# Lazy on Principle

Anonymous Author(s)

## Abstract

This is the abstract.

**ACM Reference Format:**
Anonymous Author(s). 2024. Lazy on Principle. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

When sequences of data become too large to fit into memory at once, programs need to process them lazily. From the humble iterator to asynchronous APIs for streams and sinks with error handling and buffering, every language needs libraries for working with lazy sequences.

For such a fundamental, conceptually simple, and language-agnostic problem, one might expect a principled, unified solution that programming language designers and library authors can turn to and implement in their language of choice.

But the opposite is the case. Learning a new programming language implies learning yet another, slightly (or not so slightly) different set of APIs for working with sequences. Even within a single language, there are often competing libraries — section 9 lists some thirty popular Javascript libraries alone.

Starting from "*Which* abstraction is the best?", we quickly moved to "*Is* there a best abstraction?", and then to the more constructive "*What* would make an abstraction the best?". In this paper, we present our answers to these questions. In a nutshell:

1. Abstractions for working with lazy sequences in the wild are ad-hoc designs.
2. We propose a principled way of evaluating them.
3. No prior abstractions satisfy all evaluation criteria.
4. We develop abstractions that do.
5. ~~Everybody everywhere should use our abstractions without further reflection.~~

Note that we will focus on strictly evaluated languages. This makes explicit the design elements that enable laziness.


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We further restrict our focus to the two simplemost ways of interacting with a (possibly infinite) sequence: *consuming* a sequence item by item, or *producing* a sequence item by item. Both modes of interaction are of great practical interest, they correspond, for example, to reading and writing bytes over a network. We do not consider more complex settings such as random access, or mixing reading and writing.

### 1.1 Evaluating Sequence APIs

Equipped with a vague notion of wanting to "lazily consume or produce sequences", how can we do better than simply trying to find a design that satisfies all use-cases we can come up with? In mathematics, one would define a set of criteria that a solution should satisfy, in a way that makes no assumptions about any possible solutions themselves.

For example, a mathematician might want to work with numbers "with no gaps in-between" (i.e., the real numbers). They might formalize this intuitive notion as a minimal, infinite, complete ordered field. Any candidate construction (say, the Dedekind cuts of rational numbers), can now be objectively measured against the requirements. As an added bonus, it turns out that all constructions satisfying the abstract requirements are isomorphic. Some constructions might be more convenient than others in certain settings, but ultimately, they are all interchangeable.

This approach of construction-independent axiomization is the only way we can conceive to clear the proliferation of competing library designs.

Sadly, we could not find an airtight mathematical formalism to capture our problem space. The criteria we now present leave gaps that must be filled by argumentation rather than proof, the API design still remains part art as much as science. This makes the following paragraphs the weakest link of this paper. We nevertheless think that both our approach and the designs it yields are novel — and useful.

The criteria by which we shall evaluate lazy sequence abstractions are minimality, symmetry, and expressivity.

**Minimality** asks that no aspect of the API design can be expressed through other aspects of the design. Removing any feature impacts what can be expressed.

**Symmetry** asks that reading and writing data should be dual. The two intuitive notions of producing and consuming a sequence item by item are fully symmetric and sit on the same level of abstraction. Any API design that introduces an imbalance between the two must either be contaminated with incidental complexity, or it must be missing functionality for one of the two access modes.

**Expressivity** asks that the API design is powerful enough to get the job done, but also no more powerful than necessary.

This is by far the most vague of our criteria, because we cannot simply equate more expressivity with a better design. We *can*, however, draw on the theory of formal languages to categorize the classes of sequences whose consumption of production can be described by an API. Some of these classes are more natural candidates than others.

Of these criteria, minimality is arguably the least controversial. Symmetry turns out to be the one we generally find the most neglected in the wild, and strict adherence to symmetry shapes the designs we propose to differ significantly from any others we are aware of. Expressivity might have the weakest definition, but turns out to be rather unproblematic: real-world constraints on the APIs lead to a level of expressivity that also has a convincing formal counterpart — the $\omega$-regular languages (see section 5.1 for details) — making us quite confident about the appropriate level of expressivity.

TODO argue for *consistency* as a cure to the painfulness of building async stuff

To obtain a first indicator for an appropriate level of expressivity, we examine the world of non-lazy sequences, i.e., sequences that can be fully represented in memory.

### 1.2 Case Study: Strict Sequences

Representing sequences in memory can be done in such a natural way that we have never seen any explicit discussion. We shall assume a typical type system with product types (denoted $(S, T)$), sum types (denoted $S+T$), and homogeneous array types (denoted $([T])$).

Let $T$ be a type, then $T$ is also the type of a sequence of exactly one item of type $T$. Now, let $S$ and $T$ be types of sequences. Then $(S, T)$ denotes the concatenations of sequences of type $S$ and sequences of type $T$, $S + T$ denotes the sequences either of type $S$ or $T$, and $[T]$ denotes the concatenations of arbitrarily (but finitely) many sequences of type $T$. None of this is particularly surprising, we basically just stated that algebraic data types and array types allow you to lay out data sequentially in memory.

Slightly more interesting is the blatant isomorphism to regular expressions. Each of the "sequence combinators" corresponds to an operator to construct regular expressions; the empty type and the unit type correspond to the neutral elements of the choice and concatenation operator respectively.

This is useful for making our expressivity requirement for lazy sequence APIs more precise: if the natural representation of strict sequences admits exactly the regular languages, then the regular languages are also the natural candidate level of expressivity for lazy APIs.

Unlike strict sequences that have to fit into finite memory, lazy sequences can be of infinite length. The natural generalization of the regular languages are the $\omega$-regular languages. Hence, this is the level of expressivity we want to see in lazy APIs.

The strict case also neatly supports the design goals of minimality and symmetry. Removing any combinator leads to a strictly less expressive class of languages, and every operator comes both with a way of building up values and with a way of accessing values.

By generalizing the strict case to the lazy case, we can make our requirement of expressivity more precise, leading us to our final set of requirements: We want APIs for lazily producing or consuming sequences an item at a time, such that there is a one-to-one mapping between API instances and $\omega$-regular languages, no aspect of the APIs can be removed without loosing this one-to-one mapping, and there is full symmetry between consumption and production of a sequence. Still not entirely formal, but close enough to meaningfully evaluate and design APIs.

### 1.3 Organization

TODO

## 2 Related Work

TODO

Iterees: https://link.springer.com/chapter/10.1007/978-3-642-29822-6_15 https://hackage.haskell.org/package/iterIO-0.2.2/docs/Data-IterIO.html

session types

pull-stream paper

http://jiangxi.cs.uwm.edu/publication/rebls2020.pdf

https://arxiv.org/pdf/1612.06668.pdf (good related work)

## 3 Evaluating Abstractions

We now start by introducing a notation for API designs. We then express several APIs we encountered in the wild in our notation, in order to build intuition, demonstrate sufficient notational expressivity, and to highlight typical violations of our design goals in popular APIs. We do not aim for an exhaustive survey of APIs, we merely select some examples to illustrate our points.

In the following, we use uppercase letters as type variables. $(S, T)$ denotes the product type of types $S$ and $T$ (intuitively, the cartesian product of $S$ and $T$), and $()$ denotes the unit type (intuitively, the type with only a single value). $S|T$ denotes the sum type of $S$ and $T$ (intuitively, the disjoint union of $S$ and $T$), and ! denotes the empty type (intuitively, the type that admits no value). Finally, we write $S \to T$ for the type of (pure) functions with an argument of type $S$ and a return value of type $T$. Note we take a purely functional approach here: a function does not mutate its argument, it simply produces a new value.

We specify an API as a list of named types (typically functions). Each API can quantify type variables that can be used in its *members*[1]. As an example, consider the following API:

[1] More formally, this is a notation for ad-hoc polymorphism like Haskell's type classes, Java's interfaces, or Rust's traits.

```
API Iterator<P, I>
    next: (P) -> (I, P) | ()
```

This pseudo-type fragment states that in order to obtain a concrete `Iterator`, one needs two types: a type *P* (**P**roducer) and a type *I* (**I**tem). These types have to be related through existence of a function `next`, which maps a producer to either an item and and a new producer, or to a value that signifies that no further items can be produced.

To consume this iterator, one would repeatedly call `next` on the producer returned from the prior call of `next`, until no further Items are produced.

A concrete example of an iterator are the homogenous arrays of *I*s as producers of *I*s; `next` returns () for the empty array, otherwise it returns the first item in the array and the array obtained by removing the first item.

This API is completely stateless, we never mutate any *P*. In an imperative programming language, one would typically use a function that takes a reference to a *P* and returns either an *I* or (), and then makes all implementors pinky-swear to not invoke the function with a *P* that has returned () previously.

We prefer the purely functional notation, because it can express the pinky-swearing API contract on the type level. But all our designs can easily be translated into an imperative, stateful setting. The other way around, by converting stateful references into input values and output values, we can represent APIs from imperative languages in our notation. For example, this `Iterator` API captures the semantics of commonly used iterator APIs such as those of Python[2] or Rust[3]. It handily abstracts over the fact that Rust has actual sum types, whereas Python signals the end of iteration with an exception.

Without devoting too much space to it, we want to point out that even for such a simple special case of sequence processing as iteration — i.e., synchronous production of finite sequences without buffering or error handling — there are several, non-isomorphic approaches in the wild. Both Java[4] and Javascript[5] use APIs with slightly different properties; they differ in how early the code interacting with the iterator knows that no further items will be produced.

Given the symmetry between producing and consuming data, the virtual non-existence of (synchronous, non-error-handling) APIs for consuming data step by step is jarring. Java has an `Appendable` interface[6], but it is specialized to characters, and has a much less prominent role than the `Iterator` interface. Clojure complects both production and consumption into a single Sequence interface[7]. Several languages do give the opposite of iterators a prominent role: for loops that consume an iterator item by item. Baking sequence production via first-class values into a language while turning sequence consumption into a purely syntactic component with no run-time presence massively breaks symmetry from a high level design perspective.

Moving beyond synchronous iterators, *asynchronous* producers — often called *streams* — typically come with buffering and error handling. We shall abstract over[8] both buffering and asynchronicity, as they do not influence matters of minimality, symmetry, or expressiveness regarding formal languages. We believe that designing a solid API irrespective of buffering and asynchronicity and then adding them in a way that is idiomatic for the programming language in question leads to clearer designs. In particular, we see no reason why synchronous and asynchronous sequence APIs in the same language should not be completely analogous.

An example where the asynchronous producer API is equivalent to the synchronous iterator API is — as of writing — Rust[9]. This is rather atypical, because asynchronous APIs, which are often motivated by networking, usually emphasize error handling. In the iterator API, one can use a sum type of actual items and an error type as the type *I*, but on the type level, it remains possible to continue iterating after an error. The API is accurate for recoverable errors only.

A different example is Swift[10], which superficially appears to offer an equivalent API, but the language-level feature of throwing exceptions allows the ability to express irrecoverable errors[11]. In our notation, the Swift API is:

```
API FallibleIterator<P, I, E>
    next: (P) -> (I, P) | () | E
```

This particular API design violates minimality: removing the option of returning the unit type would leave the degree of expressivity unchanged, since one could always instantiate *E* as a sum type of the unit type and an actual error type. It might be tempting to argue that the Swift API communicates intent more clearly, and allows for nicer library functions for working with streams. But these conveniences and specializations could just as well be implemented as special cases

[2] https://wiki.python.org/moin/Iterator
[3] https://doc.rust-lang.org/std/iter/trait.Iterator.html
[4] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
[5] https://tc39.es/ecma262/multipage/control-abstraction-objects.html#sec-iterator-interface
[6] https://docs.oracle.com/javase/8/docs/api/java/lang/Appendable.html
[7] https://clojure.org/reference/sequences#_the_seq_interface
[8] The seasoned Haskell-affectionado will immediately see that parameterizing our presentation over a monad for effectful computation [Wad95] would restore rigorous reasoning to our handwavy act of abstraction. We posit that to the readers who are already familiar with effect-management through monads, filling in the blanks is an easy exercise. To those readers who are not famliar with this technique — i.e., the vast majority of practicioners we would like to reach — obscuring our presentation behind higher-kinded type constructors poses an unnecessary barrier to access.
[9] https://docs.rs/futures/0.3.30/futures/stream/trait.Stream.html
[10] https://developer.apple.com/documentation/swift/asyncsequence
[11] https://developer.apple.com/documentation/swift/asyncthrowingstream

of the more general, underlying pattern. Rust nicely demonstrates this by offering a host of functions[12] for working with streams whose items are a sum type of actual items and error values. Offering a special case as the most fundamental API, like Swift does, unnecessarily reduces expressivity.

Another school of APIs defines asynchronous producers in terms of all the ways in which one would commonly interact with them: mapping, reducing, piping into a consumer, etc. Examples include Java[13] or Dart[14]. Such designs are not concerned with minimality at all. Our preferred approach is to find the minimal interface that that allows expressing all these higher-level functions on top.

The story for asynchronous *consumers* is usually highly asymmetrical again. Swift, for example, has asynchronous for loops to consume streams. Rust offers a *sink* abstraction[15] as the opposite of a stream, but the designs are highly asymmetric — the sink abstraction requires four functions to the stream abstraction's one function.

# 4 Deriving Our Design

From this backdrop of rather inconsistent designs, we now present our alternatives. We develop — in tandem — APIs for producing and consuming sequences, guided by minimality, symmetry, and expressivity (section 4.1). Next, we argue that the designs are indeed sufficiently symmetric and of appropriate expressiveness (section 5) — if our arguments are not fully formal, they are at least *formalizable*. Finally, we sketch some consequences the designs could have for language-level features such as generators or for loops (section 6).

## 4.1 Deriving Our Design

# 5 Evaluating Our Design

Symmetry: in-memory channel, and buffer. Also, encoder and decoders.

Justifying symmetry: ultimately, you want to pipe or buffer. Also, getting this right is simpler than getting them wrong.

## 5.1 Expressivity

Expressivity: omega-regular expressions. Example uses.

# 6 Sugary Syntax

conducer (aka super-coroutine, Inum, generalized anti-pipe)

pro and con as specialiazations of conducers, and alternatives to generators and for loops respectively.

# 7 Beyond the Basics

Combinators.

Lengths and limits.

Asynchrony.

Buffered.

Slices. (should be strict subtypes)

Trees. DAGs? Digraphs?

Coroutines/generators generalized.

Turing machines.

Elastic Turing machines.

Outside movement.

from sequences to structured data?

from sequences to sets (should be easier)?

proper duality?

# 8 Conclusion

TODO

vision of unified, consistent APIs throughout a full language ecosystem.

On a meta note, this paper constitutes evidence that you can share research results with broad applications to a wide range of programming practitioners, without assuming the kind of deep, intuitive familiarity with the Haskell standard library that requires years of practice to obtain. How did that ever become accepted practice in the first place?

# 9 Appendix A: Javascript Libraries

This list of javaScript libraries for working with lazy sequences is intended to demonstrate that there is a clear need for a solid design that people can fall back to rather than reinventing ad-hoc wheels over and over. We list libraries with at least 200 stars on Github, as of February 2024, found by searching Gihub for "stream", "observable", and "reactive".

- https://github.com/staltz/xstream
- https://github.com/mafintosh/streamx
- https://github.com/getify/monio
- https://github.com/getify/asynquence
- https://github.com/cyclejs/cyclejs
- https://github.com/winterbe/streamjs
- https://github.com/winterbe/sequency
- https://github.com/pull-stream/pull-stream
- https://github.com/dionyziz/stream.js
- https://github.com/caolan/highland
- https://github.com/kefirjs/kefir
- https://github.com/baconjs/bacon.js
- https://github.com/cujojs/most
- https://github.com/callbag/callbag
- https://github.com/paldepind/flyd

The following libraries do not explicitly define *streams*, but they do work with *observables*. Observables are an abstraction for values that (discretely) vary over time. For most intents and purposes, this is isomorphic to the notion of a stream.

- https://github.com/reactivex/rxjs
- https://github.com/tc39/proposal-observable
- https://github.com/zenparsing/zen-observable

[12] https://docs.rs/futures/latest/futures/stream/trait.TryStreamExt.html
[13] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
[14] https://api.dart.dev/stable/3.3.0/dart-async/Stream-class.html
[15] https://docs.rs/futures/0.3.30/futures/sink/trait.Sink.html

- https://github.com/vobyjs/oby
- https://github.com/adamhaile/S
- https://github.com/luwes/sinuous
- https://github.com/mobxjs/mobx
- https://github.com/fynyky/reactor.js
- https://github.com/ds300/derivablejs
- https://github.com/elbywan/hyperactiv
- https://github.com/component/reactive
- https://github.com/mattbaker/Reactive.js

These libraries exist in addition to language-level or runtime-level APIs such as the following:

- Node JS Streams, and their evolution:
  - streams0
  - streams1
  - streams2
  - streams3
- WHATWG Streams
- ECMAScript Iterator
- ECMAScript AsyncIterator

## References

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, pages 24–52. Springer, 1995.