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 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 \textit{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 \(^4\), a form of metaprogramming, to achieve guaranteed stream fusion. This is in contrast to past use of source-to-source transformations of functional languages \(^4\), of AST run-time rewriting \(^2\)\(^1\)\(^2\), and compile-time macros \(^2\)\(^5\) or Haskell GHC \textsc{RULES} \(^5\)\(^2\) to express domain-specific stream 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 §7.

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 \(^1\) in diverse settings. The first is an OCaml library, staged with BEK MetaOCaml \(^1\). The second is a Scala library (also usable by client code in Java and other JVM languages), staged with Lightweight Modular Staging (LMS) \(^2\)\(^5\).

We evaluate strymonas on a suite of benchmarks \(^7\), comparing with handwritten 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 handwritten 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 \(\langle e \rangle\), and \(\sim e\), and the code type \(\langle 17 \rangle\)\(^3\)\(^4\). 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.

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 \(α\)s.

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

\[
\begin{align*}
\text{let sum = fold (fun z a → (z * z)) .(0).} \\
\text{of_arr .(arr).} \\
\text{map (fun x → (x * x))} \\
\text{sum}
\end{align*}
\]

is not far from the mathematical notation. Here, \(\triangleright\), 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:

\[
\begin{align*}
\text{let s_1 = ref 0 in} \\
\text{let arr_2 = arr in} \\
\text{for i_3 = 0 to Array.length arr_2 -1 do} \\
\text{let el_4 = arr_2.(i_3) in} \\
\text{let t_5 = el_4 * el_4 in} \\
\text{s_1 := t_5 + !s_1; done;}
\end{align*}
\]

\(1\) 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:

\[
\text{let ex = of_arr \:<\:\text{arr}. \:map (fun x \rightarrow \langle x, x \rangle).}
\]
\[
\qquad \text{ex \:filter (fun x \rightarrow \langle x, x \rangle mod 17 > 7.) \:sum}
\]

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:

\[
\text{let ex = of_arr \:<\:\text{arr}. \:map (fun x \rightarrow \langle x, x \rangle).}
\]
\[
\qquad \text{ex \:filter (fun x \rightarrow \langle x, x \rangle mod 17 > 7.) \:sum}
\]

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

\[
\text{let s_1 = ref 0 in}
\]
\[
\text{let arr_3 = ref in}
\]
\[
\text{let i_3 = ref 0 in}
\]
\[
\text{let arr_4 = ref n in}
\]
\[
\text{while \:nr_4 \:> \:0 \&\& \:i_3 \:<\:\text{Array.length arr_2 - 1 do}
\]
\[
\text{let el_5 = arr_2.(! i_3) in}
\]
\[
\text{let t_6 = el_5 * el_5 in}
\]
\[
\text{incr i_3;}
\]
\[
\text{let el_21 = arr_18.(i_20) in}
\]
\[
\text{let el_22 = arr_19.(i_20) in}
\]
\[
\text{let el_2 = arr_2.(i_20) in}
\]
\[
\text{let el_3 = arr_1.(i_20) in}
\]
\[
\text{let s_17 = ref 0 in}
\]
\[
\text{let s_18 = ref in}
\]
\[
\text{let arr_18 = ref arr in}
\]
\[
\text{let arr_19 = ref arr2 in}
\]
\[
\text{for i_20 = 0 to \:min (Array.length arr_18 -1)}
\]
\[
\text{let el_21 = arr_18.(i_20) in}
\]
\[
\text{let el_22 = arr_19.(i_20) in}
\]
\[
\text{s_17 := el_21 * el_22 + !s_17}
\]
\[
\text{done; \:! s_17}
\]

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 sublist. After zipping, we fold everything into a list of tuples.

\[
\text{zip_with (fun el e2 \rightarrow \langle -el, -e2 \rangle).}
\]
\[
\text{(of_arr .\:<\:\text{arr1}. \:map (fun x \rightarrow \langle x, x \rangle).}
\]
\[
\text{take (\:<\:\text{arr1}. \:map (fun x \rightarrow \langle x, x \rangle mod 2 = 0.).}
\]
\[
\text{take (\:<\:\text{arr1}. \:map (fun x \rightarrow \langle x, x \rangle)).}
\]
\[
\text{(of_arr .\:<\:\text{arr2}. \:map (fun x \rightarrow \langle x, x \rangle).}
\]
\[
\text{take (\:<\:\text{arr2}. \:map (fun x \rightarrow \langle x, x \rangle mod 2 = 0.).}
\]
\[
\text{take (\:<\:\text{arr2}. \:map (fun x \rightarrow \langle x, x \rangle)).}
\]

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 of the extended version) 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

\[
\text{type } \alpha \:\text{stream_shape} =
\]
\[
\text{| Nil |
\]
\[
\text{| Cons of } \alpha \times \zeta |
\]

then the stream is formally defined as

\[
\text{type } \alpha \:\text{stream} = \forall \omega. (\alpha, \omega) \:\text{stream_shape} \rightarrow \omega \rightarrow \omega
\]

A stream of α 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:
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:

\[
\text{let fold : } (\zeta \rightarrow \alpha \rightarrow \zeta) \rightarrow \zeta \rightarrow \alpha \text{ stream } \rightarrow \zeta =
\text{fun } f \ z \ str \rightarrow
\text{str (function Nil } \rightarrow \ z \ | \ \text{Cons } (a,x) \rightarrow \ f \ x \ a)\]

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:

\[
\text{let map : } (\alpha \rightarrow \beta) \rightarrow \alpha \text{ stream } \rightarrow \beta \text{ stream } =
\text{fun } f \ str \rightarrow
\text{fun folder } \rightarrow \text{str (fun } x \rightarrow \text{match } x \text{ with}
\ | \ \text{Nil } \rightarrow \ z
\ | \ \text{Cons } (a,x) \rightarrow \text{folder } (\text{Cons } (f \ a,x))\]

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., \( \text{of_arr} \), 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 “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.

**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: \( \text{init} \) to initialize the state and obtain the first element, and \( \text{next} \) to advance the stream to the next element, if any. Such a “generator” (or \( \text{IEnumerator} \), 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 \( \text{unfold} \). [20]

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

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 \( \text{Nil} \), 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:

\[
\text{let of_arr : } \alpha \text{ array } \rightarrow \alpha \text{ stream } =
\text{let step } (1,\text{arr}) =
\text{if } i < \text{Array.length arr}
\text{then Cons } (\text{arr.(i)}, (i+1,\text{arr})) \text{ else } \text{Nil}
\text{in fun arr → ((0,arr),step)}\]

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 \( \text{step} \) until it reports the end-of-stream. (Although the types of \( \text{of_arr}, \text{fold}, \text{and map} \), etc. nominally remain the same, the meaning of \( \alpha \text{ stream} \) has changed.)

\[
\text{let fold : } (\zeta \rightarrow \alpha \rightarrow \zeta) \rightarrow \zeta \rightarrow \alpha \text{ stream } \rightarrow \zeta =
\text{fun } f \ z \ (a,\text{step}) \rightarrow
\text{let rec loop } z \ s = \text{match } \text{step} \text{ with}
\ | \ \text{Nil } \rightarrow \ z
\ | \ \text{Cons } (a,t) \rightarrow \text{loop } (f \ a \ t) \ t
\text{in } \text{loop } z \ s\]

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

\[
\text{let map : } (\alpha \rightarrow \beta) \rightarrow \alpha \text{ stream } \rightarrow \beta \text{ stream } =
\text{fun } f \ (s,\text{step}) \rightarrow
\text{let new_step } = \text{fun } s \rightarrow \text{match } \text{step} \text{ with}
\ | \ \text{Nil } \rightarrow \ z
\ | \ \text{Cons } (a,t) \rightarrow \text{Cons } (f \ a, t)
\text{in } (s,\text{new_step})\]

merely transforms its step function: \( \text{new_step} \) calls the old step and \( \text{maps} \) 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 \( \text{take} \), 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 \( \text{stream_shape} \), 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 \( \text{zip} \text{ with} \) 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 \( \text{map} \) stream combinator transforms the current stream element by passing it to some function \( f \) received as an argument of \( \text{map} \). A hand-written implementation would have no other function calls. However, the pull-stream \( \text{map} \) combinator introduces a closure: \( \text{new_step} \), which receives a \( \text{stream_shape} \) value from the old step, pattern-matches on it and constructs the new \( \text{stream_shape} \). The push-stream \( \text{map} \) has the same problem: The step function of \( \text{of_arr} \) 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
a high-level abstraction into efficient code. In fact, the original de-
forestation algorithm in the literature is closely related to par-
tial evaluation. 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. shows that straightforward
staging (or automated partial evaluation) does not achieve full de-
forestation. We have to re-think general stream processing.

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, which is a dialect
of OCaml with MSP extensions. The first MSP feature is brackets,
which enclose a code template. For example, (1+2), is a template
for generating code to add two literals 1 and 2.

The output of the interpreter demonstrates that the code template
is a first-class object; moreover, it is a value: a
Caml can print such values, and also write them into a file to com-

By adding staging annotations. This is not a simple matter of anno-
tating—i.e., placing brackets and escapes on—the original map
code according to the decided binding-times:

```
let map = fun f (s,step) ->
    let new_step = fun s
        with
        | Cons (a,t) -> Cons (~f .a.), t).
    in (s,new_step)
```

The combinator of_arr and fold are staged analogously. We use
the method to prove the correctness, which easily applies
to this case, given that map is non-recursive. The sample processing
pipeline (the first example from

```
let rec loop_1 z_2 s_3 =
    match match match s_3 with
        | (1,4,arr_5) ->
            if 1_4 < (Array.length arr_5)
                then Cons ((arr_5.(i_4)),
                    (1+(i_4 + 1), arr_5))
            else Nil
        | 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 oper-
ator 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 abstrac-
tion overhead and generate code of hand-written quality and per-
fomance. We will be continuing the simple running example of
the earlier sections, of summing up squared elements of an array.
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 return 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:

```plaintext
type α st_stream = σ

let fold : (ω code → α code → ω code) * (ω code) → σ

let rec loop z s = fold (s,step) in
```

That is, a stream is again a pair of a hidden state, σ (only known dynamically, i.e., 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

```plaintext
let map : (α code → β code) → α st_stream → β st_stream =

fun f (s,step) →
let new_step s k = step s @ f in
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:

```plaintext
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 (z, s)) in
in loop ~ z ~ s.
```

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

```plaintext
val c : int code = (λ)

let rec loop_1 z_2 s_3 =
match s_3 with
| (1_4, arr_5) →
  if 1_4 < (Array.length arr_5) then
    let el_6 = arr_5.(1_4) in
    let a'_7 = el_6 * el_6 in
    loop_1 (z_2 + a'_7) ((1_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 \[\] by also using continuation-passing style for the initial state. The definition of the stream type (α st_stream) now becomes:

```plaintext
type α st_stream = σ

let new_step s k = step s @ k in
```

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
and the in-place update of the state (the allocation of the reference cell for the current array index) in `init`, and the in-place update of the state (the `incr` operation):

```ocaml
let of_arr : array code → 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 * array code) code` and becomes `int ref code * 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:

```ocaml
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 (a₀) → (.loop ~(f .(z). a₀)).)
        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:

```ocaml
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 generated 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 subarching 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:

```ocaml
type (α, σ) producer_t =
| For of: 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 =
  ∃θ. (∀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, `up`, 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

```ocaml
((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 `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 §6.3.

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

```ocaml
let of_arr : array code → st_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 : (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 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 ⟨0⟩ fun s0 → (let i = ref 0 in ⟨(k .(i)),s0⟩).
and term (i,s0) = .!(~i) ≤ ⟨(upb s_o)⟩.
and step (i,s0) k = index s_o.!(~i).
00
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 ⟨0⟩ fun e → tr e k in
(init, For (g with index))
| (init,Unfold (step; as g)) → let step s k = step s ⟨0⟩ 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 §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, for getting 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 transformer:

```
type card_t = AtMost1 | Many

type (α,σ) producer_t =
```
let take = fun n → function
| linear (init, For (upb;index)) →
| let upb s = (min (∼(n-1)) ∼(upb s)). in
| Linear (init, For (upb;index))

...  

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 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., `zip`:

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

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 characteristic of the operation `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
```

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 (j, z) k → k (x, y)) @
                                  push_linear (for_unfold prod1)
                                  (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 exactly one element when advancing the state. 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 =
    let str1 = of_arr arr1
    in (init, For {upb; index})
    push_linear (for_unfold prod1)
    push_linear (for_unfold prod2)
    let str2 = of_arr arr1
    in (init, For {upb; index})
    push_linear (for_unfold prod1, nestf1)

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 refines 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 [2];
- **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;
- **flatMapAfter_zipWith:** compute \( \sum (x_i + x_j) y_j \), 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;
- **flatMap_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 of the extended version. 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 (strynon), 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: Batteries [7] 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.

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: \( x_i = i \mod 10 \). 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 `flatMapAfterZipWith` 10,000, for the `zipWithAfterflatMap` 10,000,000 and for the `flatMap_map_take` \( N \) numbers sub-sized by 20% of \( N \).

**Setup:** The system we use runs an x64 OS X 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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Staging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Java 8 (1.8.0.65)</td>
<td></td>
</tr>
<tr>
<td>Scala</td>
<td>2.11.2</td>
<td>LMS 0.9.0</td>
</tr>
<tr>
<td>OCaml</td>
<td>4.02.1</td>
<td>BER MetaOCaml N102</td>
</tr>
</tbody>
</table>

**Automation:** For Java and Scala benchmarks we used the Java Microbenchmark Harness (JMH) 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 `ocamlrun` 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 [15, 39], 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 Synchroné [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. 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 iterates [16] and many of their implementations support incremental processing even in the presence of effects, for nested streams and for several consumers and producers. Although iterates avoid intermediate streams they still suffer large overheads for captured continuations, closures, and coroutine calls.

We disregard the lazy IO misfeature [10].
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 be 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 them-
selves offer no guarantee, even the guarantee that the re-written code is well-typed. Multi-stage programming ensures that all staging annotations are type-correct.

Jonalagedda et al. present a library using only CPS encodings (fold-based) [12]. It uses the Gill et al. foldl/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 `CArray` data type, corresponds to staged arrays (of type `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:

```haskell
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:

```haskell
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:

```haskell
let sum = fold (fun z a → add a z) zero
of_arr arr
  ⊢ map (fun x → mul x x) ⊢ 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.

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 streamonas 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 Reps in Haskell-Cafe, captures well the frustrations of advanced library writers: “The compilation method [...] depends on the GHC simplifier acting in a certain way—and 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


