Stream Fusion, to Completeness

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Aggelos Biboudis
University of Athens, Greece
biboudis@di.uoa.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. According to one approach, 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 [1]. 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 streamlining 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 [23] or Haskell GHC RULES [5, 23] 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-scopeed, 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 [1] 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) [25].

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)

\[
\text{type } \alpha \text{ stream}
\]

Producers

\[
\begin{align*}
\text{val of_arr} & : \alpha \text{ array code } \rightarrow \alpha \text{ stream} \\
\text{val unfold} & : (\zeta \text{ code } \rightarrow (\alpha \times \zeta) \text{ option code}) \rightarrow \\
& \quad \zeta \text{ code } \rightarrow \alpha \text{ stream } \rightarrow \zeta \text{ code}
\end{align*}
\]

Consumer

\[
\begin{align*}
\text{val fold} & : (\zeta \text{ code } \rightarrow \alpha \text{ code } \rightarrow \zeta \text{ code}) \\
& \quad \rightarrow \zeta \text{ code } \rightarrow \alpha \text{ stream } \rightarrow \zeta \text{ code}
\end{align*}
\]

Transformers

\[
\begin{align*}
\text{val map} & : (\alpha \text{ code } \rightarrow \beta \text{ code}) \rightarrow \alpha \text{ stream } \rightarrow \beta \text{ stream} \\
\text{val filter} & : (\alpha \text{ code } \rightarrow \text{ bool code}) \\
& \quad \rightarrow \alpha \text{ stream } \rightarrow \alpha \text{ stream} \\
\text{val take} & : \text{ int code } \rightarrow \alpha \text{ stream } \rightarrow \alpha \text{ stream} \\
\text{val flat_map} & : (\alpha \text{ code } \rightarrow \beta \text{ stream}) \\
& \quad \rightarrow \alpha \text{ stream } \rightarrow \beta \text{ stream} \\
\text{val zip_with} & : (\alpha \text{ code } \rightarrow \beta \text{ code } \rightarrow \gamma \text{ code}) \\
& \quad \rightarrow (\alpha \text{ stream } \rightarrow \beta \text{ stream } \rightarrow \gamma \text{ stream})
\end{align*}
\]

\[\text{Figure 1: The library interface}\]

The generated code contains no extra heap allocations in the main loop (Thm[1]). The code

\[
\begin{align*}
\text{let sum} & = \text{fold (fun } z\ a\ -> (.~a + ~z).)\ (.0). \\
\text{of_arr} & .(\text{arr}). \\
\triangleright \text{map (fun } x\ -> (.~x * ~x).) \\
\triangleright \text{sum}
\end{align*}
\]

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:

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

[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:

\[
\begin{align*}
\text{let ex = of_arr .(arr).} & \Rightarrow \text{map (fun x \rightarrow \langle x  \rightarrow x \rangle.)} \\\\\\\\\\\text{ex} \Rightarrow \text{filter (fun x \rightarrow \langle x \rightarrow \langle x \rightarrow x \rangle \rangle.)} \Rightarrow \text{sum}
\end{align*}
\]

We have abstracted out the mapped stream as ex. The earlier example is, hence, ex \Rightarrow 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 \( n \):

\[
\begin{align*}
\text{ex} & \Rightarrow \text{filter (fun x \rightarrow \langle x \rightarrow \langle x \rightarrow x \rangle \rangle.)} \\\\\\\\\\\text{ex} & \Rightarrow \text{take \( n \).} \\
& \Rightarrow \text{sum}
\end{align*}
\]

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

\[
\begin{align*}
\text{let s_1 = ref 0 in} & \\\\\\text{let arr_1 = ref in} \\\\\\text{let i_3 = ref 0 in} \\\\\\text{let nr_4 = ref n in} \\
\text{while \( \text{nr}_4 > 0 \) \&\& \( i_3 \leq \text{of_arr .(arr) .(of_arr .(arr)).} \Rightarrow \text{map (fun x \rightarrow \langle \langle x \rightarrow x \rangle \rangle.)} \Rightarrow \text{take \( n \).} \Rightarrow \text{sum}
\end{align*}
\]

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 \text{take \( n \).} part of the pipeline contributes to three separate places in the code: where the \text{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: \text{take} and of_arr.

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

\[
\text{zip_with (fun el e2 \rightarrow \langle el * e2 \rangle.)} \\
\text{(of_arr .(arr1).} \Rightarrow \text{(of_arr .(arr2).} \Rightarrow \text{sum}
\]

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

\[
\begin{align*}
\text{let s_17 = ref 0 in} \\\\\\text{let arr_18 = ref in} \\\\\\text{let arr_19 = arr2 in} \\
\text{for i_20 = 0 to} \\\\\\text{min (of_arr .(arr1) .(of_arr .(arr2)).} \Rightarrow \text{sum}
\end{align*}
\]

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.

\[
\begin{align*}
\text{zip_with (fun el e2 \rightarrow \langle el * e2 \rangle.)} \\
\text{(of_arr .(arr1).} \Rightarrow \text{(of_arr .(arr1).} \Rightarrow \text{map (fun x \rightarrow \langle \langle x \rightarrow x \rangle \rangle.)} \Rightarrow \text{take \( i_2 \).} \Rightarrow \text{filter (fun x \rightarrow \langle \langle x \rightarrow x \rangle \rangle.)} \Rightarrow \text{map (fun x \rightarrow \langle \langle x \rightarrow x \rangle \rangle.)} \Rightarrow \text{fold (fun z a \rightarrow \langle \langle a : i_2 \rangle \rangle.)} \Rightarrow \text{sum}
\end{align*}
\]

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.

\[\text{Push Streams.} \]

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 \( \alpha \) as

\[
\begin{align*}
\text{type } \alpha (\omega) \text{ stream_shape } = \\
\text{Nil} \\
\text{Cons } \alpha (\omega) \text{ stream_shape } \rightarrow \omega \\
\end{align*}
\]

then the stream is formally defined as:

\[
\begin{align*}
\text{type } \alpha \text{ stream } & \Rightarrow \omega \Rightarrow (\alpha (\omega) \text{ stream_shape } \rightarrow \omega ) \rightarrow \omega
\end{align*}
\]

A stream of \( \alpha \) is hence a function with the ability to turn any generic “folder” (i.e., a function from \( \alpha (\omega) \text{ stream_shape } \rightarrow \omega \) to a single \( \omega \). The “folder” function is formally called an F-algebra for \( \alpha (\omega) \text{ stream_shape } \text{ functor} \).

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

\[
\begin{align*}
\text{let arr_1 = ref in} \\\\\\text{let arr_19 = arr2 in} \\
\text{for i_20 = 0 to} \\\\\\text{min (of_arr .(arr) .(of_arr .(arr)).} \Rightarrow \text{sum}
\end{align*}
\]

\(3\) 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 fold →
  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:

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

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:

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

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) 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: 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 [20].

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

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 Nil, or the current element and the next state. (Formally, the step function is the F-co-algebra for the \((\alpha,\cdot) \text{ stream_shape} \) 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:

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

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 a stream has changed.)

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

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

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

merely transforms its step function: new_step calls the old step and maps the returned current element, passing it immediately to the consumer, with no buffering. That is, like push streams, pull streams also accomplish fusion. Bifitting 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
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, \(⟨\ldots⟩\), which enclose a code template. For example, \(⟨1+2⟩\) is a template for generating code to add two literals 1 and 2.

\[\begin{align*}
\text{let } c &= \cdot(1 + 2). \\
\text{val } c &: \text{int code} = \cdot(1 + 2).
\end{align*}\]

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 \(cf\) fills the holes with the expression \(c\) created earlier.

\[\begin{align*}
\text{let } cf x &= \cdot(\sim x + \sim x). \\
\text{val } cf &: \text{int code} → \text{int code} = \langle \text{fun}\rangle \\
\text{cf } c &= \cdot : \text{int code} = \cdot((1 + 2) + (1 + 2)).
\end{align*}\]

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).

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 method of annotating one of the standard definitions (either pull- or push-style) of \(\alpha\) 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 runtime (“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 \(\text{map}\) combinator, which, recall, has a type signature:

\[\begin{align*}
type \alpha\text{ stream} = \exists \sigma. \sigma * (\alpha → (\alpha, \sigma)) \text{ stream_shape} \\
\text{val } \text{map} &: (\alpha → \beta) → \alpha\text{ stream} → \beta\text{ stream}
\end{align*}\]

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 \(\alpha\text{ code} → \beta\text{ code}\): given code to compute \(\alpha\), the mapping function, \(f\), is a static way to produce code to compute \(\beta\).

The second argument of \(\text{map}\) is the pull stream, a tuple of the current state \((\sigma)\) 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

\[\begin{align*}
type \alpha\text{ st_stream} = \\
\exists \beta. \beta\text{ code} * (\beta\text{ code} → (\alpha, \beta)\text{ stream_shape code})
\end{align*}\]

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

\[\begin{align*}
\text{let } \text{map} &: (\alpha\text{ code} → \beta\text{ code}) → \\
\alpha\text{ st_stream} → \beta\text{ st_stream} = \\
\text{fun } f (s, \text{step}) → \\
\text{let } \text{new_step} = \text{fun } s → \langle\text{match } \sim(\text{step } s)\text{ with} \\
| \text{Nil} → \text{Nil} \mid \text{Cons } (a, t) → \text{Cons } (∼(f .(\cdot a)), t) , \text{t} \rangle \\
\text{in } s(\text{new_step})
\end{align*}\]

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

\[\begin{align*}
\text{of_arr} &: [(0;1;2;3;4)]. \\
\text{> map} &: (\text{fun } a → (∼a * ∼a),) \\
\text{> fold} &: (\text{fun } x y → (∼x + ∼y),) :0).
\end{align*}\]

then produces the following code:

\[\begin{align*}
\text{let rec } \text{loop}_1 z_2 s_3 = \\
\text{match } \text{match } s_3 \text{ with} \\
| (1, 4, \text{arr}_5) → \\
\text{if } 1 < (\text{Array.length } \text{arr}_5) \text{ then } \text{Cons } ((\text{arr}_5 . (\cdot 1)), \langle(1 + 1), \text{arr}_5\rangle) \\
\text{else } \text{Nil} \\
\text{with} \\
| \text{Nil} → \text{Nil} \\
| \text{Cons } (a_6, t_7) → \text{Cons } ((\cdot a_6 + a_6), t_7) \\
\text{with} \\
| \text{Nil} → z_2 \\
| \text{Cons } (a_8, t_9) → \text{loop}_1 (z_2 + a_8) t_9 \text{ in } \text{loop}_1 0 \langle0, [0;1;2;3;4]\rangle).
\end{align*}\]

As expected, no lists, buffers or other variable-size data structures are created. Some constant overhead is gone too: the squaring operation of \(\text{map}\) is inline. However, the triple-nested match betrays the remaining overhead of constructing and deconstructing \(\text{stream_shape}\) values. Intuitively, the clean abstraction of streams (encoded in the pull streams type of \(\alpha\) 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 (\(\text{of_arr}, \text{map}, \text{and } \text{fold}\)) will synchronize, with all of them processing elements until the same last one. Eliminating the overhead requires a different computation model for streams.
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 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 next stream element or the continuation returns. We do know however that the step function invariantly 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 deconstruction of the stream_shape is eliminated. 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. 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 [H], by also using continuation-passing style for the initial state. The definition of the stream_shape data type (α st_stream) now becomes:

```plaintext
let fold : (∃ ζ. α code → ζ code → ζ code) → ζ code → (∃ ζ. α st_stream → ζ code) = fun f z (s,step) →
  let rec loop z s = s → (step s @ f z).
  match s with
  | Nil → Nil
  | Cons (a,t) → .(let a' = ~(f .(a)) in
    loop~(t).)
  in loop~z~z.
```

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

```plaintext
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;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 [H], by also using continuation-passing style for the initial state. The definition of the stream_shape data type (α st_stream) now becomes:

```plaintext
type α st_stream =
  (∃ σ. α code → σ code) →
  (∃ σ. α code → σ code) →
  (∃ σ. α st_stream → σ code) = fun f s k s ⇒
    let new_step s k = s → (step s @ f z).
    match s with
    | Nil → Nil
    | Cons (a,t) → .(let a' = ~(f .(a)) in
      new_step~(t).)
    in new_step~k~k
```

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, \( \omega \)) 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 \( \text{of}_\text{arr} \) combinator demonstrates the let-insertion (the allocation of the reference cell for the current array index) in \( \text{init} \), and the in-place update of the state (the allocation of the reference cell for the current array index) in \( \text{init} \).

```ocaml
let of_arr : \( \alpha \) array code \( \rightarrow \alpha \) st_stream = let init arr k = 
  \( \langle \text{init arr}, \text{step} \rangle \) k = 
  \( \langle \text{init arr}, \text{step} \rangle \) k = 
  \( \text{arr} \) \( \mapsto \text{arr} \in \sim (k \, (\langle i \rangle \ldots \langle \text{arr} \rangle)) \). 
and step \((i, \text{arr})\) k = 
  \( \langle \text{init arr}, \text{step} \rangle \) k = 
  \( \langle \text{init arr}, \text{step} \rangle \) k = 
  \( \text{arr} \) \( \mapsto \text{arr} \in \sim (k \, (\langle i \rangle \ldots \langle \text{arr} \rangle)) \). 

in 

fun arr \( \rightarrow \langle \text{init arr}, \text{step} \rangle \) 

Once again, until now the state of the \( \text{of}_\text{arr} \) stream had the type \( \langle \text{int} \times \alpha \text{ array code} \rangle \). It has become \( \text{int ref code} \times \alpha \text{ array code} \), the statically known pair of two \( \alpha \text{ code} \) values. The construction and deconstruction of that pair now happens at code-generation time.

The earlier \( \text{map} \) 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 \( \text{fold} \) combinator no longer has to thread the state through its loop:

```ocaml
let fold : \((\langle \text{code} \rightarrow \alpha \text{ code} \rightarrow \zeta \text{ code} \rangle \rightarrow \zeta \text{ code} \rightarrow \alpha \text{ st_stream} \rightarrow \zeta \text{ code} \rangle 
  = \text{fun f z (init,step) \rightarrow init } \text{Nil} \rightarrow \ldots z. 
  | \text{Cons (a, \_)} \rightarrow \ldots (\text{loop } \sim (f . (z, a))). 

in \text{loop } \sim z.
```

It obtains the state from the initializer and passes it to the \( \text{step} \) 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 < \! \text{Array.length} \! arr_9 then 
    let el_12 = arr_9.(! i_8) in 
    incr i_8; 
    let el_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 \((\alpha \text{ code}, \text{unit}) \text{ stream_shape} \) in the \( \alpha \text{ st_stream} \) type looks odd: the \( \text{stream_shape} \) data type has become artificial. Although \( \text{unit} \) 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 subadding or zipping (described in detail in \( \S 6 \) 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 \((\alpha, \sigma) \) \text{ producer}_t =
| \text{For} of \( \upb \) : \( \sigma \rightarrow \text{int code} \); 
  \text{index}: \( \sigma \rightarrow \text{int code} \rightarrow (\alpha \rightarrow \text{unit code}) \rightarrow \text{unit code} \) 
| \text{Unfold of} \ (\text{term: } \sigma \rightarrow \text{bool code} \); 
  \text{step: } \sigma \rightarrow (\alpha \rightarrow \text{unit code}) \rightarrow \text{unit code} \) 
and \( \alpha \text{ st_stream} = \exists \sigma. (\forall \omega. (\sigma \rightarrow \omega \text{ code}) \rightarrow \omega \text{ code}) \ast 
(\alpha, \sigma) \text{ producer}_t 
and \( \alpha \text{ stream} = \alpha \text{ code} \text{ st_stream} \)
```

That is, a stream type is a pair of an \( \text{init} \) 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 \( \text{index} \) 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

\[(\alpha \text{ code}, \text{unit}) \text{ stream_shape} \rightarrow \omega \text{ code}\) \rightarrow \omega \text{ code}\]

into two components: the termination test, \( \text{term} \), producing a dynamic bool value (if the test yields true for the current state, the loop is finished) and the \( \text{step} \) 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 (\( \text{fold} \) state from one iteration to another. It is easier to accumulate the state in a mutable cell. Therefore, the answer type of the \( \text{step} \) and \( \text{index} \) functions can be \text{unit code} rather than \( \omega \text{ code} \).

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 \( \text{step} \) or \( \text{index} \) functions has the type \( \alpha \) 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 \( \S 6 \).

We can now see the new design in action. The stream producer of \( \text{arr} \) is surely the for-loop-style producer:

```ocaml
let of_arr : \( \alpha \text{ array code} \rightarrow \alpha \text{ stream} \) \text{ = fun arr } \rightarrow 
  let init k = .(let arr = \sim \text{arr} \in \sim (k . (\text{arr}))). 
```

\(^6\) Actually, our benchmarking reveals that for- and while-loops are currently faster even in OCaml.
and upb arr = ⟨(Array.length ~arr - 1), (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 @0 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) @0 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 code → β unit code) → unit code |
| (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,s0)).
| and step (i,s0) k = index s0 .!(~i) @0 fun a → .(incr ~i; ~(k a)). in (init, Unfold (term;step)) |
| x → x

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 → .(.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-annotation 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 §6.3.

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

type card_t = AtMost1 | Many

type (α,σ) producer_t =
For the producer, noting if it generates possibly many elements or at most nested streams. We also added a cardinality marker to the general filter of a sum data type with variants "Linear": nesting of a stream that produces at most one element:

\[ \text{take} \]

For example, the stream combinator 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:

\[ \text{let take = fun n → function} \]

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

\[ \text{let take = fun n → function} \]

The idea is straightforward: allocate a reference cell \( n \) with the remaining element count (initially 0). add the check \(!n > 0\) to the termination condition of the stream producer, and arrange to decrement the \( n \) 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 \(!n > 0\) has to be added not only to the first producer but to the producers of all nested substrings: this is the role of function \( \text{more_termination} \) (see the accompanying code for its definition) in the fragment above. The operation \( \text{add} \) allocates cell \( n \) 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 \( \text{for_unfold} \), which we defined in §5. The operation \( \text{add} \) (definition not shown) also adds \( n \) to the produced value: The result of \( \text{add} \) \( n \) (for_unfold p) of type \( \text{int_ref code, α code} \) \( \text{st_stream} \). Adding the operation to decrement \( n \) is conveniently done with \( \text{map_raw} \) 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:

\[ \text{val zip_with : (α code → β code → γ code) → (α stream → β stream → γ stream)} \]

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 \( \text{map_raw} \) operation turn out to be important abstractions that make the problem tractable.

One of the use cases of \( \text{zip_with} \) 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:

\[ \text{val zip_raw : (α code → β stream → γ code) → (α stream → β stream → γ stream)} \]

The auxiliary \( \text{zip_raw} \) 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 \( \text{zip_raw} \) 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,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 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 =
  fun p1 p2 -> match (p1,p2) with
  | (11,For f1), (12,For f2) ->
  let init k =
  11.init ℧ fun s1 ->
  12.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
  > flatten_map (fun _ -> of_arr .[[1;2];]).
let str3 = of_arr arr1
  > flatten_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 stre after we are sure to have obtained the element of the non-linear stream strq. 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 el k ->
           stre.step sq (fun e2 -> k (e1,e2))) stre

The above code is an outline: we have to initialize stre 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 de-construct 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 4.2 on;
- **sumOfSquaresEven:** the sumOfSquares benchmark with added filter, summing the squares of only the even array elements;
- **cart:** \( \sum x_i y_i \) 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 \( \sum (x_i + x_i) y_i \), 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 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 (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.

Batteries[7] in OCaml and the standard Java 8 and Scala streams. As there is not a unifying module that implements all the combinatorial types sub-sized by the zipWith function. The setup, we employed, 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 test we used 10, 000, 000 integers, for the flatMapAfter_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 of 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 [5, 7], 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 [13] 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” [13]. 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 Synchronous [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 [14]. The all-too-common memory leaks point out that lazy lists do not offer, either by design, stream fusion. Overcoming the drawbacks of lazy lists, coroutine-like iterates [15] and many of their reimplementations 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.

---

[1]: 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 [15] and many of their reimplementations 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.

[3]: 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 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 \rightarrow \text{even} x \text{ of }
    True \rightarrow \text{enumFromToS } 1 x
    False \rightarrow \text{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:

```
\text{flat_map_cps} (\text{fun } k \rightarrow 
    .(\text{if } (\text{even} \sim x) \text{ then } \sim(k \text{ enumFromToS ...))}
    \text{else } \sim(k \text{ 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.

Jonnalagedda et al. present a library using only CPS encodings (fold-based) [13]. It uses the Gill et al. fold/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 [12] 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 coexist 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 unsafe data type, corresponds to staged arrays (of type α array code) in our work. The library they derive from the defunctionalization of Pull 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 (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 let expressions (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.

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 streamonias 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


A. Generated code for the Complex example

We show the generated code for the last example of Section 6 repeated below for reference:

\[
\begin{align*}
\text{(of_arr } &\text{::}) \\
\text{of_map } &\text{::} \\
\text{of_filter } &\text{::} \\
\text{of_loop } &\text{::}
\end{align*}
\]

The generated code is:

```
let s_23 = ref [ ] in
let arr_24 = arr1 in
let i_25 = ref 0 in
let size_2 = Array.length arr_1 in
let adv_32 = ref None in
curr_26 := None;
while (! size_2 > 0) do
  let rec loop_1 acc_1 =
    \if iarr_1
    acc_1
    \else
    let mapsS
    \fi
  in
  incr iarr_1;
  incr acc_1_tmp
done
```

```
let cart = fun (arr1, arr2) ->
  of_arr arr1
  of_map (fun y -> .(~ x * y)).
fold (fun z a -> .(~ z + a)). (.0).;
```

B. 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 adv_27 = ref None in
let loop_2 = ref 0 in
let rec loop_1 acc_1 =
  adv_32 ();
let s_33 = ref (Some (1, (1 + 1))) in
let rec loop_2 acc2_tmp =
  \if ! iarr_1
  \if iarr_1
  acc_2
  \else
  let el_2 = arr2.(iarr_1) in
  incr iarr_2;
  incr acc2_tmp
  in
  loop_2 acc2_tmp
  \fi
  \fi
```

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 adv_27 = ref None in
let loop_2 = ref 0 in
let rec loop_1 acc_1 =
  \if ! iarr_1
  \if iarr_1
  acc_1
  \else
  let el_2 = arr2.(iarr_1) in
  incr iarr_1;
  incr acc_1_tmp
  in
  loop_1 acc_1_tmp
  \fi
  \fi
```

D. Streams and baseline benchmarks

```
let sumS
  = fun arr ->
    of_arr arr
    fold (fun z a -> .(~ z + a)). (.0).;
```

```
let sumShabd
  = 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 sumOfSquaresShabd
  = 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)).
```
let maps_hand = fun (arr1, arr2) -> let sumOfSquaresEvenShand = fun arr1 -> let filters_hand = fun arr1 -> let sumOfSquaresEvenShand = fun arr1 -> let cartS = fun (arr1, arr2) -> let cartShand = fun (arr1, arr2) -> let dotProductS = fun (arr1, arr2) -> let dotProductShand = fun (arr1, arr2) -> let flatMap_after_zipWithShand = fun (arr1, arr2) -> let zipWith_after_flatMapShand = fun (arr1, arr2) ->
incr i1;
flag1 := (!i1) ≤ ((Array.length ~arr1) - 1));
sum := ((!sum) + (elz + e11 + e12)) done
done;
!sum).;;

let flat_map_takeS
= fun (arr1, arr2) →
of_arr arr1
d flat_map (fun x → of_arr arr2
d map (fun y → .(~x * ~y).))
d take.(20000000).
d 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
  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).;;