarrow (\alpha, \sigma) \text{ stream_shape})
\]

\[
\text{type ('a,'z) stream_shape} = \\
| \text{Nil} \\
| \text{Cons of 'a * 'z}
\]

based on unfoldr: functional analogue of iterators
Naive Staging

binding-time analysis

type $\alpha$ stream = $\exists \sigma. \sigma$ code $\ast$ ($\sigma$ code $\rightarrow$ ($\alpha, \sigma$) stream_shape code)

classify variables as static and dynamic
let map : ('a code -> 'b code) -> 'a stream -> 'b stream =
  fun f (s, step) ->
    let new_step = fun s ->
      match step s with
      | Nil       -> Nil
      | Cons (a, t) -> Cons (f a, t)
    in (s, new_step);;
let rec loop_1 z_2 s_3 =
  match match match s_3 with
  | (i_4, arr_5) ->
    if i_4 < (Array.length arr_5)
    then Cons ((arr_5.(i_4)),((i_4 + 1), arr_5))
    else Nil
  with
  | Nil -> Nil
  | Cons (a_6, t_7) -> Cons ((a_6 * a_6), t_7)
  with
  | Nil -> z_2
  | Cons (a_8, t_9) -> loop_1 (z_2 + a_8) t_9
Factor out static knowledge:
After 3 key domain-specific optimizations*

1. The structure of the stepper is known:
   use that at staging time!

2. The structure of the state is known:
   use that at staging time, too!

3. Tail recursion vs Iteration:
   modularize the loop structure (**for** vs **while**)

* 6 domain-specific optimizations in total, accommodating linearity (filter and flat_map), sub-ranging, infinite streams (take and unfold), and parallel stream fusion (zip)
let s_1 = ref 0 in
let arr_2 = [|0;1;2;3;4|] in
for i_3 = 0 to (Array.length arr_2) - 1 do
  let el_4 = arr_2.(i_3) in
  let t_5 = el_4 * el_4 in s_1 := !s_1 + t_5
done;
!s_1
Applications

• **StreamAlg design**
  ✓ pluggable streams
  ✓ pluggable optimizers
  ✓ pluggable database engines

• **Recaf**
  ✓ generative or interpretive
  ✓ PL playground
  ✓ embedding libraries

• **Strymonas**
  ✓ general purpose, fast library
  ✓ evolve it for HPC + data parallelism + multidimensional data
Current Limitations

- **StreamAlg**
  - in Java is verbose due to lack of HKT, not in Scala
- **Recaf**
  - interpretation is slow, not for generation or embeddings
  - not modularly type safe
- **Strymonas**
  - MetaOCaml and LMS are not “main branch”
  - MetaOCaml annotations may confuse (LMS doesn’t have)
  - streams are not reusable (as in Java 8 Streams)
Lessons/Contributions

• We can enhance streams with modularity & separation and maintain a high-level structure!

• Evolving the streaming library only:
  *
  * interpretations and optimizations are pluggable
  *
  * domain-specific optimizations in “active” Stream APIs instead of “sufficiently-smart compilers”
Papers/Teams

- **Clash of the Lambdas**, A. Biboudis, N. Palladinos and Y. Smaragdakis. ICOOOLPS’14 — [github.com/biboudis/clashofthelambdas](http://github.com/biboudis/clashofthelambdas)

- **Streams à la carte: Extensible Pipelines with Object Algebras**, A. Biboudis, N. Palladinos, G. Fourtounis and Y. Smaragdakis. ECOOP’15 — [github.com/biboudis/streamalg](http://github.com/biboudis/streamalg)


- **Stream Fusion, to Completeness**, O. Kiselyov, A. Biboudis, N. Palladinos and Y. Smaragdakis. POPL’17 — [github.com/strymonas](http://github.com/strymonas)