Stream Fusion, to Completeness

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

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

Nick Palladinos
Nessos IT S.A. Athens, Greece
npal@nessos.gr

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

Abstract

Stream processing is mainstream (again): Widely-used stream libraries are now available for virtually all modern OO and functional languages, from Java to C# to Scala to OCaml to Haskell. Yet expressivity and performance are still lacking. For instance, the popular, well-optimized Java 8 streams do not support the `zip` operator and are still an order of magnitude slower than hand-written loops.

We present the first approach that represents the full generality of stream processing and eliminates overheads, via the use of staging. It is based on an unusually rich semantic model of stream interaction. We support *any* combination of zipping, nesting (or flat-mapping), sub-ranging, filtering, mapping—of finite or infinite streams. Our model captures idiosyncrasies that a programmer uses in optimizing stream pipelines, such as rate differences and the choice of a “for” vs. “while” loops. Our approach delivers hand-written–like code, but automatically. It explicitly avoids the reliance on black-box optimizers and sufficiently-smart compilers, offering highest, guaranteed and portable performance.

Our approach relies on high-level concepts that are then readily mapped into an implementation. Accordingly, we have two distinct implementations: an OCaml stream library, staged via MetaOCaml, and a Scala library for the JVM, staged via LMS. In both cases, we derive libraries richer and simultaneously many tens of times faster than past work. We greatly exceed in performance the standard stream libraries available in Java, Scala and OCaml, including the well-optimized Java 8 streams.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Code Generation; D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms Languages, Performance

Keywords Code generation, multi-stage programming, optimization, stream fusion, streams

1. Introduction

Stream processing defines a pipeline of operators that transform, combine, or reduce (even to a single scalar) large amounts of data. Characteristically, data is accessed strictly linearly rather than randomly and repeatedly—and processed uniformly. The upside of the limited expressiveness is the opportunity to process large amount of data efficiently, in constant and small space. Functional stream libraries let us easily build such pipelines, by composing sequences of simple transformers such as `map` or `filter` with producers (backed by an array, a file, or a generating function) and consumers (reducers). The purely applicative approach of building a complex pipeline from simple immutable pieces simplifies programming and reasoning: the assembled pipeline is an executable specification. To be practical, however, a library has to be efficient: at the very least, it should avoid creating intermediate structures (files, lists, etc.) whose size grows with the length of the stream.

Most modern programming languages—Java, Scala, C#, F#, OCaml, Haskell, Clojure, to name a few—currently offer functional stream libraries. They all provide basic mapping and filtering. Handling of infinite, nested or parallel (zipping) streams is rare—especially all in the same library. Although all mature libraries avoid unbounded intermediate structures, they all suffer, in various degrees, from the overhead of abstraction and compositionality: extra function calls, the creation of closures, objects and other bounded intermediate structures.

An excellent example is the Java 8 Streams, often taken as the standard of stream libraries. It stresses performance: e.g., streaming from a known source, such as an array, amounts to an ordinary loop, well-optimized by a Java JIT compiler [3]. However, Java 8 Streams are still much slower than hand-optimized loops for non-trivial pipelines (e.g., over 10x slower on the standard cartesian product benchmark [2]). Furthermore, the library cannot handle (‘zip’) several streams in parallel[1] and cannot deal with nesting of infinite streams. These are not mere omissions: infinite nested streams demand a different iteration model, which is hard to efficiently implement with a simple loop.

This paper presents ***strymonas***: a streaming library design that offers both high expressivity and *guaranteed*, highest performance. First, we support the full range of streaming operators (a.k.a. stream *transformers* or *combinators*) from past libraries: not just `map` and `filter` but also sub-ranging (`take`), nesting (`flat_map`—a.k.a. `concatMap`) and parallel (`zip_with`) stream processing. All operators are freely composable: e.g., `zip_with` and `flat_map` can be

[1] One could emulate `zip` using `iterator` from push-streams—at significant drop in performance.

used together, repeatedly, with finite or infinite streams. Our novel stream representation captures the essence of stream processing for virtually all combinators examined in past literature.

Second, our stream representation allows eliminating the abstraction overhead altogether, for the full set of stream operators. We perform *stream fusion* (§3) and other aggressive optimization. The generated code contains no extra heap allocations in the main loop (Thm.1). By not generating tuples or other objects, we avoid the overhead of dynamic object construction and pattern-matching, and also the hidden, often significant overhead of memory pressure and boxing of primitive types. The result not merely approaches but attains the performance of hand-optimized code, from the simplest to the most complex cases, up to *well over* the complexity point where hand-written code becomes infeasible. Although the library combinators are purely functional and freely composable, the actual running stream code is loop-based, highly tangled and imperative.

Our technique relies on staging (§4.1), a form of metaprogramming, to achieve guaranteed stream fusion. This is in contrast to past use of source-to-source transformations of functional languages [14], of AST run-time rewriting [21, 22], compile-time macros [25] or Haskell GHC RULES [5, 23] to express domain-specific streaming optimizations. Rather than relying on an optimizer to eliminate artifacts of stream composition, we do not introduce the artifacts in the first place. Our library transforms highly abstract stream pipelines to code fragments that use the most suitable imperative features of the host language. The appeal of staging is its certainty and guarantees. Unlike the aforementioned techniques, staging also ensures that the generated code is well-typed and well-scoped, by construction. We discuss the trade-offs of staging in §9.

Our work describes a general approach, and not just a single library design. To demonstrate the generality of the principles, we implemented two library versions [2], in diverse settings. The first is an OCaml library, staged with BER MetaOCaml [17]. The second is a Scala library (also usable by client code in Java and other JVM languages), staged with Lightweight Modular Staging (LMS) [26].

We evaluate ***strymonas*** on a suite of benchmarks (§7), comparing with hand-written code as well as with other stream libraries (including Java 8 Streams). Our staged implementation is up to more than two orders-of-magnitude faster than standard Java/Scala/OCaml stream libraries, matching the performance of hand-optimized loops. (Indeed, we occasionally had to improve hand-written baseline code, because it was slower than the library.)

Thus, our contributions are: (i) the principles and the design of stream libraries that support the widest set of operations from past libraries and also permit the full elimination of abstraction overhead. The main principle is a novel representation of streams that captures rate properties of stream transformers and the form of termination conditions, while separating and abstracting components of the entire stream state. This decomposition of the essence of stream iteration is what allows us to perform very aggressive optimization, via staging, regardless of the streaming pipeline configuration. (ii) The implementation of the design in terms of two distinct library versions for different languages and staging methods: OCaml/MetaOCaml and Scala/JVM/LMS.

2. Overview: A Taste of the Library

We first give an overview of our approach, presenting the client code (i.e., how the library is used) alongside the generated code (i.e., what our approach achieves). Although we have implemented two separate library versions, one for OCaml and one for Scala/JVM languages, for simplicity, all examples in the paper will be in (Meta)OCaml, which was also our original implementation.

```
Stream representation (abstract)
type α stream

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

Consumer
val fold : (ζ code → α code → ζ code) →
           ζ code → α stream → ζ code

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

Figure 1: The library interface

For the sake of exposition, we take a few liberties with the OCaml notation, simplifying the syntax of the universal and existential quantification and of sum data types with record components. (The latter simplification—inline records—is supported in the latest, 4.03, version of OCaml.) The paper is accompanied by the complete code for the ***strymonas*** library (as an open-source repository), also including our examples, tests, and benchmarks.

MetaOCaml is a dialect of OCaml with staging annotations .⟨e⟩. and ∼e, and the code type [17, 34]. In the Scala version of our library, staging annotations are implicit: they are determined by inferred types. Staging annotations are optimization directives, guiding the partial evaluation of library expressions. Thus, staging annotations are not crucial to understanding what our library can express, only how it is optimized. On first read, staging annotations may be simply disregarded. We get back to them, in detail, in §4.1.

The (Meta)OCaml library interface is given in Figure 1. The library includes stream producers (one generic—`unfold`, and one specifically for arrays—`of_arr`), the generic stream consumer (or stream reducer) `fold`, and a number of stream transformers. Ignoring `code` annotations, the signatures are standard. For instance, the generic `unfold` combinator takes a function from a state, ζ, to a value α and a new state (or nothing at all), and, given an initial state ζ, produces an opaque stream of αs.

The first example is summing the squares of elements of an array `arr`—in mathematical notation, $\sum a_i^2$. The code

```
let sum = fold (fun z a → .⟨∼a + ∼z⟩.) .⟨0⟩.

of_arr .⟨arr⟩.
  ▷ map (fun x → .⟨∼x * ∼x⟩.)
  ▷ sum
```

is not far from the mathematical notation. Here, ▷, like the similar operator in F#, is the inverse function application: argument to the left, function to the right. The stream components are first-class and hence may be passed around, bound to identifiers and shared; in short, we can build libraries of more complex components.
In this simple example, the generated code is understandable:

```
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
```

[2] https://strymonas.github.io/.

It is relatively easy to see which part of the code came from which part of the pipeline "specification". The generated code has no closures, tuples or other heap-allocated structures: it looks as if it were hand-written by a competent OCaml programmer. The iteration is driven by the source operator, `of_arr`, of the pipeline. This is precisely the iteration pattern that Java 8 streams optimize. As we will see in later examples, this is but one of the optimal iteration patterns arising in stream pipelines.

The next example sums only some elements:

```
let ex = of_arr .⟨arr⟩. ▷ map (fun x → .⟨∼x * ∼x⟩.)

ex ▷ filter (fun x → .⟨∼x mod 17 > 7⟩.) ▷ sum
```

We have abstracted out the mapped stream as `ex`. The earlier example is, hence, `ex ▷ sum`. The current example applies `ex` to the more complex summator that first filters out elements before summing the rest. The next example limits the number of summed elements to a user-specified value `n`

```
ex ▷ filter (fun x → .⟨∼x mod 17 >7⟩.)
   ▷ take .⟨n⟩.
   ▷ sum
```

We stress that the limit is applied to the filtered stream, not to the original input; writing this example in mathematical notation would be cumbersome. The generated code

```
let s_1 = ref 0 in
let arr_2 = arr in
let i_3 = ref 0 in
let nr_4 = ref n in
while !nr_4 > 0 && !i_3 ≤ Array.length arr_2 -1 do
   let el_5 = arr_2.(! i_3) in
   let t_6 = el_5 * el_5 in
   incr i_3;
   if t_6 mod 17 > 7
   then (decr nr_4; s_1 := t_6+ !s_1)
done; ! s_1
```

again looks as if it were handwritten, by a competent programmer. However, compared to the first example, the code is more tangled; for example, the `take .⟨n⟩.` part of the pipeline contributes to three separate places in the code: where the `nr_4` reference cell is created, tested and mutated. The iteration pattern is more complex. Instead of a `for` loop there is a `while`, whose termination conditions come from two different pipeline operators: `take` and `of_arr`.

The dot-product of two arrays `arr1` and `arr2` looks just as simple

```
zip_with (fun e1 e2 → .⟨∼e1 * ∼e2⟩.)
         (of_arr .⟨arr1⟩.)
         (of_arr .⟨arr2⟩.) ▷ sum
```

showing off the zipping of two streams, with the straightforward, again hand-written quality, generated code:

```
let s_17 = ref 0 in
let arr_18 = arr1 in let arr_19 = arr2 in
 for i_20 = 0 to
  min (Array.length arr_18 -1)
      (Array.length arr_19 -1) do
   let el_21 = arr_18.(i_20) in
   let el_22 = arr_19.(i_20) in
   s_17 := el_21 * el_22 + !s_17
 done; ! s_17
```

The optimal iteration pattern is different still (though simple): the loop condition as well as the loop body are equally influenced by two `of_arr` operators.

In the final, complex example we zip two complicated streams. The first is a finite stream from an array, mapped, subranged, filtered and mapped again. The second is an infinite stream of natural numbers from 1, with a filtered flattened nested substream. After zipping, we fold everything into a list of tuples.

```
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⟩.) .⟨[]⟩.
```

We did not show any types, but they exist (and have been inferred). Therefore, an attempt to use an invalid operation on stream elements (like concatenating integers or applying an ill-fitting stream component) will be immediately rejected by the type-checker.

Although the above pipeline is purely functional, modular and rather compact, the generated code (shown in Appendix A) is large, entangled and highly imperative. Writing such code correctly by hand is clearly challenging.

3. Stream Fusion Problem

The key to an expressive and performant stream library is a representation of streams that fully captures the generality of streaming pipelines and allows desired optimizations. To understand how the representation affects implementation and optimization choices, we review past approaches. We see that, although some of them take care of the egregious overhead, none manage to eliminate all of it: the assembled stream pipeline remains slower than hand-written code.

The most straightforward representation of streams is a linked list, or a file, of elements. It is also the least performing. The first example in §2, of summing squares, will entail: (1) creating a stream from an array by copying all elements into it; (2) traversing the list creating another stream, with squared elements; (3) traversing the result, summing the elements. We end up creating three intermediate lists. Although the whole processing still takes time linear in the size of the stream, it requires repeated traversals and the production of linear-size intermediate structures. Also, this straightforward representation cannot cope with sources that are always ready with an element: "infinite streams".

The problem, thus, is deforestation [35]: eliminating intermediate, working data structures. For streams, in particular, deforestation is typically called "stream fusion". One can discern two main groups of stream representations that let us avoid building intermediate data structures of unbounded size.

Push Streams. The first, heavily algebraic approach, represents a stream by its reducer (the fold operation) [20]. If we introduce the "shape functor" for a stream with elements of type α as

```
type (α,ζ) stream_shape =
  | Nil
  | Cons of α * ζ
```

then the stream is formally defined as:[3]

```
type α stream = ∀ω. ((α,ω) stream_shape → ω) → ω
```

A stream of αs is hence a function with the ability to turn any generic "folder" (i.e., a function from `(α,ω) stream_shape` to ω) to a single ω. The "folder" function is formally called an `F`-algebra for the `(α,-) stream_shape` functor.

For instance, an array is easily representable as such a fold:

[3] Strictly speaking, `stream` should be a record type: in OCaml, only record or object components may have the type with explicitly quantified type variables. For the sake of clarity we lift this restriction in the paper.

```
let of_arr : α array → α stream =
   fun arr → fun folder →
   let s = ref (folder Nil) in
   for i=0 to Array.length arr - 1 do
     s := folder (Cons (arr.(i),!s))
   done; !s
```

Reducing a stream with the reducing function `f` and the initial value `z` is especially straightforward in this representation:

```
let fold : (ζ → α → ζ) → ζ → α stream → ζ =
 fun f z str →
  str (function Nil → z | Cons (a,x) → f x a)
```

More germane to our discussion is that mapping over the stream (as well as `filter`-ing and `flat_map`-ing) are also easily expressible, without creating any variable-size intermediate data structures:

```
let map : (α → β) → α stream → β stream =
 fun f str →
  fun folder → str (fun x → match x with
  | Nil         → folder Nil
  | Cons (a,x) → folder (Cons (f a,x)))
```

A stream element `a` is transformed "on the fly" without collecting in working buffers. Our sample squaring-accumulating pipeline runs in constant memory now. Deforestation, or stream fusion, has been accomplished. The simplicity of this so-called "push stream" approach makes it popular: it is used, for example, in the reducers of Clojure as well as in the OCaml "batteries" library. It is also the basis of Java 8 Streams, under an object-oriented reformulation of the same concepts.

In push streams, it is the stream producer, e.g., `of_arr`, that drives the optimal execution of the stream. Implementing `take` and other such combinators that restrict the processing to a prefix of the stream requires extending the representation with some sort of a "feedback" mechanism (often implemented via exceptions). Where push streams stumble is the zipping of two streams, i.e., the processing of two streams in parallel. This simply cannot be done with constant per-element processing cost. Zipping becomes especially complicated (as we shall see in §6.3) when the two pipelines contain nested streams and hence produce elements at generally different rates.[4]

Pull Streams. An alternative representation of streams, pull streams, has a long pedigree, all the way from the generators of Alphard [28] in the '70s. These are objects that implement two methods: `init` to initialize the state and obtain the first element, and `next` to advance the stream to the next element, if any. Such a "generator" (or IEnumerator, as it has come to be popularly known) can also be understood algebraically—or rather, co-algebraically. Whereas push streams represent a stream as a fold, pull streams, dually, are the expression of an *unfold* [8, 20]:[5]

```
type α stream = ∃σ. σ * (σ → (α,σ) stream_shape)
```

The stream is, hence, a pair of the current state and the so-called "step" function that, given a state, reports the end-of-stream condition `Nil`, or the current element and the next state. (Formally, the step function is the F-co-algebra for the $(\alpha,\text{-})$ `stream_shape` functor.) The existential quantification over the state keeps it private: the only permissible operation is to pass it to the step function.

When an array is represented as a pull stream, the state is the tuple of the array and the current index:

```
let of_arr : α array → α stream =
  let step (i,arr) =
    if i < Array.length arr
       then Cons (arr.(i), (i+1,arr)) else Nil
  in fun arr → ((0,arr),step)
```

The step function—a pure combinator rather than a closure—dereferences the current element and advances the index. Reducing the pull stream now requires an iteration, of repeatedly calling `step` until it reports the end-of-stream. (Although the types of `of_arr`, `fold`, and `map`, etc. nominally remain the same, the meaning of α `stream` has changed.)

```
let fold : (ζ → α → ζ) → ζ → α stream → ζ =
 fun f z (s,step) →
  let rec loop z s = match step s with
  | Nil        → z
  | Cons (a,t) → loop (f z a) t
  in loop z s
```

With pull streams, it is the reducer, i.e., the stream consumer, that drives the processing. Mapping over the stream

```
let map : (α → β) → α stream → β 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)
```

merely transforms its step function: `new_step` calls the old step and `map`s the returned current element, passing it immediately to the consumer, with no buffering. That is, like push streams, pull streams also accomplish fusion. Befitting their co-algebraic nature, pull streams can represent both finite and infinite streams. Stream combinators, like `take`, that cut evaluation short are also easy. On the other hand, skipping elements (filtering) and nested streaming is more complex with pull streams, requiring the generalization of the `stream_shape`, as we shall see in §6. The main advantage of pull streams over push streams is in expressiveness: pull streams have the ability to process streams in parallel, enabling `zip_with` as well as more complex stream merging. Therefore, we take pull streams as the basis of our library.

Imperfect Deforestation. Both push and pull streams eliminate the intermediate lists (variable-size buffers) that plague a naive implementation of the stream library. Yet they do not eliminate all the abstraction overhead. For example, the `map` stream combinator transforms the current stream element by passing it to some function `f` received as an argument of `map`. A hand-written implementation would have no other function calls. However, the pull-stream `map` combinator introduces a closure: `new_step`, which receives a `stream_shape` value from the old `step`, pattern-matches on it and constructs the new `stream_shape`. The push-stream `map` has the same problem: The step function of `of_arr` unpacks the current state and then packs the array and the new index again into the tuple. This repeated deconstruction and construction of tuples and co-products is the abstraction overhead, which a complete deforestation should eliminate, but pull and push streams, as commonly implemented, do not. Such "constant" factors make library-assembled stream processing much slower than the hand-written version (by up to two orders of magnitude—see §7).

4. Staging Streams

A well-known way of eliminating abstraction overhead and delivering "abstraction without guilt" is program generation: compiling

[4] The Reactive Extensions (Rx) framework [1] gives a real-life example of the complexities of implementing `zip`. Rx is push-based and supports `zip` at the cost of maintaining an unbounded intermediate queue. This deals with the "backpressure in Zip" issue, extensively-discussed in the Rx github repo. Furthermore, Rx seems to have abandoned blocking zip implementations since 2014.

[5] For the sake of explanation, we took another liberty with the OCaml notation, avoiding the GADT syntax for the existential.

a high-level abstraction into efficient code. In fact, the original deforestation algorithm in the literature [35] is closely related to partial evaluation [30]. This section introduces staging: one particular, manual technique of partial evaluation. It lets us achieve our goal of eliminating all abstraction overhead from the stream library. Perfect stream fusion with staging is hard: §4.2 shows that straightforward staging (or automated partial evaluation) does not achieve full deforestation. We have to re-think general stream processing (§5).

4.1 Multi-Stage Programming

Multi-stage programming (MSP), or *staging* for short, is a way to write programs that generate programs. MSP may be thought of as a principled version of the familiar "code templates", where the templates ensure by their very construction that the generated code is not only syntactically well-formed but also well-scoped and well-typed.

In this paper we use BER MetaOCaml [17], which is a dialect of OCaml with MSP extensions. The first MSP feature is brackets, .⟨ and ⟩., which enclose a code template. For example, `.⟨1+ 2⟩.` is a template for generating code to add two literals 1 and 2.

```
let c = .⟨1 + 2⟩.
⇝  val c : int code = .⟨1 + 2⟩.
```

The output of the interpreter demonstrates that the code template is a first-class object; moreover, it is a value: a *code value*. MetaOCaml can print such values, and also write them into a file to compile it later. The code value is typed: our sample template generates integer-valued code.

As behooves templates, they can have holes to splice-in other templates. The splicing MSP feature, ∼, is called an *escape*. In the following example, the template `cf` has two holes, to be filled in with the same expression. Then `cf c` fills the holes with the expression `c` created earlier.

```
let cf x = .⟨∼x + ∼x⟩.
⇝  val cf : int code → int code = <fun>
cf c
⇝  - : int code = .⟨(1 + 2) + (1 + 2)⟩.
```

One may regard brackets and escapes as annotating code: which portions should be evaluated as usual (at the present stage, so to speak) and which in the future (when the generated code is compiled and run).

4.2 Simple Staging of Streams

We can turn a library into, effectively, a compiler of efficient code by adding staging annotations. This is not a simple matter of annotating one of the standard definitions (either pull- or push-style) of α `stream`, however. To see this, we next consider staging a set of pull-stream combinators. Staging helps with performance, but the abstraction overhead still remains.

The first step in using staging is the so-called "binding-time analysis": finding out which values can be known only at run-time ("dynamically") and what is known already at code-generation time, ("statically") and hence can be pre-computed. Partial evaluators perform binding-time analysis, with various degrees of sophistication and success, automatically and opaquely. In staging, binding-time analysis is manual and explicit.

We start with the pull streams `map` combinator, which, recall, has a type signature:

```
type α stream = ∃σ. σ * (σ → (α,σ) stream_shape)
val map : (α → β) → α stream → β stream
```

Its first argument, the mapping function `f`, takes the current stream element, which is clearly not known until the processing pipeline is run. The result is likewise dynamic. However, the mapping operation itself can be known statically. Hence the staged `f` may be given the type `α code → β code`: given code to compute αs, the mapping function, `f`, is a static way to produce code to compute βs.

The second argument of `map` is the pull stream, a tuple of the current state (σ) and the step function. The state is not known statically. The result of the step function depends on the current state and, hence, is fully dynamic. The step function itself, however, can be statically known. Hence we arrive at the following type of the staged stream

```
type α st_stream =
  ∃σ. σ code * (σ code → (α,σ) stream_shape code)
```

Having done such binding-time analysis for the arguments of the `map` combinator, it is straightforward to write the staged `map`, by annotating—i.e., placing brackets and escapes on—the original `map` code according to the decided binding-times:

```
let map : (α code → β code) →
           α st_stream → β st_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)
```

The combinators `of_arr` and `fold` are staged analogously. We use the method of [11] to prove the correctness, which easily applies to this case, given that `map` is non-recursive. The sample processing pipeline (the first example from §2)

```
of_arr .⟨[|0;1;2;3;4|]⟩.
       ▷ map (fun a → .⟨∼a * ∼a⟩.)
       ▷ fold (fun x y → .⟨∼x + ∼y⟩.) .⟨0⟩.
```

then produces the following code:

```
- : int code = .⟨
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 in
loop_1 0 (0, [|0;1;2;3;4|])⟩.
```

As expected, no lists, buffers or other variable-size data structures are created. Some constant overhead is gone too: the squaring operation of `map` is inlined. However, the triple-nested `match` betrays the remaining overhead of constructing and deconstructing `stream_shape` values. Intuitively, the clean abstraction of streams (encoded in the pull streams type of α `stream`) isolates each operator from others. The result does not take advantage of the property that, for this pipeline (and others of the same style), the looping of all three operators (`of_arr`, `map`, and `fold`) will synchronize, with all of them processing elements until the same last one. Eliminating the overhead requires a different computation model for streams.

5. Eliminating All Abstraction Overhead in Three Steps

We next describe how to purge all of the stream library abstraction overhead and generate code of hand-written quality and performance. We will be continuing the simple running example of the earlier sections, of summing up squared elements of an array. (§6 will later lift the same insights to more complex pipelines.) As in §4.2, we will be relying on staging to generate well-formed and well-typed code. The key to eliminating abstraction overhead from the generated code is to move it to a generator, by making the generator take better advantage of the available static knowledge. This is easier said than done: we have to use increasingly more sophisticated transformations of the stream representation to expose more static information and make it exploitable. The three transformations we show next require more-and-more creativity and domain knowledge, and cannot be performed by a simple tool, such as an automated partial evaluator. In the process, we will identify three interesting concepts in stream processing: the structure of iteration (§5.1), the state kept (§5.2), and the optimal kind of loop construct and its contributors (§5.3).

5.1 Fusing the Stepper

Modularity is the cause of the abstraction overhead we observed in §4.2: structuring the library as a collection of composable components forces them to conform to a single interface. For example, each component has to use the uniform stepper function interface (see the `st_stream` type) to report the next stream element or the end of the stream. Hence, each component has to generate code to examine (deconstruct) and construct the `stream_shape` data type.

At first glance, nothing can be done about this: the result of the step function, whether it is `Nil` or a `Cons`, depends on the current state, which is surely not known until the stream processing pipeline is run. We do know however that the step function invariably returns either `Nil` or a `Cons`, and the caller must be ready to handle both alternatives. We should exploit this static knowledge.

To statically (at code generation-time) make sure that the caller of the step function handles both alternatives of its result, we have to change the function to accept a pair of handlers: one for a `Nil` result and one for a `Cons`. In other words, we have to change the result's representation, from the sum `stream_shape` to a product of eliminators. Such a replacement effectively removes the need to construct the `stream_shape` data type at run-time in the first place. Essentially, we change `step` to be in continuation-passing style, i.e., to accept the continuation for its result. The `stream_shape` data type nominally remains, but it becomes the argument to the continuation and we mark its variants as statically known (with no need to construct it at run-time). All in all, we arrive at the following type for the staged stream

```
type α st_stream =
 ∃σ. σ code *
   (∀ω. σ code →
     ((α code,σ code) stream_shape → ω code) →
        ω code)
```

That is, a stream is again a pair of a hidden state, σ (only known dynamically, i.e., `σ code`), and a step function, but the step function does not return `stream_shape` values (of dynamic αs and σs) but accepts an extra argument (the continuation) to pass such values to. The step function returns whatever (generic type ω, only known dynamically) the continuation returns.

The variants of the `stream_shape` are now known when `step` calls its continuation, which happens at code-generation time. The `map` combinator becomes

```
let map : (α code → β code) →
          α st_stream → β st_stream =
  fun f (s,step) →
    let new_step s k = step s @@ function
    | Nil        → k Nil
    | Cons (a,t) → .⟨let a' = ∼(f a) in
                    ∼(k @@ Cons (.⟨a'⟩., t))⟩.
    in (s,new_step)
```

taking into account that `step`, instead of returning the result, calls a continuation on it. Although the data-type `stream_shape` remains, its construction and pattern-matching now happen at code-generation time, i.e., statically. As another example, the `fold` combinator becomes:

```
let fold : (ζ code → α code → ζ code) →
           ζ code → α st_stream → ζ code
 = fun f z (s,step) →
 .⟨let rec loop z s = ∼(step .⟨s⟩. @@ function
   | Nil        → .⟨z⟩.
   | Cons (a,t) → .⟨loop ∼(f .⟨z⟩. a) ∼t⟩.)
   in loop ∼z ∼s⟩.
```

Our running example pipeline, summing the squares of all elements of a sample array, now generates the following code

```
val c : int code = .⟨
  let rec loop_1 z_2 s_3 =
    match s_3 with
    | (i_4,arr_5) →
        if i_4 < (Array.length arr_5)
        then
          let el_6 = arr_5.(i_4) in
          let a'_7 = el_6 * el_6 in
          loop_1 (z_2 + a'_7) ((i_4 + 1), arr_5)
        else z_2 in
  loop_1 0 (0, [|0;1;2;3;4|])⟩.
```

In stark contrast with the naive staging of §4.2, the generated code has no traces of the `stream_shape` data type. Although the data type is still constructed and deconstructed, the corresponding overhead is shifted from the generated code to the code-generator. Generating code may take a bit longer but the result is more efficient. For full fusion, we will need to shift overhead to the generator two more times.

5.2 Fusing the Stream State

Although we have removed the most noticeable repeated construction and deconstruction of the `stream_shape` data type, the abstraction overhead still remains. The main loop in the generated code pattern-matches on the current state, which is the pair of the index and the array. The recursive invocation of the loop packs the index and the array back into a pair. Our task is to deforest the pair away. This seems rather difficult, however: the state is being updated on every iteration of the loop, and the loop structure (e.g., number of iterations) is generally not statically known. Although it is the (statically known) `step` function that computes the updated state, the state has to be threaded through the fold's `loop`, which treats it as a black-box piece of code. The fact it is a pair cannot be exploited and, hence, the overhead cannot be shifted to the generator. There is a way out, however. It requires a non-trivial step: The threading of the state through the loop can be eliminated if the state is mutable.

The step function no longer has to return (strictly speaking: pass to its continuation) the updated state: the update happens in place. Therefore, the state no longer has to be annotated as dynamic—its structure can be known to the generator. Finally, in order to have the appropriate operator allocate the reference cell for the array index, we need to employ the let-insertion technique [4], by also using continuation-passing style for the initial state. The definition of the stream type (`α st_stream`) now becomes:

```
type α st_stream =
 ∃σ.
   (∀ω. (σ → ω code) → ω code) *
   (∀ω. σ →
     ((α code,unit) stream_shape → ω code) →
        ω code)
```

That is, a stream is a pair of an `init` function and a `step` function. The `init` function implicitly hides a state: it knows how to call a continuation (that accepts a static state and returns a generic dynamic value, ω) and returns the result of the continuation. The `step` function is much like before, but operating on a statically-known state (or more correctly, a hidden state with a statically-known structure).

The new `of_arr` combinator demonstrates the let-insertion (the allocation of the reference cell for the current array index) in `init`, and the in-place update of the state (the `incr` operation):

```
let of_arr : α array code → α st_stream =
  let init arr k =
    .⟨let i = ref 0 and
          arr = ∼arr in ∼(k (.⟨i⟩.,.⟨arr⟩.))⟩.
  and step (i,arr) k =
   .⟨if !(∼i) < Array.length ∼arr
      then
       let el = (∼arr).(!(∼i)) in
       incr ∼i;
       ∼(k @@ Cons (.⟨el⟩., ()))
      else ∼(k Nil)⟩.
  in
  fun arr → (init arr,step)
```

Once again, until now the state of the `of_arr` stream had the type `(int * α array) code`. It has become `int ref code * α array code`, the statically known pair of two code values. The construction and deconstruction of that pair now happens at code-generation time.

The earlier `map` combinator did not even look at the current state (nor could it), therefore its code remains unaffected by the change in the state representation. The `fold` combinator no longer has to thread the state through its loop:

```
let fold : (ζ code → α code → ζ code) →
           ζ code → α st_stream → ζ code
 = fun f z (init,step) →
 init @@ fun s →
 .⟨let rec loop z = ∼(step s @@ function
   | Nil        → .⟨z⟩.
   | Cons (a,_) → .⟨loop ∼(f .⟨z⟩. a)⟩.)
   in loop ∼z⟩.
```

It obtains the state from the initializer and passes it to the step function, which knows its structure. The generated code for the running-example stream-processing pipeline is:

```
val c : int code = .⟨
  let i_8 = ref 0
  and arr_9 = [|0;1;2;3;4|] in
  let rec loop_10 z_11 =
    if ! i_8 < Array.length arr_9
    then
      let el_12 = arr_9.(! i_8) in
      incr i_8;
      let a'_13 = el_12 * el_12 in
      loop_10 (z_11+ a'_13)
    else z_11 in
  loop_10 0⟩.
```

The resulting code shows the absence of any overhead. All intermediate data structures have been eliminated. The code is what we could expect to get from a competent OCaml programmer.

5.3 Generating Imperative Loops

It seems we have achieved our goal. The library (extended for filtering, zipping, and nested streams) can be used in (Meta)OCaml practice. It relies, however, on tail-recursive function calls. These may be a good fit for OCaml,[6] but not for Java or Scala. (In Scala, tail-recursion is only supported with significant run-time overhead.) The fastest way to iterate is to use the native while-loops, especially in Java or Scala. Also, the dummy `(α code,unit) stream_shape` in the `α st_stream` type looks odd: the `stream_shape` data type has become artificial. Although `unit` has no effect on generated code, it is less than pleasing aesthetically to need a placeholder type in our signature. For these reasons, we embark on one last transformation.

The last step of stream staging is driven by several insights. First of all, most languages provide two sorts of imperative loops: a general while-loop and the more specific, and often more efficient (at least in OCaml) for-loops. We would like to be able to generate for-loops if possible, for instance, in our running example. However, with added subranging or zipping (described in detail in §6, below) the pipeline can no longer be represented as an OCaml for-loop, which cannot accommodate extra termination tests. Therefore, the stream producer should not commit to any particular loop representation. Rather, it has to collect all the needed information for loop generation, but leave the actual generation to the stream consumer, when the entire pipeline is known. Thus the stream representation type becomes as follows:

```
type (α,σ) producer_t =
  | For    of
     {upb:   σ → int code;
      index: σ → int code → (α → unit code) →
                 unit code}
  | Unfold of
     {term: σ → bool code;
      step: σ → (α → unit code) → unit code}
 and α st_stream =
   ∃σ. (∀ω. (σ → ω code) → ω code) *
             (α,σ) producer_t
 and α stream = α code st_stream
```

That is, a stream type is a pair of an `init` function (which, as before, has the ability to call a continuation with a hidden state) and an encoding of a producer. We distinguish two sorts of producers: a producer that can be driven by a for-loop or a general "unfold" producer. Each of them supports two functions. A for-loop producer carries the exact upper bound, `upb`, for the loop index variable and the `index` function that returns the stream element given an index. For a general producer, we refactor (with an eye for the while-loop) the earlier representation

```
((α code,unit) stream_shape → ω code) → ω code
```

into two components: the termination test, `term`, producing a dynamic `bool` value (if the test yields `false` for the current state, the loop is finished) and the `step` function, to produce a new stream element and advance the state. We also used another insight: the imperative-loop–style of the processing pipeline makes it unnecessary (moreover, difficult) to be passing around the consumer (`fold`) state from one iteration to another. It is easier to accumulate the state in a mutable cell. Therefore, the answer type of the `step` and `index` functions can be `unit code` rather than `ω code`.

There is one more difference from the earlier staged stream, which is a bit harder to see. Previously, the stream value was annotated as dynamic: we really cannot know before running the pipeline what the current element is. Now, the value produced by the `step` or `index` functions has the type α without any `code` annotations, meaning that it is statically known! Although the value of the current stream element is determined only when the pipeline is run, its structure can be known earlier. For example, the new type lets the producer yield a pair of values: even though the values themselves are annotated as dynamic (of a `code` type) the fact that it is a pair can be known statically. We use this extra flexibility of the more general stream value type extensively in §6.2.

We can now see the new design in action. The stream producer `of_arr` is surely the for-loop-style producer:

```
let of_arr : α array code → α stream = fun arr →
  let init k = .⟨let arr = ∼arr in ∼(k .⟨arr⟩.)⟩.
```

[6] Actually, our benchmarking reveals that for- and while-loops are currently faster even in OCaml.

```
  and upb arr = .⟨Array.length ~arr - 1⟩.
  and index arr i k =
      .⟨let el = (~arr).(~i) in ~(k .⟨el⟩.)⟩.
  in (init, For {upb;index})
```

In contrast, the unfold combinator

```
let unfold : (ζ code → (α * ζ) option code) →
             ζ code → α stream = ...
```

is an `Unfold` producer.

Importantly, a producer that starts as a for-loop may later be converted to a more general while-loop producer, (so as to tack on extra termination tests—see `take` in §6.2). Therefore, we need the conversion function

```
let for_unfold : α st_stream → α st_stream= function
 | (init,For {upb;index}) →
    let init k = init @@ fun s0 →
           .⟨let i = ref 0 in ~(k (.⟨i⟩.,s0))⟩.
    and term (i,s0)   = .⟨!(~i) ≤ ~(upb s0)⟩.
    and step (i,s0) k =
      index s0 .⟨!(~i)⟩. @@
           fun a → .⟨(incr ~i; ~(k a))⟩.
    in (init, Unfold {term;step})
 | x → x
```

used internally within the library.

The stream mapping operation composes the mapping function with the `index` or `step`: transforming, as before, the produced value "in-flight", so to speak.

```
let rec map_raw: (α → (β → unit code) → unit code)
                 → α st_stream → β st_stream =
 fun tr → function
 | (init,For ({index;_} as g)) →
     let index s i k = index s i @@ fun e → tr e k in
     (init, For {g with index})
 | (init,Unfold ({step;_} as g)) →
     let step s k = step s @@ fun e → tr e k in
     (init, Unfold {g with step})
```

We have defined `map_raw` with the general type (to be used later, e.g., in §6.2); the familiar `map` is a special case:

```
let map : (α code → β code) → α stream → β stream
  = fun f str → map_raw (fun a k →
      .⟨let t = ~(f a) in ~(k .⟨t⟩.)⟩.) str
```

The mapper `tr` in `map_raw` is in the continuation-passing style with the `unit code` answer-type. This allows us to perform let-insertion [4], binding the mapped value to a variable, and hence avoiding the potential duplication of the mapping operation.

As behooves pull-style streams, the consumer at the end of the pipeline generates the loop to drive the iteration. Yet we do manage to generate for-loops, characteristic of push-streams, see §3.

```
let rec fold_raw :
 (α → unit code) → α st_stream → unit code
 = fun consumer → function
   | (init,For {upb;index}) →
         init @@ fun sp →
           .⟨for i = 0 to ~(upb sp) do
              ~(index sp .⟨i⟩. @@ consumer)
             done⟩.
   | (init,Unfold {term;step}) →
         init @@ fun sp →
           .⟨while ~(term sp) do
              ~(step sp @@ consumer)
           done⟩.
```

It is simpler (especially when we add nesting later) to implement a more general `fold_raw`, which feeds the eventually produced stream element to the given imperative `consumer`. The ordinary `fold` is a wrapper that provides such a consumer, accumulating the result in a mutable cell and extracting it at the end.

```
let fold : (ζ code → α code → ζ code) →
           ζ code → α stream → ζ code
= fun f z str →
  .⟨let s = ref ~z in
    (~(fold_raw
         (fun a → .⟨s := ~(f .⟨!s⟩. a)⟩.)
         str);
    !s)⟩.
```

The generated code for our running example is:

```
val c : int code = .⟨
  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⟩.
```

This code could not be better. It is what we expect an OCaml programmer to write, and, furthermore, such code performs ultimately well in Scala, Java and other languages. We have achieved our goal—for simple pipelines, at least.

6. Full Library

The previous section presented our approach of eliminating all abstraction overhead of a stream library through the creative use of staging—generating code of hand-written quality and efficiency. However, a full stream library has more combinators than we have dealt with so far. This section describes the remaining facilities: filtering, sub-ranging, nested streams and parallel streams (zipping). Consistently achieving deforestation and high performance in the presence of all these features is a challenge. We identify three concepts of stream processing that drive our effort: the rate of production and consumption of stream elements (*linearity* and filtering—§6.1), size-limiting a stream (§6.2), and processing multiple streams in tandem (zipping—§6.3). We conclude our core discussion with a theorem of eliminating all overhead.

6.1 Filtered and Nested Streams

Our library is primarily based on the design presented at the end of §5. Filtering and nested streams (`flat_map`) require an extension, however, which lets us treat filtering and flat-mapping uniformly.

Let us look back at this design. It centers on two operations, `term` and `step`: forgetting for a moment the staging annotations, `term s` decides whether the stream still continues, while `step s` produces the current element and advances the state. Exactly one stream element is produced per advance in state. We call such streams *linear*. They have many useful algebraic properties, especially when it comes to zipping. We will exploit them in §6.3.

Clearly the `of_arr` stream producer and the more general `unfold` producers build linear streams. The `map` operation preserves the linearity. What destroys it is filtering and nesting. In the filtered stream `prod ▷ filter p`, the advancement of the `prod` state is no longer always accompanied by the production of the stream element: if the filter predicate `p` rejects the element, the pipeline will yield nothing for that iteration. Likewise, in the nested stream `prod ▷ flat_map (fun x → inner_prod x)`, the advancement of the `prod` state may lead to zero, one, or many stream elements given to the pipeline consumer.

Given the importance of linearity (to be seen in full in §6.3) we keep track of it in the stream representation. We represent a non-linear stream as a composition of an always-linear producer with a non-linear transformer:

```
type card_t = AtMost1 | Many

type (α,σ) producer_t =
  | For    of
     {upb:   σ → int code;
      index: σ → int code → (α → unit code) →
                   unit code}
  | Unfold of
     {term: σ → bool code;
      card: card_t;
      step: σ → (α → unit code) → unit code}
and α producer =
  ∃σ. (∀ω. (σ → ω code) → ω code) *
            (α,σ) producer_t
and α st_stream =
 | Linear of α producer
 | Nested of ∃β. β producer * (β → α st_stream)
and α stream = α code st_stream
```

The difference from the earlier representation in §5 is the addition of a sum data type with variants `Linear` and `Nested`, for linear and nested streams. We also added a cardinality marker to the general producer, noting if it generates possibly many elements or at most one.

The `flat_map` combinator adds a non-linear transformer to the stream (recursively descending into the already nested stream):

```
let rec flat_map_raw :
  (α → β st_stream) → α st_stream → β st_stream =
 fun tr → function
 | Linear prod         → Nested (prod,tr)
 | Nested (prod,nestf) →
    Nested (prod,fun a → flat_map_raw tr @@ nestf a)

let flat_map :
  (α code → β stream) → α stream → β stream =
  flat_map_raw
```

The `filter` combinator becomes just a particular case of flat-mapping: nesting of a stream that produces at most one element:

```
let filter : (α code → bool code) →
             α stream → α stream = fun f →
  let filter_stream a =
  ((fun k → k a),
   Unfold {card = AtMost1; term = f;
           step = fun a k → k a})
  in flat_map_raw (fun x → Linear (filter_stream x))
```

The addition of recursively `Nested` streams requires an adjustment of the earlier, §5, `map_raw` and `fold` definitions to recursively descend down the nesting. The adjustment is straightforward; please see the accompanying source code for details. The adjusted `fold` will generate nested loops for nested streams.

6.2 Sub-Ranging and Infinite Streams

The stream combinator `take` limits the size of the stream:

```
val take : int code → α stream → α stream
```

For example, `take .⟨10⟩. str` is a stream of the first 10 elements of `str`, if there are that many. It is the `take` combinator that lets us handle conceptually infinite streams. Such infinite streams are easily created with `unfold`: for example, `iota n`, the stream of all natural numbers from `n` up:

```
let iota n = unfold (fun n → .⟨Some (∼n,∼n+ 1)⟩.) n
```

The implementation of `take` demonstrates and justifies design decisions that might have seemed arbitrary earlier. For example, distinguishing linear streams and indexed, for-loop–style producers in the representation type pays off. In a linear stream pipeline, the number of elements at the end of the pipeline is the same as the number of produced elements. Therefore, for a linear stream, `take` can impose the limit close to the production. The for-loop-style producer is particularly easy to limit in size: we merely need to adjust the upper bound:

```
let take = fun n → function
 | Linear (init, For {upb;index}) →
    let upb s = .⟨min (∼n-1) ∼(upb s)⟩. in
    Linear (init, For {upb;index})
 ...
```

Limiting the size of a non-linear stream is slightly more complicated:

```
let take = fun n → function
 ...
 | Nested (p,nestf) →
    Nested (add_nr n (for_unfold p),
     fun (nr,a) →
      map_raw (fun a k → .⟨(decr ∼nr; ∼(k a))⟩.) @@
      more_termination .⟨! ∼nr > 0⟩. (nestf a))
```

The idea is straightforward: allocate a reference cell `nr` with the remaining element count (initially `n`), add the check `!nr > 0` to the termination condition of the stream producer, and arrange to decrement the `nr` count at the end of the stream. Recall, for a non-linear stream—a composition of several producers—the count of eventually produced elements may differ arbitrarily from the count of the elements emitted by the first producer. A moment of thought shows that the range check `!nr > 0` has to be added not only to the first producer but to the producers of all nested substreams: this is the role of function `more_termination` (see the accompanying code for its definition) in the fragment above. The operation `add_nr` allocates cell `nr` and adds the termination condition to the first producer. Recall that, since for-loops in OCaml cannot take extra termination conditions, a for-loop-style producer has to be first converted to a general unfold-style producer, using `for_unfold`, which we defined in §5. The operation `add_nr` (definition not shown) also adds `nr` to the produced value: The result of `add_nr n (for_unfold p)` is of type `(int ref code,α code) st_stream`. Adding the operation to decrement `nr` is conveniently done with `map_raw` from §5. We, thus, now see the use for the more general (α and not just `α code`) stream type and the general stream mapping function.

6.3 zip: Fusing Parallel Streams

This section describes the most complex operation: handling two streams in tandem, i.e., zipping:

```
val zip_with   : (α code → β code → γ code) →
                 (α stream → β stream → γ stream)
```

Many stream libraries lack this operation: first, because zipping is practically impossible with push streams, due to inherent complexity, as we shall see shortly. Linear streams and the general `map_raw` operation turn out to be important abstractions that make the problem tractable.

One cause of the complexity of `zip_with` is the need to consider many special cases, so as to generate code of hand-written quality. All cases share the operation of combining the elements of two streams to obtain the element of the zipped stream. It is convenient to factor out this operation:

```
val zip_raw: α st_stream → β st_stream →
             (α * β) st_stream

let zip_with f str1 str2 =
   map_raw (fun (x,y) k → k (f x y)) @@
   zip_raw str1 str2
```

The auxiliary `zip_raw` builds a stream of pairs—statically known pairs of dynamic values. Therefore, the overhead of constructing and deconstructing the pairs is incurred only once, in the generator. There is no tupling in the generated code.

The `zip_raw` function is a dispatcher for various special cases, to be explained below.

```
let rec zip_raw str1 str2 = match (str1,str2) with
  | (Linear prod1, Linear prod2) →
      Linear (zip_producer prod1 prod2)
  | (Linear prod1, Nested (prod2,nestf2)) →
      push_linear (for_unfold prod1)
                  (for_unfold prod2,nestf2)
  | (Nested (prod1,nestf1), Linear prod2) →
      map_raw (fun (y,x) k → k (x,y)) @@
      push_linear (for_unfold prod2)
                  (for_unfold prod1,nestf1)
  | (str1,str2) →
      zip_raw (Linear (make_linear str1)) str2
```

The simplest case is zipping two linear streams. Recall, a linear stream produces exactly one element when advancing the state. Zipped linear streams, hence, yield a linear stream that produces a pair of elements by advancing the state of both argument streams exactly once. The pairing of the stream advancement is especially efficient for for-loop–style streams, which share a common state, the index:

```
let rec zip_producer:
  α producer → β producer → (α * β) producer =
 fun p1 p2 → match (p1,p2) with
 | (i1,For f1), (i2,For f2) →
     let init k =
       i1.init @@ fun s1 →
       i2.init @@ fun s2 → k (s1,s2)
     and upb (s1,s2) = .⟨min ~(f1.upb s1)
                              ~(f2.upb s2)⟩.)
     and index fun (s1,s2) i k =
       f1.index s1 i @@ fun e1 →
       f2.index s2 i @@ fun e2 → k (e1,e2)
     in (init, For {upb;index})
  | (* elided *)
```

In the general case, `zip_raw str1 str2` has to determine how to advance the state of `str1` and `str2` to produce one element of the zipped stream: the pair of the current elements of `str1` and `str2`. Informally, we have to reason all the way from the production of an element to the advancement of the state. For linear streams, the relation between the current element and the state is one-to-one. In general, the state of the two components of the zipped stream advance at different paces. Consider the following sample streams:

```
let stre = of_arr arr1
           ▷ filter (fun x → .⟨~x mod 2 = 0⟩.)
let strq = of_arr arr2
           ▷ map (fun x → .⟨~x * ~x⟩.)
let str2 = of_arr arr1
           ▷ flat_map (fun _ → of_arr .⟨[|1;2]⟩.)
let str3 = of_arr arr1
           ▷ flat_map (fun _ → of_arr .⟨[|1;2;3]⟩.)
```

To produce one element of `zip_raw stre strq`, the state of `stre` has to be advanced a statically-unknown number of times. Zipping nested streams is even harder—e.g., `zip_raw str2 str3`, where the states advance in complex patterns and the end of the inner stream of `str2` does not align with the end of the inner stream in `str3`.

Zipping simplifies if one of the streams is linear, as in `zip_raw stre strq`. The key insight is to advance the linear stream `strq` after we are sure to have obtained the element of the non-linear stream `stre`. This idea is elegantly realized as mapping of the step function of `strq` over `stre` (the latter, is, recall, `int stream`, which is `int code st_stream`), obtaining the desired zipped `(int code, int code) st_stream`:

```
map_raw (fun e1 k →
         strq.step sq (fun e2 → k (e1,e2))) stre
```

The above code is an outline: we have to initialize `strq` to obtain its state `sq`, and we need to push the termination condition of `strq` into `stre`. Function `push_linear` in the accompanying code takes care of all these details.

The last and most complex case is zipping two non-linear streams. Our solution is to convert one of them to a linear stream, and then use the approach just described. Turning a non-linear stream to a producer involves "reifying" a stream: converting an `α stream` data type to essentially a `(unit → α option) code` function, which, when called, reports the new element or the end of the stream. We have to create a closure and generate and deconstruct the intermediate data type `α option`. There is no way around this: in one form or another, we have to capture the non-linear stream's continuation. The human programmer will have to do the same—this is precisely what makes zipping so difficult in practice. Our library reifies only one of the two zipped streams, without relying on tail-call optimization, for maximum portability.

6.4 Elimination of All Overhead, Formally

Sections 2, above, and 7, below, demonstrate the elimination of abstraction overhead on selected examples and benchmarks. We now state how and why the overhead is eliminated in all cases.

We call the higher-order arguments of `map`, `filter`, `zip_with`, etc. "user-generators": they are specified by the library user and provide per-element stream processing.

THEOREM 1. *Any well-typed pipeline generator—built by composing a stream producer, Fig.1, with an arbitrary combination of transformers followed by a reducer—terminates, provided the user-generators do. The resulting code—with the sole exception of pipelines zipping two flat-mapped streams—constructs no data structures beyond those constructed by the user-generators.*

Therefore, if the user generators proceed without construction/allocation, the entire pipeline, after the initial set-up, runs without allocations. The only exception is the zipping of two streams that are both made by flattening inner streams. In this case, the rate-adjusting allocation is inevitable, even in hand-written code, and is not considered overhead.

Proof sketch: The proof is simple, thanks to the explicitness of staging and treating the generated code as an opaque value that cannot be deconstructed and examined. Therefore, the only tuple construction operations in the generated code are those that we have explicitly generated. Hence, to prove our theorem, we only have to inspect the brackets that appear in our library implementation, checking for tuples or other objects.

7. Experiments

We evaluated our approach on several benchmarks from past literature, measuring the iteration throughput:

- **sum**: the simplest `of_arr arr ▷ sum` pipeline, summing the elements of an array;
- **sumOfSquares**: our running example from §4.2 on;
- **sumOfSquaresEven**: the sumOfSquares benchmark with added filter, summing the squares of only the even array elements;
- **cart**: $\sum x_i y_j$, using `flat_map` to build the outer-product stream;
- **maps**: consecutive map operations with integer multiplication;
- **filters**: consecutive filter operations using integer comparison;
- **dotProduct**: compute dot product of two arrays using `zip_with`;
- **flatMap_after_zipWith**: compute $\sum (x_i + x_i) y_j$, like cart above, doubling the `x` array via `zip_with (+)` with itself;
- **zipWith_after_flatMap**: `zip_with` of two streams one of which is the result of `flat_map`;
- **flat_map_take**: `flat_map` followed by `take`.

The source code of all benchmarks is available at the project's repository and the OCaml versions are also listed in Appendix D. Our benchmarks come from the sets by Murray et al. [21] and Coutts et al. [5], to which we added more complex combinations (the last three on the list above). (The Murray and Coutts sets also contain a few more simple operator combinations, which we omit for conciseness, as they share the performance characteristics of other benchmarks.)

The staged code was generated using our library (***strymonas***), with MetaOCaml on the OCaml platform and LMS on Scala, as detailed below. As one basis of comparison, we have implemented all benchmarks using the streams libraries available on each platform[7]: Batteries [8] in OCaml and the standard Java 8 and Scala streams. As there is not a unifying module that implements all the combinators we employ, we use data type conversions where possible. Java 8 does not support a `zip` operator, hence some benchmarks are missing for that setup.[9]

As the baseline and the other basis of comparison, we have hand-coded all the benchmarks, using high-performance, imperative code, with `while` or index-based `for`-loops, as applicable. In Scala we use only `while`-loops as they are the analogue of imperative iterations; `for`-loops in Scala operate over `Ranges` and have worse performance. In fact, in one case we had to re-code the hand-optimized loop upon discovering that it was not as optimal as we thought: the library-generated code significantly outperformed it!

Input: All tests were run with the same input set. For the **sum**, **sumOfSquares**, **sumOfSquaresEven**, **maps**, **filters** we used an array of $N = 100,000,000$ small integers: $x_i = i \bmod 10$. The **cart** test iterates over two arrays. An outer one of $10,000,000$ integers and an inner one of 10. For the **dotProduct** we used $10,000,000$ integers, for the **flatMap_after_zipWith** $10,000$, for the **zipWith_after_flatMap** $10,000,000$ and for the **flat_map_take** N numbers sub-sized by 20% of N.

Setup: The system we use runs an x64 OSX El Capitan 10.11.4 operating system on bare metal. It is equipped with a 2.7 GHz Intel Core i5 CPU (I5-5257U) having 2 physical and 2 logical cores. The total memory of the system is 8 GB of type 1867 MHz DDR3. We use version build 1.8.0_65-b17 of the Open JDK. The compiler versions of our setup are presented in the table below:

Language	Compiler	Staging
Java	Java 8 (1.8.0_65)	—
Scala	2.11.2	LMS 0.9.0
OCaml	4.02.1	BER MetaOCaml N102

Automation: For Java and Scala benchmarks we used the Java Microbenchmark Harness (JMH) [29] 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 30 warm-up iterations and 30 proper iterations. We also force garbage collection before benchmark execution and between runs. All OCaml code was compiled with `ocamlopt` into machine code. In particular, the MetaOCaml-generated code was saved into a file, compiled, and then benchmarked in isolation. The test harness invokes the compiled executable via `Sys.command`, which is not included in the results. The harness calculates the average execution time, computing the mean error and standard deviation using the Student-T distribution. The same method is employed in JMH. For all tests, we do not measure the time needed to initialize data-structures (filling arrays), nor the run-time compilation cost of staging. These costs are constant (i.e., they become proportionally insignificant for larger inputs or more iterations) and they were small, between 5 and 10ms, for all our runs.

Results: In Figures 2 and 3 we present the results of our experiments divided into two categories: a) the OCaml microbenchmarks of baseline, staged and batteries experiments and b) the JVM microbenchmarks. The JVM diagram contains the baselines for both Java and Scala. Shorter bars are better. Recall that all "baseline" implementations are carefully hand-optimized code.

As can be seen, our staged library achieves extremely high performance, matching hand-written code (in either OCaml, Java, or Scala) and outperforming other library options by orders of magnitude. Notably, the highly-optimized Java 8 streams are more than 10x slower for perfectly realistic benchmarks, when those do not conform to the optimal pattern (linear loop) of push streams.

8. Related Work

The literature on stream library designs is rich. Our approach is the first to offer full generality while eliminating processing overhead. We discuss individual related work in more detail next.

One of the earliest stream libraries that rely on staging is Common Lisp's SERIES [36, 37], which extensively relies on Lisp macros to interpret a subset of Lisp code as a stream EDSL. It builds a data flow graph and then compiles it into a single loop. It can handle filtering, multiple producers and consumers, but not nested streams. The (over)reliance on macros may lead to surprises since the programmer might not be aware that what looks like CL code is actually a DSL, with a slightly different semantics and syntax. An experimental Pipes package [15] attempts to re-implement and extend SERIES, using, this time, a proper EDSL. Pipes extends SERIES by allowing nesting, but restricts zipping to simple cases. It was posited that "arbitrary outputs per input, multiple consumers, multiple producers: choose two" [15]. Pipes "almost manages" (according to its author) to implement all three features. Our library demonstrates the conjecture is false by supporting all three facilities in full generality and with high performance.

Lippmeier et al. [18] present a line of work based on SERIES. They aim to transform first-order, non-recursive, synchronous, finite data-flow programs into fused pipelines. They derive inspiration from traditional data-flow languages like Lustre [10] and Lucid Synchrone [24]. In contrast, our library supports a greater range of fusible combinators, but for bulk data processing.

Haskell has lazy lists, which seem to offer incremental processing by design. Lazy lists cannot express pipelines that require side-effects such as reading or writing files.[10] The all-too-common memory leaks point out that lazy lists do not offer, again by design, stream fusion. Overcoming the drawbacks of lazy lists, coroutine-like iteratees [16] and many of their reimplementations support incremental processing even in the presence of effects, for nested streams and for several consumers and producers. Although iteratees avoid intermediate streams they still suffer large overheads for captured continuations, closures, and coroutine calls.

[7] We restrict our attention to the closest feature-rich apples-to-apples comparables: the industry-standard libraries for OCaml+JVM languages. We also report qualitative comparisons in §8.

[8] Batteries is the widely used "extended standard" library in OCaml `http://batteries.forge.ocamlcore.org/`.

[9] One could emulate `zip` using `iterator` from Java 8 push-streams—at significant drop in performance. This encoding also markedly differs from the structure of our other stream implementations.

[10] We disregard the lazy IO misfeature [16].

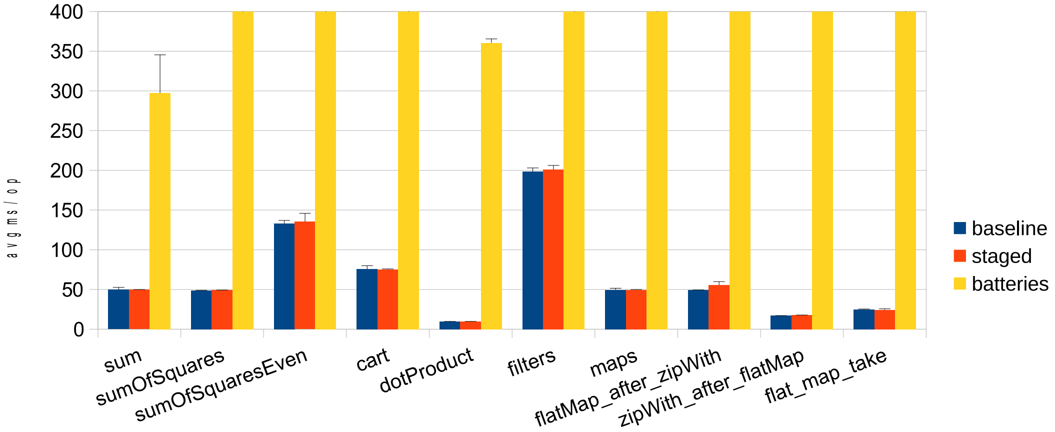

Figure 2: OCaml microbenchmarks in msec / iteration (avg. of 30, with mean-error bars shown). “Staged” is our library (***strymonas***). The figure is truncated: OCaml batteries take more than 60sec (per iteration!) for some complex benchmarks.

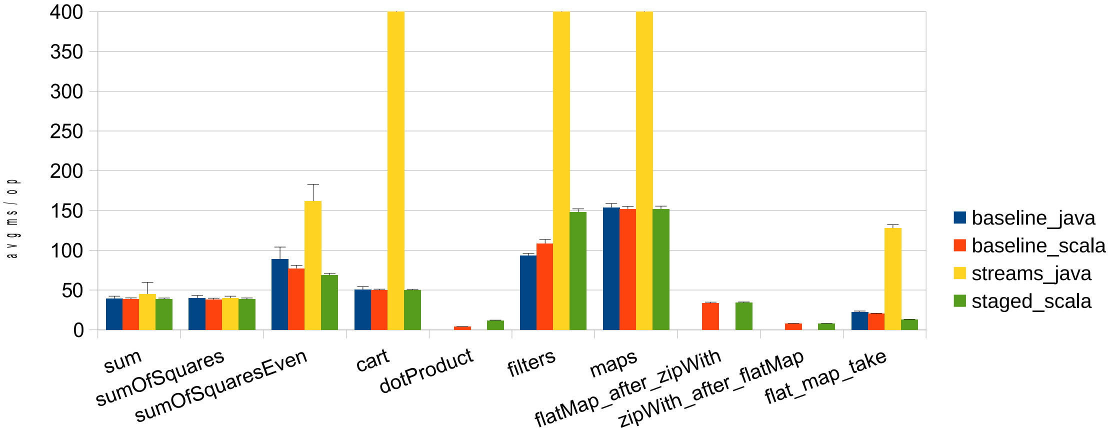

Figure 3: JVM microbenchmarks (both Java and Scala) in msec / iteration (avg. of 30, with mean-error bars shown). “Staged_scala” is our library (***strymonas***). The figure is truncated.

Coutts et al. [5] proposed *Stream Fusion* (the approach that has become associated with this fixed term), building on previous work (`build`/`foldr` [9] and `destroy`/`unfoldr` [32]) by fusing maps, filters, folds, zips and nested lists. The approach relies on the rewrite GHC RULES. Its notable contribution is the support for stream filtering. In that approach there is no specific treatment of linearity. The Coutts et al. stream fusion supports zipping, but only in simple cases (no zipping of nested, subranged streams). Finally, the Coutts et al. approach does not fully fuse pipelines that contain nested streams (`concatMap`). The reason is that the stream created by the transformation of `concatMap` uses an internal function that cannot by optimized by GHC by employing simple case reduction. The problem is presented very concisely by Farmer et al. in the *Hermit in the Stream* work [6].

The application of HERMIT [6] to streams [7] fixes the shortcomings of the Coutts et al. Stream Fusion [5] for `concatMap`. As the authors and Coutts say, `concatMap` is complicated because its mapping function may create any stream whose size is not statically known. The authors implement Coutts’s idea of transforming `concatMap` to `flatten`; the latter supports fusion for a constant inner stream. Using HERMIT instead of GHC RULES, Farmer et al. present as motivating examples two cases. Our approach handles the *non-constant inner stream case* without any additional action.

The second case is about *multiple inner streams* (of the same state type). Farmer et al. eliminate some overhead yet do not produce fully fused code. E.g., pipelines such as the following (in Haskell) are not fully fused:

```
concatMapS (\x → case even x of
   True → enumFromToS 1 x
   False → enumFromToS 1 (x + 1))
```

(Farmer et al. raise the question of how often such cases arise in a real program.) Our library internally places no restrictions on inner streams; it may well be that the flat-mapping function produces streams of different structure for each element of the outer stream. On the other hand, the `flat_map` interface only supports nested streams of a fixed structure—hence with the applicative rather than monadic interface. We can provide a more general `flat_map` with the continuation-passing interface for the mapping function, which then implements:

```
flat_map_cps (fun x k →
  .⟨if (even ~x) then ~(k (enumFromToS ...))
                 else ~(k (enumFromToS ...))⟩.)
```

We have refrained from offering this more general interface since there does not seem to be a practical need.

GHC RULES [23], extensively used in Stream Fusion, are applied to typed code but by themselves are not typed and are not guaranteed type-preserving. To write GHC rules, one has to have a very good understanding of GHC optimization passes, to ensure that the RULE matches and has any effect at all. RULES by themselves offer no guarantee, even the guarantee that the re-written code is well-typed. Multi-stage programming ensures that all staging transformations are type-correct.

Jonnalagedda et al. present a library using only CPS encodings (fold-based) [12]. It uses the Gill et al. foldr/build technique [9] to get staged streams in Scala. Like foldr/build, it does not support combinators with multiple inputs such as `zip`.

In our work, we employ the traditional MSP programming model to implement a performant streaming library. Rompf et al. [27] demonstrate a loop fusion and deforestation algorithm for data parallel loops and traversals. They use staging as a compiler transformation pass and apply to query processing for in-memory objects. That technique lacks the rich range of fused combinators over finite or infinite sources that we support, but seems adequate for the case studies presented in that work. Porting our technique from the staged-library level to the compiler-transformation level may be applicable in the context of Scala/LMS.

Generalized Stream Fusion [19] puts forward the idea of *bundled* stream representations. Each representation is designed to fit a particular stream consumer following the documented cost model. Although this design does not present a concrete range of optimizations to fuse combinators and generate loop-based code directly, it presents a generalized model that can "host" any number of specialized stream representations. Conceptually, this framework could be used to implement our optimizations. However, it relies on the black-box GHC optimizer—which is the opposite of our approach of full transparency and portability.

Ziria [31], a language for wireless systems' programming, compiles high-level reconfigurable data-flow programs to vectorized, fused C-code. Ziria's `tick` and `process` (pull and push respectively) demonstrate the benefits of having both processing styles in the same library. It would be interesting to combine our general-purpose stream library with Ziria's generation of vectorized C code.

Svensson et al.[33] unify pull- and push- arrays into a single library by defunctionalizing push arrays, concisely explaining why pull and push must co-exist under a unified library. They use a compile monad to interpret their embedded language into an imperative target one. In our work we get that for free from staging. Similarly, the representation of arrays in memory, with their `CMMem` data type, corresponds to staged arrays (of type α `array code`) in our work. The library they derive from the defunctionalization of `Push` streams is called `PushT` and the authors provide evidence that indexing a push array can, indeed, be efficient (as opposed to simple push-based streams). The paper does not seem to handle more challenging combinators like `concatMap` and `take` and does not efficiently handle the combinations of infinite and finite sources. Still, we share the same goal: to unify both styles of streams under one roof. Finally, Svensson et al. target arrays for embedded languages, while we target arrays natively in the language. Fusion is achieved by our library without relying on a compiler to intelligently handle all corner cases.

9. Discussion: Why Staging?

Our approach relies on staging. This may impose a barrier to the practical use of the library: staging annotations are unfamiliar to many programmers. Furthermore, it is natural to ask whether our approach could be implemented as a compiler optimization pass.

Complexity of staging. How much burden staging really imposes on a programmer is an empirical question. As our library becomes known and more-used we hope to collect data to answer this. In the meantime, we note that staging can be effectively hidden in code combinators. The first code example of §2 (summing the squares of elements of an array) can be written without the use of staging annotations as:

```
let sum = fold (fun z a → add a z) zero

of_arr arr
  ▷ map (fun x → mul x x)
  ▷ sum
```

In this form, the functions that handle stream elements are written using a small combinator library, with operations `add`, `mul`, etc. that hide all staging. The operations are defined simply as

```
let add x y = .⟨~x + ~y⟩. and mul x y = .⟨~x * ~y⟩.
let zero = .⟨0⟩.
```

Furthermore, our Scala implementation has no explicit staging annotations, only `Rep` types (which are arguably less intrusive). For instance, a simple pipeline is shown below:

```
def test (xs : Rep[Array[Int]]) : Rep[Int] =
  Stream[Int](xs).filter(d ⇒ d % 2 == 0).sum
```

Staging vs. compiler optimization. Our approach can certainly be cast as an optimization pass. The current staging formulation is an excellent blueprint for such a compiler rewrite. However, staging is both less intrusive and more disciplined—with high-level type safety guarantees—than changing the compiler. Furthermore, optimization is guaranteed only with full control of the compiler. Such control is possible in a domain-specific language, but not in a general-purpose language, such as the ones we target. Relying on a general-purpose compiler for library optimization is slippery. Although compiler analyses and transformations are (usually) sound, they are almost never complete: a compiler generally offers no guarantee that any optimization will be successfully applied.[11] There are several instances when an innocuous change to a program makes it much slower. The compiler is a black box, with the programmer forced into constantly reorganizing the program in unintuitive ways in order to achieve the desired performance.

10. Conclusions

We have presented the principles and the design of stream libraries that support the widest set of operations from past libraries and also permit elimination of the entire abstraction overhead. The design has been implemented as the ***strymonas*** library, for OCaml and for Scala/JVM. As confirmed experimentally, our library indeed offers the highest, guaranteed, and portable performance. Underlying the library is a representation of streams that captures the essence of iteration in streaming pipelines. It recognizes which operators drive the iteration, which contribute to filtering conditions, whether parts of the stream have linearity properties, and more. This decomposition of the essence of stream iteration is what allows us to perform very aggressive optimization, via staging, regardless of the streaming pipeline configuration.

Acknowledgments

We thank the anonymous reviewers of both the program committee and the artifact evaluation committee for their constructive comments. We gratefully acknowledge funding by the European Research Council under grant 307334 (SPADE).

[11] A recent quote by Ben Lippmeier, discussing RePa [13] on Haskell-Cafe, captures well the frustrations of advanced library writers: "The compilation method [...] depends on the GHC simplifier acting in a certain way—yet there is no specification of exactly what the simplifier should do, and no easy way to check that it did what was expected other than eyeballing the intermediate code. We really need a different approach to program optimisation [...] The [current approach] is fine for general purpose code optimisation but not 'compile by transformation' where we really depend on the transformations doing what they're supposed to."—`http://mail.haskell.org/pipermail/haskell-cafe/2016-July/124324.html`

References

[1] Reactive extensions, 2016. URL https://github.com/Reactive-Extensions.

[2] A. Biboudis, N. Palladinos, and Y. Smaragdakis. Clash of the Lambdas. *arXiv preprint arXiv:1406.6631*, 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, 2014. URL http://arxiv.org/abs/1406.6631.

[3] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams a la carte: Extensible Pipelines with Object Algebras. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37, pages 591–613, 2015. ISBN 978-3-939897-86-6.

[4] A. Bondorf. Improving binding times without explicit CPS-conversion. In *Lisp & Functional Programming*, pages 1–10, 1992.

[5] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL http://doi.acm.org/10.1145/1291151.1291199.

[6] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 1–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364508. URL http://doi.acm.org/10.1145/2364506.2364508.

[7] A. Farmer, C. Hoener zu Siederdissen, and A. Gill. The HERMIT in the Stream: Fusing Stream Fusion's concatMap. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 97–108, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543736. URL http://doi.acm.org/10.1145/2543728.2543736.

[8] J. Gibbons and G. Jones. The under-appreciated unfold. In *ICFP '98: Proceedings of the ACM International Conference on Functional Programming*, volume 34(1), pages 273–279, New York, Sept. 1998. ACM Press.

[9] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165214. URL http://doi.acm.org/10.1145/165180.165214.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[11] J. Inoue and W. Taha. Reasoning about multi-stage programs. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2012. URL http://dx.doi.org/10.1007/978-3-642-28869-2.

[12] M. Jonnalagedda and S. Stucki. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, SCALA 2015, pages 41–50, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3626-0. doi: 10.1145/2774975.2774981. URL http://doi.acm.org/10.1145/2774975.2774981.

[13] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863582. URL http://doi.acm.org/10.1145/1863543.1863582.

[14] R. Kelsey and P. Hudak. Realistic compilation by program transformation (detailed summary). In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 281–292, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75302. URL http://doi.acm.org/10.1145/75277.75302.

[15] P. Khuong. Introducing pipes, a lightweight stream fusion edsl, 2011. URL http://pvk.ca/Blog/Lisp/Pipes/.

[16] O. Kiselyov. Iteratees. In *FLOPS*, volume 7294 of *LNCS*, pages 166–181. Springer, 2012.

[17] O. Kiselyov. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, pages 86–102. Springer, 2014. URL http://link.springer.com/chapter/10.1007/978-3-319-07151-0_6.

[18] B. Lippmeier, M. M. Chakravarty, G. Keller, and A. Robinson. Data flow fusion with series expressions in Haskell. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 93–104, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2383-3. doi: 10.1145/2503778.2503782. URL http://doi.acm.org/10.1145/2503778.2503782.

[19] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 37–48, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500601. URL http://doi.acm.org/10.1145/2500365.2500601.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture: 5th Conference*, number 523 in Lecture Notes in Computer Science, pages 124–144, Berlin, 1991. The Association for Computing Machinery, Springer. URL http://research.microsoft.com/~emeijer/Papers/fpca91.pdfhttp://wwwhome.cs.utwente.nl/~fokkinga/mmf91m.pshttp://www.cse.ogi.edu/~erik/Personal/classic.htm#bananas.

[21] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Notices*, volume 46, pages 121–131. ACM, 2011. URL http://dl.acm.org/citation.cfm?id=1993513.

[22] N. Palladinos and K. Rontogiannis. LinqOptimizer: An automatic query optimizer for LINQ to Objects and PLINQ. Technical report, Nessos Information Technologies S.A., 2013. URL http://nessos.github.io/LinqOptimizer/.

[23] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, volume 1, pages 203–233, 2001. URL https://www.haskell.org/haskell-symposium/2001/2001-62.pdf#page=209.

[24] M. Pouzet. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006.

[25] A. Prokopec and D. Petrashko. ScalaBlitz: Lightning-fast Scala collections framework. Technical report, LAMP Scala Team, EPFL, 2013. URL http://scala-blitz.github.io/.

[26] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, June 2012. ISSN 0001-0782. doi: 10.1145/2184319.2184345. URL http://doi.acm.org/10.1145/2184319.2184345.

[27] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 497–510, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429128. URL http://doi.acm.org/10.1145/2429069.2429128.

[28] M. Shaw, W. A. Wulf, and R. L. London. Abstraction and verification in Alphard: defining and specifying iteration and generators. *Communications of the ACM*, 20(8):553–564, 1977.

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

[30] M. H. B. Sørensen, R. Glück, and N. D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized

partial computation. In D. Sannella, editor, *Programming Languages and Systems: Proceedings of ESOP'94, 5th European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Berlin, 11–13 Apr. 1994. Springer. URL ftp://ftp.diku.dk/diku/semantics/papers/D-190.ps.gz.

[31] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agullo. Ziria: A DSL for wireless systems programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 415–428, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694368. URL http://doi.acm.org/10.1145/2694344.2694368.

[32] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581491. URL http://doi.acm.org/10.1145/581478.581491.

[33] B. J. Svensson and J. Svenningsson. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 43–52, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3040-4. doi: 10.1145/2636228.2636231. URL http://doi.acm.org/10.1145/2636228.2636231.

[34] W. Taha. A Gentle Introduction to Multi-stage Programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, pages 30–50. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22119-7 978-3-540-25935-0. URL http://link.springer.com/chapter/10.1007/978-3-540-25935-0_3.

[35] P. L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. URL http://homepages.inf.ed.ac.uk/wadler/topics/deforestation.html.

[36] R. C. Waters. User manual for the series macro package. MIT AI Memo 1082, 1989. URL ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-1082.pdf.

[37] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. Syst.*, 13(1):52–98, Jan. 1991. ISSN 0164-0925. doi: 10.1145/114005.102806. URL http://doi.acm.org/10.1145/114005.102806.

A. Generated code for the Complex example

We show the generated code for the last example of Section §2, repeated below for reference:

```
(* Zipping function *)
zip_with (fun e1 e2 → .⟨(∼e1,∼e2)⟩.)
 (* First stream to zip *)
 (of_arr .⟨arr1⟩.
   ▷ map (fun x → .⟨∼x * ∼x⟩.)
   ▷ take .⟨12⟩.
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨∼x * ∼x⟩.))
 (* Second stream to zip *)
 (iota .⟨1⟩.
   ▷ flat_map (fun x → iota .⟨∼x+ 1⟩. ▷ take .⟨3⟩.)
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.))
 ▷ fold (fun z a → .⟨∼a :: ∼z⟩.) .⟨[]⟩.
```

The generated code is:

```
let s_23 = ref [] in
let arr_24 = arr1 in
 let i_25 = ref 0 in
 let curr_26 = ref None in
 let nadv_27 = ref None in
 let adv_32 () =
   curr_26 := None;
   while
     ((! curr_26) = None) &&
       ((! nadv_27 ≠ None) ||
          (! i_25 ≤ (min (12 - 1) (Array.length arr_24 - 1))))
     do
     match ! nadv_27 with
      | Some adv_28 → adv_28 ()
      | None  →
          let el_29 = arr_24.(! i_25) in
          let t_30 = el_29 * el_29 in
          incr i_25;
          if (t_30 mod 2) = 0
          then let t_31 = t_30 * t_30 in
               curr_26 := Some t_31
     done in
 adv_32 ();
 let s_33 = ref (Some (1, (1 + 1))) in
 let term1r_34 = ref (! curr_26 ≠ None) in
 while ! term1r_34 && ! s_33 ≠ None do
    match ! s_33 with
    | Some (el_35,s'_36) →
        s_33 := (Some (s'_36, (s'_36 + 1)));
        let s_37 =
           ref (Some (el_35 + 1, (el_35 + 1) + 1)) in
        let nr_38 = ref 3 in
        while (! term1r_34) &&
              (((! nr_38) > 0) && ((! s_37) ≠ None)) do
          match ! s_37 with
            | Some (el_39,s'_40) →
                 s_37 := Some (s'_40, (s'_40 + 1));
                 decr nr_38;
                 if el_39 mod 2 = 0
                 then
                   (match ! curr_26 with
                    | Some el_41 →
                        adv_32 ();
                        term1r_34 := !curr_26 ≠ None;
                        s_23 := (el_41, el_39) :: ! s_23)
          done
    done;
! s_23
```

B. Cartesian Product

```
let cart = fun (arr1, arr2) →
  ofArr arr1
  ▷ flat_map (fun x →
      ofArr arr2 ▷ map (fun y → .⟨ ∼x * ∼y⟩.))
  ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;
```

C. Generated code for Cartesian Product

```
let x = Array.init 1000 (fun i_1  → i_1) in
let y = Array.init 10   (fun i_2  → i_2) in
let arr_1 = x in
let size_1 = Array.length arr_1 in
let iarr_1 = ref 0 in
let rec loop_1 acc_1 =
  if (!iarr_1) ≥ size_1
  then acc_1
  else
    (let el_1 = arr_1.(!iarr_1) in
     incr iarr_117;
     (let acc1_tmp =
        let arr_2 = y in
        let size_2 = Array.length arr_2 in
        let iarr_2 = ref 0 in
        let rec loop_2 acc_2 =
          if (!iarr_2) ≥ size_2
          then acc_2
          else
            (let el_2 = arr_2.(!iarr_2) in
             incr iarr_2;
             (let acc2_tmp =
                acc_2 + (el_1  * el_2) in
              loop_2 acc2_tmp)) in
        loop_2 acc_1 in
      loop_1 acc1_tmp)) in
loop_1 0
```

D. Streams and baseline benchmarks

```
let sumS
= fun arr →
   of_arr arr
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let sumShand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
      sum := !sum + (∼arr1).(counter1);
   done;
   !sum ⟩.;;

let sumOfSquaresS
= fun arr →
   of_arr arr
   ▷ map (fun x → .⟨∼x * ∼x⟩.)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let sumOfSquaresShand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
    let item1 = (∼arr1).(counter1) in
    sum := !sum + item1*item1;
   done;
   !sum⟩.;;

let mapsS
= fun arr →
   of_arr arr
   ▷ map (fun x → .⟨∼x * 1⟩.)
   ▷ map (fun x → .⟨∼x * 2⟩.)
   ▷ map (fun x → .⟨∼x * 3⟩.)
   ▷ map (fun x → .⟨∼x * 4⟩.)
   ▷ map (fun x → .⟨∼x * 5⟩.)
   ▷ map (fun x → .⟨∼x * 6⟩.)
   ▷ map (fun x → .⟨∼x * 7⟩.)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let maps_hand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
   let item1 = (∼arr1).(counter1) in
    sum := !sum + item1*1*2*3*4*5*6*7;
   done;
   !sum⟩.;;

let filtersS
= fun arr →
   of_arr arr
   ▷ filter (fun x → .⟨∼x > 1⟩.)
   ▷ filter (fun x → .⟨∼x > 2⟩.)
   ▷ filter (fun x → .⟨∼x > 3⟩.)
   ▷ filter (fun x → .⟨∼x > 4⟩.)
   ▷ filter (fun x → .⟨∼x > 5⟩.)
   ▷ filter (fun x → .⟨∼x > 6⟩.)
   ▷ filter (fun x → .⟨∼x > 7⟩.)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let filters_hand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
      let item1 = (∼arr1).(counter1) in
      if (item1 > 1 && item1 > 2 && item1 > 3 &&
        item1 > 4 && item1 > 5 && item1 > 6 &&
        item1 > 7) then
      begin
      sum := !sum + item1;
      end;
   done;
   !sum⟩.;;

let sumOfSquaresEvenS
= fun arr →
   of_arr arr
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨∼x * ∼x⟩.)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let sumOfSquaresEvenShand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
   let item1 = (∼arr1).(counter1) in
   if item1 mod 2 = 0 then
   begin
     sum := !sum + item1*item1
   end;
   done;
   !sum⟩.;;

let cartS
= fun (arr1, arr2) →
   of_arr arr1
   ▷ flat_map (fun x →
      of_arr arr2 ▷ map (fun y → .⟨ ∼x * ∼y⟩.))
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let cartShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
      let item1 = (∼arr1).(counter1) in
      for counter2 = 0 to Array.length ∼arr2 - 1 do
         let item2 = (∼arr2).(counter2) in
         sum := !sum + item1 * item2;
      done;
   done;
   !sum ⟩.;;

let dotProductS
= fun (arr1, arr2) →
   zip_with (fun e1 e2 → .⟨∼e1 * ∼e2⟩.)
            (of_arr arr1) (of_arr arr2)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let dotProductShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   for counter = 0 to
         min (Array.length ∼arr1)
             (Array.length ∼arr2) - 1 do
      let item1 = (∼arr1).(counter) in
      let item2 = (∼arr2).(counter) in
      sum := !sum + item1 * item2;
   done;
  !sum⟩.;;

let flatMap_after_zipWithS
= fun (arr1, arr2) →
   zip_with (fun e1 e2 → .⟨∼e1 + ∼e2⟩.)
            (of_arr arr1) (of_arr arr1)
   ▷ flat_map (fun x → of_arr arr2
          ▷ map (fun el → .⟨∼el + ∼x⟩.))
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let flatMap_after_zipWithShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 - 1 do
     let x = (∼arr1).(counter1)
             + (∼arr1).(counter1) in
     for counter2 = 0 to Array.length ∼arr2 - 1 do
     let item2 = (∼arr2).(counter2) in
       sum := !sum + item2 + x;
     done;
   done;
   !sum⟩.;;

let zipWith_after_flatMapS
= fun (arr1, arr2) →
   of_arr arr1
   ▷ flat_map (fun x →
      of_arr arr2 ▷ map (fun y → .⟨∼y + ∼x⟩.))
   ▷ zip_with (fun e1 e2 → .⟨∼e1 + ∼e2⟩.)
               (of_arr arr1)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let zipWith_after_flatMapShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   let i1 = ref 0 in
   let i2 = ref 0 in
   let flag1 = ref ((!i1) ≤ ((Array.length ∼arr1) - 1)) in
   while (!flag1) && ((!i2) ≤ ((Array.length ∼arr2) - 1)) do
      let el2 = (∼arr2).(!i2) in
      incr i2;
      (let i_zip = ref 0 in
       while (!flag1) &&
             ((!i_zip) ≤ ((Array.length ∼arr1) - 1)) do
         let el1 = (∼arr1).(!i_zip) in
         incr i_zip;
         let elz = (∼arr1).(!i1) in
         incr i1;
         flag1 := ((!i1) ≤ ((Array.length ∼arr1) - 1));
         sum := ((!sum) + (elz + el1 + el2))
         done)
      done;
   !sum⟩.;;
```

```
let flat_map_takeS
= fun (arr1, arr2) →
   of_arr arr1
   ▷ flat_map (fun x → of_arr arr2
      ▷ map (fun y → .⟨ ∼x * ∼y⟩.))
   ▷ take .⟨20000000⟩.
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let flat_map_takeShand
= fun (arr1, arr2) → .⟨
   let counter1 = ref 0 in
   let counter2 = ref 0 in
   let sum = ref 0 in
   let n = ref 0 in
   let flag = ref true in
   let size1 = Array.length ∼arr1 in
   let size2 = Array.length ∼arr2 in
   while !counter1 < size1 && !flag do
      let item1 = (∼arr1).(!counter1) in
      while !counter2 < size2 && !flag do
         let item2 = (∼arr2).(!counter2) in
         sum := !sum + item1 * item2;
         counter2 := !counter2 + 1;
         n := !n + 1;
         if !n = 20000000 then
         flag := false
      done;
      counter2 := 0;
      counter1 := !counter1 + 1;
   done;
   !sum ⟩.;;
```