Strict Principles for Lazy Sequences

Aljoscha Meyer
Technical University Berlin
Berlin, Germany
research@aljoscha-meyer.de

Abstract

Many common programming tasks, such as networking, are conceptually about lazily working with sequences of unknown length. There are plenty of APIs to choose from — stream and sink, reader and writer, iterator and oops-missing-counterpart. But these APIs typically vary between languages or libraries. Even within a single ecosystem, there often are inconsistencies between the processing models the different APIs induce.

We argue that a unified design is possible. We aim to provide a starting point for future language and library designers, as well as raise several interesting research questions that arise from taking a principled look at lazy sequences.

ACM Reference Format:
Aljoscha Meyer. 2024. Strict Principles for Lazy Sequences. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 14 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 — appendix A lists some thirty popular Javascript libraries alone.

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

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 essay, 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.~~

We further argue that common designs are needlessly inconsistent.

To give a concrete example: in Rust, the (de-facto standard) `Stream` API[1] for receiving data from an asynchronous data source has no notion of irrecoverable errors. Meanwhile, the `Read` API[2] for receiving many items from a data source with a single function call has a notion of irrecoverable errors. Items are hardcoded to individual bytes, however, and errors are hardcoded to a general-purpose I/O error type. Thus, it is neither possible to turn any `Stream` into a `Reader`, nor the other way around.

Conceptually, these two APIs deal with the same issue: lazily producing a sequence. Rust has language-level mechanisms for expressing specialization of APIs and subtyping relations between APIs. Yet each of these abstractions is defined in isolation, with fundamentally different choices of expressivity that make it impossible to fluidly convert between them.

Additional issues arise when considering the opposite of receiving data: sending data off to be processed somehow. The asynchronous, single-item `Sink`[3] is the conceptual analogon of the `Stream` API, yet it also supports irrecoverable errors. On the other hand, while the `Stream` API has a synchronous counterpart in the `Iterator` API[4], there is no such counterpart for `Sink`s.

Such a lack of consistency causes unnecessary education efforts, forces programmers to adopt inefficient code (raw bytes are not the only items for which bulk processing is more efficient than individual processing), and introduces

[1] https://docs.rs/futures/0.3.30/futures/stream/trait.Stream.html
[2] https://doc.rust-lang.org/std/io/trait.Read.html
[3] https://docs.rs/futures/0.3.30/futures/sink/trait.Sink.html
[4] https://doc.rust-lang.org/std/iter/trait.Iterator.html

frustratingly arbitrary barriers to expressing a conceptual architecture of data flows and error handling as actual code.

Hence, the abstractions we propose emphasize consistency, and they build on top of each other. While that *should* sound boring and obvious, it apparently *is* not.

To keep the scope manageable, we 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 TCP connection. We do not consider more complex settings such as random access in our main treatment.

We assume a strictly evaluated language. This makes explicit the design elements that enable laziness.

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 reasonably expect to bring clarity to the proliferation of competing API designs for lazy sequences.

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. We nevertheless think that both our approach and our results 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 is either contaminated with incidental complexity, or it lacks 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. 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 ω-regular languages (see section 3.3 for details) — making us quite confident about the appropriate level of expressivity.

To obtain a good indicator for an appropriate level of expressivity, we examine the world of non-lazy sequences, i.e., sequences that can be fully stored 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 to infinite strings are the ω-regular languages. Hence, this is the level of expressivity we want to see in lazy APIs.

The strict case also neatly validates 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 one item at a time, such that there is a one-to-one mapping between API instances and ω-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 sequences. Still not entirely formal, but close enough to meaningfully evaluate and design APIs.

2 Describing Abstractions

We now start by introducing a notation for API designs. 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 (the type that admits no values). Finally, we write $S \rightarrow T$ 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 function declarations[5]. As an example, consider the following API:

```
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 a call returns $()$.

A concrete example of such an iterator are the homogenous arrays of Is as producers of items of type Is; `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 make 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[6] or Rust[7]. It handily abstracts over the fact that Rust has actual sum types, whereas Python signals the end of iteration with an exception.

Lazy sequence abstractions often come up in the context of asynchronous programming. Programming languages typically have an idiomatic approach to asynchronous functions; most modern languages have them return a `Future<T>` or `Promise<T>`, that is, a value that represents that some value of type T will become available in the future. Other approaches include passing the code to process the result of the asynchronous function as a continuation (often called a *callback*), or concurrency via lightweight process abstractions.

We posit that there is little reason for the sequence abstractions to differ between synchronous and asynchronous settings. In most modern languages, the change to convert a synchronous function signature to an asynchronous one is purely mechanical. Hence, we will implicitly abstract over asynchrony and not mention it in our API designs.

For completeness sake, we should mention that there also are techniques for *explicitly* abstracting over asynchrony and other *effects* via monadic effect management [Wad95]. To the readers already familiar with this technique, adjusting our designs is not difficult. To everyone else — i.e., to the vast majority of practicioners we would like to reach — obscuring our presentation behind higher-kinded type constructors poses an unnecessary barrier to access. Thus we keep the act of abstraction implicit. We point the interested reader to a fairly recent example of a monad for asynchrony [ZBL20], as well as to the alternate formalism of asynchronous *algebraic* effects [Lei17][AP21].

Another kind of effect that *will* come up later is that of errors. Similar to how an asynchronous function is like a regular function but might take its time, a fallible function is like a regular function but might abort with an error. And similar to how modern languages have their idioms for asynchrony (often, async-await syntax), they also have idioms for fallible computations (often, try-catch syntax). Unlike asynchrony, error handling has some interesting implications for

[5]More formally, this is a notation for ad-hoc polymorphism [WB89] like Haskell's type classes, Java's interfaces, or Rust's traits.

[6]https://wiki.python.org/moin/Iterator

[7]https://doc.rust-lang.org/std/iter/trait.Iterator.html

communication flows in the API designs, so we return to the topic proper in section 3.2.

3 A Principled Design

We now derive APIs for producing and consuming sequences one item at a time, guided by minimality, symmetry, and expressivity (section 3.1). The issue of error handling deserves its own discussion (section 3.2). Finally, we argue that the designs are indeed sufficiently symmetric and of appropriate expressiveness (section 3.3) — while our arguments are not fully formal, they are at least *formalizable*.

3.1 Deriving Our Design

To derive a principled design step by step, we start with simplemost producer API: a producer that emits an infinite stream of items of the same type.

```
API InfiniteProducer<P, I>
    produce: P -> (I, P)
```

An iterator, in contrast, expresses a *finite* stream of items, by making the result of a sum type with a unit type option to signal termination. This is not to say that you could not implement infinite iterators, but the typing for those is unprecise — a statically typed language forces programmers to provide code for the end-of-iteration case, even though they might now it will newer occur.

We can easily abstract over both finite and infinite producers through a simple realization: we can rewrite the `produce` function of the `InfiniteProducer` as a sum with the empty type, without changing the semantics at all (it is impossible to provide an instance of the empty type, so the function must always return another item when called):

```
API AlsoAnInfiniteProducer<P, I>
    produce: P -> (I, P) | !
```

Now, the infinite producer and the finite iterator have the exact same form, and we can introduce a type parameter for the summand to express either:

```
API Producer<P, I, F>
    produce: P -> (I, P) | F
```

Setting the type parameter `F` (for *final item*) to `!` or `()` yields the infinite streams and the finite streams over a single type of items, respectively.

Another natural choice for `F` are irrecoverable error types; most APIs with this design designate the type parameter as a type of errors explicitly. This denotation obscures how the same abstraction can also represent iterators or infinite streams, however.

Moreover, it obscures that `F` might be another producer itself, with which to continue production. Through this use of the API, we can effectively concatenate any producer after any finite producer. This usage is the cornerstone of achieving the expressivity of the ω-regular languages, and one we have not encountered in the wild at all.

To give a tangible example of how this degree of expressivity can be useful, consider a networking protocol that proceeds in stages: first a handshake for connection establishment, followed by an exchange of key-value pairs that signify the capabilities of an endpoint, followed by the application-level message exchange. With an API parameterized over arbitrary final values, you can implement each stage in a type-safe way, and then concatenate the stages both in execution and on the type-level. Traditional APIs force programmers to either lump the different kinds of messages (handshake, key-value pairs, application-level) into a single sum type, or to forego helpful typing altogether and operate on the level of bytes.

A symmetric consumer API should be one that can be given either an item of type `I` — returning a new consumer value to continue the process — or a final item of some type `F` — without returning a consumer to continue with.

Ideally, we should be able to mechanically derive this API as a dual of the producer API. A tempting option is to "flip all arrows" and simply swap argument and return type of the `produce` function:

```
API Recudorp<P, I, F>
    ecudorp: ((I, P) | F) -> P
```

We can clean this up by splitting the function of a sum type argument into two independent functions (the resulting types are isomorphic), and giving more conventional names:

```
API NotQuiteConsumer<C, I, F>
    consume: (I, C) -> C
    create: F -> C
```

Unfortunately, this does not give the kind of API we were hoping for. The `consume` function is appropriate, but the second function is not closing a consumer, but creates a consumer. A straightforward dual construction gives too strong of a reversal to yield an API suitable for practical use.

Hence, Instead of a fully dual construction, we instead derive a consumer API in steps analogous to those for deriving the `Producer` API. We start again with the consumers of infinite sequences and the consumers of finite sequences:

```
API InfiniteConsumer<C, I>
    consume: (I, C) -> C
```

```
API FiniteConsumer<C, I>
    consume: (I, C) -> C
    close: C -> C
```

We can again introduce a type parameter for the final sequence item to unify the APIs; observe how using ! or () for the parameter `F` in the following API yields results isomorphic to the `InfiniteConsumer` and `FiniteConsumer` APIs respectively:

```
API Consumer<C, I, F>
    consume: (I, C) -> C
    close: (F, C) -> ()
```

This API is fully symmetric to the `Producer`: the consumer can consume exactly those sequences that a producer can produce, by feeding the final item into *close*. It is rather unusual in that we have never seen an API whose `close` function takes an argument in the wild.

Another unusual aspect is the inability of a consumer to report errors to its calling code. This is severe enough of a departure from typical APIs to warrant a dedicated discussion.

3.2 Communication Flow

The inability to emit errors appears to make our proposed `Consumer` API unsuitable for network programming, for example. The underlying characteristics of the design are more general than just error reporting: code interacting with a *consumer* can pass information *to* the consumer but cannot obtain any information *from* the consumer. Observe that conversely, code interacting with a *producer* can obtain information *from* the producer but cannot pass any information *to* the producer.

This rigorous a restriction on communication flows evokes design choices such as the unidirectional communication primitives of security-focussed micro-kernels like seL4 [MMB+13], so there clearly is a place for such constrained APIs. But the consumer API does not seem appropriate as a general-purpose API.

Trying to add the missing communication flows raises some interesting questions. Should `consumer` return the next consumer *and* another piece of information, or the next consumer *or* another piece of information? What about `close` — should it be able to return extra information, or not? How should symmetry be preserved — does the `Producer` API require a `stop` function that lets the surrounding code communicate that (and why) `produce` will no longer be called? Should `produce` itself take a piece of information as input?

Our fairly principled approach of aiming for minimal, symmetric, regular-languagy APIs provides no guidance here, as these communication flows exist outside our semi-formalized problem domain. Any choices we need to make are essentially arbitrary.

We see two ways out of this problem. The simplemost solution is acceptance. When a programmer wishes to write data to a network through a consumer interface, they need a corresponding producer to emit any feedback such as connection failures. Considering that typical networking APIs use the same error type for reading and writing data, this doesn't seem too far-fetched. Then again, the difficulties in migrating from more typical APIs to this style of error handling are hard to estimate. An argumentative essay like this one cannot conclusively establish a result, we merely want to raise that accepting a consumer API without error reporting might be more feasible than it appears at first glance.

The other solution is to consider fallibility as an *effect*. Just like the functions we use in our APIs might be asynchronous, they might also be fallible. Different programming languages could represent this in different ways: some could use exceptions, others could consistently use a `Result` type (a sum type of either the actual value of interest or an error value) — the latter is a simple and classic example of monadic effect handling. We can keep using the same notation as before, but consider every function as possibly fallible.

Nevertheless, it is instructive to look at the APIs that result from adding explicit error return options (of some type *E*) to all functions:

```
API FallibleConsumer<C, I, F, E>
    consume: (I, C) -> C | E
    close: (F, C) -> () | E

API FallibleProducer<P, I, F, E>
    produce: P -> (I, P) | F | E
```

The APIs look quite asymmetric suddenly, because the `FallibleProducer` does not mirror the communication flow of the consumer, as that would require functions that take *E*s as arguments. Further, the return type of `produce` appears to violate minimality, as *E* and *F* could be combined into a single type parameter in principle. This demonstrates that the perspective of errors as effects is crucial to meaningfully evaluating sequence APIs — both ours, and those in the wild.

We will continue our discussion with the raw `Producer` and `Consumer` APIs, and leave it to the reader to decide whether their functions should be fallible (and/or asynchronous, for that matter), or not.

3.3 Evaluating Our Design

Are our `Producer` and `Consumer` APIs minimal, symmetric, and expressive on the level of (ω-) regular languages?

```
API Producer<P, I, F>
    produce: P -> (I, P) | F

API Consumer<C, I, F>
    consume: (I, C) -> C
    close: (F, C) -> ()
```

Symmetry is not immediately apparent; there is no obvious sense in which the two APIs are dual. We derived the APIs in analogous steps, but that is not an inherent property. And they even have a different number of functions!

Still, we can make a solid argument based on the observation that the APIs *compose* in a satisfying way.

Composing a producer with a consumer amounts to piping the data that the producer produces into the consumer:

```
Require: P, C, I, F are types such that Producer<P, I, F>
  and Consumer<C, I, F>
  procedure PIPE(p : P, c : C): ()
    loop
      x ← PRODUCE(p)
      if x is of type F then
        CLOSE(x, c)
        return ()
      else
        c ← CONSUME(x.0)
        p ← x.1
      end if
    end loop
  end procedure
```

The `pipe` function returns the unit type. On a purely abstract level, composing to the unit type evokes the concept of an element and its inverse composing to an identity element. This seems as strong a formal notion of symmetry (without *actually* formalizing things) we can hope for, aside from immediate duality.

The *close* function taking an argument nicely mirrors the *produce* function emitting a final argument. In particular, if F is another producer, then the consumer can *pipe* it in its *close* implementation into an inner consumer. The overall return type is still () — the unassuming *pipe* function can handle multi-stage processing pipelines wihout any modification.

We can make a similar compositional argument for composing the other way around: it should be possible to create a pair of a consumer and a producer such that the producer produces everything that the consumer consumes (in the same order, i.e., as an in-memory queue). Such a queue is, in some vague sense, the neutral element of transformation steps in a pipeline (we return to this concept in section 4.1). Here, we see another benefit of the `close` function taking an argument: we can map this argument directly to the final value to be emitted by `produce`.

Having argued that our design is indeed symmetric in a meaningful way, we turn to the question of expressivity. Our core argument rests on the observation that each `Producer` (or `Consumer`) defines a formal language over an alphabet of atomic types. More precisely, a `Producer<P, I, F>` emits an arbitrary number of repetitions of values of type I, followed by a single value of type F. In more traditional notation of a language as a set of strings, it denotes the set $\{I\}^* \circ \{F\}$.

Given this mapping from sequence APIs to languages, which class of languages do our APIs describe? We claim they — in concert with sums, products, and functions — describe the union of the regular and the ω-regular languages.

The *ω-regular languages* over Σ are the sets of infinite strings over Σ that are either a concatenation of infinitely many words from the same regular language[8] over Σ (*infinite iteration*), or the concatenation of a regular language and an ω-regular language over Σ, or a choice between finitely many ω-regular languages over Σ.

As already argued in section 1.2, sum types and product tyes correspond to choice and concatenation of regular expressions respectively. Unlike the strict case, we cannot rely on homogeneous arrays to act as the counterpart to the Kleene operator, but this is where the `Producer` API comes in (everything applies analogously for the `Consumer` API): a `Producer<P, I, F>` can produce an arbitrary number of repetitions of `I`s, followed by a single `F`. In particular, a `Producer<P, I, ()>` corresponds to the Kleene operator, and a `Producer<P, I, !>` corresponds to infinite iteration.

Unfortunately, this simple perspective is not fully accurate. Product types as concatenation are too powerful for us: consider a product (P_1, P_2), where P_1 is a `Producer<P, I, !>`. The corresponding language would be a concatenation with an ω-language on the left, but this is explicitly ruled out by the definition of ω-regular languages. Another facet of the same problem is that the type (S, T) is not one that describes *first* emitting an S and *then* a T, as it presents both simultaneously.

To solve this, we can restrict the set of well-formed sequence types we consider to pairs $(S, () \to T)$ for (sums of) non-repeated types S and arbitrary types T, and `Producer<P, S, T>` for repeated types S. This removes the ability to express concatenations with an ω-language as the left operand, and introduces the required indirection to express "first S, then T" (remember that we assume our functions to abstract over effects, so there might well be asynchronicity involved in obtaining the T after processing the S).

We shall not dwell on this subtlety in greater detail, because it does not affect our two main points: our API is expressive enough to decribe regular (ω-) regular languages, whereas a more traditional API *without* a dedicated type for the final item is *not* expressive enough, resulting in an unjustified reduction in expressive power compared to representing strict sequences in memory. In particular, traditional APIs cannot express concatenation of two sequences with different item types.

Finally, our designs are indeed minimal: removing any feature reduces expressivity, because all features are necessary to obtain the correspondence to the (ω-) regular languages.

4 Working With Producers and Consumers

Having settled on designs for `Producer` and `Consumer` APIs, we now turn to how they can or should should be used in practice. We note a powerful pattern of composability in

[8] We assume familiarity with regular languages, for an introduction see [HU69], for example. Or do the sensible thing of searching for "regular language" on Wikipedia.

section 4.1, muse about language-level support in section 4.2, before turning to matters of efficiency in section 4.3.

4.1 Conducers

In section 3.3, we briefly considered an in-memory queue: a pair of a consumer and a producer such that the producer emits exactly the item consumed by the consumer. We can consider such a pair as a single value that implements both the `Consumer` and the `Producer` API; we shall call such a value a *naïve conducer* (portmanteau of *con*sumer and pro*ducer*). The *naïveté* will become apparent once we go from intuitive notions of composability to actual implementation; for now we ask the reader to suspend some disbelieve and let the concept guide us toward the more useful *actual conducers*.

Naïve conducers make an appealing foundation for constructing and composing producers and consumers. You can use a single naïve conducer definition to both obtain a new producer from a producer or a new consumer from a consumer. Consider the naïve queue conducer: composing a producer with the naïve conducer yields a new producer that buffers some number of items before emitting them. Composing the naïve conducer with a consumer yields a new consumer that buffers some number of items before consuming them in the inner consumer.

This dual-purpose usage constitutes a tangible advantage of being hellbent on symmetry. As a second example, consider a naïve conducer constructed from some function of type $S \rightarrow T$ that is a consumer for items of type S and a producer for items of type T. This naïve *map* conducer can both adapt the items emitted by a producer, or adapt the items accepted by a consumer.

Naïve conducers need not preserve a one-to-one mapping between consumed items and produced items. The common tasks of encoding and decoding values for transport can be captured elegantly by naïve conducers: a *decoder* consumes items of some type S (often, S would be the type of bytes) and occasionally produces an item of some type T, an *encoder* consumes items of some type T and produces many items of some type S.

Unfortunately, none of this actually works. In order to, for example, compose a naïve conducer in front of a consumer, the *consume* function of the resulting consumer would have to first call the *consume* function of the naïve conducer. Then, it would need to correctly guess how many times to call the naïve conducer's *produce* function, in order to feed the results to the inner consumer. A general-purpose composition routine can neither know how many items the inner consumer expects, nor how many items the naïve conducer can produce at any point in time.

One obvious solution is to explicitly manage metadata about which functions can and should be called at runtime, but this creates computational overhead. Another simple solution is to restrict naïve conducer to producing exactly one item per item they consume, but this severely restricts expressivity — in particular, it prohibits encoders and decoders.

Toward a zero-overhead, expressive solution, we temporarily abandon the dual-usage intuition behind naïve conducers, and examine consumers and producers separately. We define a *consumer adapter* as a function that maps an arbitrary consumer to another consumer, and a *producer adapter* as a function that maps an arbitrary producer to another producer.

These adapters can implement the same functionality as naïve conducers, but in a way that actually works. Consider, for example, a consumer adapter for encoding items of type S to many items of type T. The consumer adapter can produce a consumer that consumes an item of type S, computes the encoding, and calls the *consume* function of the inner consumer once for each T of the encoding. The corresponding producer adapter, when asked to produce a value of type T, asks the wrapped producer for value of type S, and computes the encoding. It then returns the first T of the encoding and buffers the remaining encoding, to be admitted on subsequent calls to *produce*. Only when the buffer has become empty does it request another item from the wrapped producer.

There is a large amount of overlap and symmetry between the encoding consumer adapter and the encoding producer adapter, note how both use the same procedure for the actual encoding, and both need to buffer the result in between subsequent calls to the wrapped consumer or producer respectively. We call a pair of consumer and producer adapters that implements a naïve conducer an (actual) *conducer*.

While such conducers are an interesting tool to reason about working with lazy sequences, they do not provide an immediate software engineering benefit: the two adapters need to be implemented independently. In the spirit of full symmetry, we now have to duplicate all implementation efforts.

To improve on this, we next take a look at how programming language syntax (or macros) can make it possible to write a single definition that then yields both adapters of a conducer. To do so, we first need to investigate dedicated syntax for producers and consumers separately.

4.2 Syntax Considerations

Many programming languages offer generator syntax for creating iterators, and for loops for consuming iterators. A language designed with our APIs in mind could provide more powerful syntax.

Generators[9] provide dedicated syntax for creating producers, with `yield` emitting repeated items and `return` emitting the final value. As an example, the following pseudo-code emits the numbers from zero to nine and then the final string

[9] https://peps.python.org/pep-0255/

"hi". We use atypical choices of keywords (`producer` instead of `generator`, `produce` instead of `yield`, and `produce final` instead of `return`) to be obnoxiously explicit about the intended semantics, and to prepare for a symmtric consumer design:

```
producer
    i = 0
    while i < 10
        produce i
    produce final "hi"
```

We are not aware of any language that provides a symmetric construction for creating consumers. Dreaming up an initial symmetric design seems straightfoward enough:

```
consumer
    x = consume
    y = consume
until consume final z
    doSomething(x + y + z)
```

This design does leave open some questions: what if the `consume` function of the created consumer is called more often then there are `consume` keywords in the main consumer body? And should it always be valid to jump to the `until consume final` block, or only at the end of the main consumer body?

Since the basic consumer design allows no communication to the calling code, a simple solution to the problem of too many *consume* calls is to implicitly wrap the main consumer body in a loop. In a setting with fallible consumers, a consumer that wants to limit the number of possible calls to *consume* can simply add an extra `consume` expression and throw from there:

```
consumer
    x = consume
    y = consume
    _ = consume
    throw "too␣much␣information"
until consume final z
    doSomething(x + y + z)
```

To allow for control about what to do when *close* is called depending on the current state of the consumer, the naïve `until consume final` can be replaced with a mechanism that mimics `try-catch` blocks:

```
consumer
    consumeblock
        x = consume
    until _
        throw "too␣little␣information"
    consumeblock
        y = consume
    until z
        doSomething(x + y + z)
    consumeblock
        _ = consume
    until _
        throw "too␣much␣information"
```

Our syntax is deliberately painful: we do not claim that these are the best design choices, we merely want to demonstrate that providing a meaningful and useful consumer syntax is indeed possible. And after extrapolating the logic that leads to our API designs, designing generators into languages without a corresponding consumer equivalent feels questionable.

A particular usecase we want to highlight for explicit (asynchronous) `consumer` syntax is that of implementing asynchronous parsers. Typically this involves writing a state-machine or otherwise putting a lot of manual work into ensuring a parser that can suspend its execution when reaching the temporary end of input and then resume once more input becomes available. The `consumer` syntax allows writing asynchronous parsers that look just like synchronous ones.

Assuming the questions around dedicated `consumer` syntax have been solved, the next logical step is to combine the `consumer` and `producer` keywords into a more powerful `conducer` language construct. As an example, we sketch an encoder conducer for converting 16-bit integers into sequences (pairs) of 8-bit integers:

```
conducer
    consumeblock
        x = consume
        produce x / 256
        produce x % 256
    until f
        produce final f
```

From such a construct, both a consumer adapter and a producer adapter can be generated. For the consumer adapter, the `consume` expressions provide the entry points to the state machine of the *consume* function, and each `produce` expression translates to a *consume* call of the wrapped consumer. For the producer adapter, the `produce` expressions provide the entry points to the state machine of the *produce* function, and each `consume` expression translates to a *produce* call of the wrapped producer.

Finally, we want to draw a parallel to coroutines[MI09], as implemented, for example, in Lua[Ier06]. In (that particular brand of) coroutines, the `yield` expression in the coroutine implementation not only yields a value to the outside world, but it also evaluates to a value that is given as part of the expression that resumes the coroutine. We can see our conducer syntax as a generalization of this pattern. Coroutines tie incoming and outgoing communication to the same points in the coroutine, marked by `yield`, whereas our design decouples them via `consume` and `produce`. In fact, Lua's coroutine approach is equivalent to naïve conducers restricted to maintaining a one-to-one correspondence between consumption and production. Our syntax allows arbitrarily splitting the communication. Hence, conducers generalize coroutines.

4.3 Buffering and Bulk Processing

We now turn to questions of efficiency. While consumers and producers make for nice building blocks of programs because they are conceptually simple to reason about, it is inefficient in practice to process items one by one.

One problem of processing items one at a time is that performing side effects is often expensive, for example, when system calls are involved. Writing a file byte by byte with individual system calls is orders of magnitude slower than buffering bytes sequentially in memory and writing many bytes with a single system call.

An easy solution is to allow consumers to buffer items internally, leaving them the freedom to arbitrarily delay actual processing indefinitely to optimize for efficiency. When writing to a consumer in order to perform side-effects, the programmer needs a way to force the consumer to stop delaying, *flush* its buffer, and actually trigger the effects:

```
API BufferedConsumer<C, I, F>
    consume: (I, C) -> C
    close: (F, C) -> ()
    flush: C -> C
```

The buffered consumer with a `flush` function is a staple of real-world APIs. The analogous functionality for producers, however, is one we have never encountered. The opposite of *flushing* as much data as possible *out of* a buffer is *slurping* as much data as possible *into* a buffer.

```
API BufferedProducer<P, I, F>
    produce: P -> (I, P) | F
    slurp: P -> P
```

Unlike flushing a consumer, slurping a producer does not serve to immediately trigger effectful production of items. Still, there are arguments in favor of a slurp function on producers that go beyond the consistency gains of maintaining symmetry (although that alone would already suffice in our opinion). Consider a producer that emits items from some effectful source which might stop working at any moment (e.g., a network connection). Slurping allows the programmer to pre-fetch data even though processing the available data might be time-consuming and not yet finished, thus reducing the probability that a later connection failure leads to data loss.

System calls are not the only reason for processing data in bulk. Simply copying consecutive bytes in memory from one location to another is significantly more efficient than copying each byte individually. Hence, many programming languages offer APIs for producing or consuming many items at a time by way of *slices* (a pointer paired with the number of items stored consecutively in memory starting at the pointed-to address).

A typical example of such *readers* (producers of many bytes simultaneously) and *writers* (consumers of many bytes simultaneously) are the `Reader`[10] and `Writer`[11] abstractions of the Go language. To translate them into pseudo-types, we write `&r[T]` for a slice of values of type `T` that may be read but not written, and `&w[T]` for a slice of values of type `T` that may be written but not read. The Go APIs then translate to the following:

```
API Reader<R, I, E>
    read: (R, &w[I]) -> (R, Nat) | E

API Writer<W, I, E>
    write: (W, &r[I]) -> (W, Nat) | E
```

The `read` function *writes* (produces) some number of items into a slice, and returns how many items were written. The `write` function *reads* (consumes) some number of items from a slice, and returns how many items were read. A return value of zero typically indicates the end of the sequence[12]. We can easily generalize to arbitrary final values of some type F by requiring the returned number to be non-zero and extending the return sum type by a third[13] option of type F.

Setting aside the interesting naming choices and the fact that most langages unnecessarily specialize the item type to that of 8-bit integers, these APIs display a perfect symmetry that APIs for operating on individual items usually lack.

It is tempting to think of readers and writers as *generalizations* of producers and consumers respectively, but that viewpoint brings a problematic amount of freedom — which parts should be generalized, and which parts should stay the same? Consider, for example, our restrictions to exclusively reading or writing from slices. This is more restrictive than allowing arbitrary access to the slices, and, given the defaults of programming languages (no mainstream languages support write-only pointers), the default choice of many is unrestricted access to the slices. The Rust community has had to put a lot of energy into dealing with the consequences of such an oversight in its standard library[14].

Instead, we propose to think about readers and writers as optimization details: any *read* must be equivalent to a

[10] https://pkg.go.dev/io#Reader

[11] https://pkg.go.dev/io#Writer

[12] In a synchronous setting, if no data is currently available but there might be more data in the future, the functions should block instead of returning zero. In an asynchronous setting, the functions should be parked to be resumed at a later point.

[13] Or a *second* option, if we consider the error case as an effect.

[14] Rust allows for uninitialized memory, but *reading* from unitialized memory is unsafe. See https://github.com/rust-lang/rfcs/blob/master/text/2930-read-buf.md and https://blog.yoshuawuyts.com/uninit-read-write/ for details on how this affects its reader API.

series of zero or more calls to *produce*, and any *write* must be equivalent to a series of zero or more calls to *consume*. This viewpoint precisely defines the semantics of the reader and writer APIs, and cleanly specifies answers to questions that might otherwise be non-obvious: may *read* access the contents of the slice? No. What should *read* or *write* do when given an empty slice? Nothing. Is every (buffered) reader or writer a (buffered) producer or consumer respectively? Absolutely.

This last question is crucial: readers are subtypes of producers, and writers are subtypes of consumers. If you take away only one point from this essay, this is the one.

Readers and writers stem from file system abstractions, the duality of reading and writing to or from a file make their symmetry an obvious requirement. Streams and sinks trace back to iterators, which arose from traversal of (polymorphic) data structures, hence making the genericity of items an obvious requirement. If programming languages had routinely linked the two abstractions by a subtyping relation, we could have had fully symmetric, fully generic, unified APIs for decades. Instead, these abstractions have remained incomplete, and, consequently, interoperate badly.

One problem with the reader and writer APIs is that they do not compose very nicely: in order to move data from a reader to a writer, you need to specifically allocate an array *into* which to first copy the data via *read*, and *from* which to then copy the data via *write*. An alternate API choice without this problem is to *expose* slices of *internal* buffers instead of *processing* slices of *external* buffers:

```
API BulkProducer<P, I, F>
extends BufferedProducer<P, I, F>
    producer_slots: (P) -> &r[I] | F
    process_produced: (P, Nat) -> P

API BulkConsumer<C, I, F>
extends BufferedConsumer<C, I, F>
    consumer_slots: (C) -> &w[I]
    process_consumed: (P, Nat) -> P
# To close, use the BufferedConsumer
# close function
```

The *consumer_slots* function provides a slice into an inner buffer of a `BulkConsumer`, into which the calling code can write. To trigger actual processing of the written items, the *proces_consumed* function notifies the consumer how many items were written and tasks it to consume them. The semantics of calling *process_consumed* with some argument n must be those of calling *consume* n times, with the items written to the slice returned by *consumer_slots*. The `BulkProducer` API works analogously.

Whereas a writer API requires the data to be *consumed* to be in an array, the bulk consumer is required to organize its *internal buffer* as an array. In practice, things are most efficient if *both* sides of the exchange store data consecutively in memory, so we don't expect this shift in responsibility to make a difference to anyone who uses bulk processing for efficiency reasons.

Our APIs are more low-level than the traditional reader and writer APIs: The traditional *read* and *write* functions — we propose to call them *bulk_produce* and *bulk_consume* — can easily be implemented as helper functions that take a slice and copy from or into (respectively) the slots exposed by the bulk API.

Given such *bulk_produce* and *bulk_consume* functions, there are now two semantically equivalent ways of piping a bulk producer into a bulk consumer: *pipe_bulk_consume* uses the *producer_slots* of the producer as the slice argument to *bulk_consume* on the consumer, and *pipe_bulk_produce* uses the *consumer_slots* of the consumer as the slice argument to *bulk_produce* on the producer. Neither of these requires allocation of an external buffer to facilitate the communication.

A final, interesting observation on this topic concerns memory safety. In a language with a concept of uninitialized memory that is acceptable to write to but not to read from, a bulk consumer is free to expose a (write-only) slice of uninitialized memory in its *consumer_slots* function. Whenever *process_consumed* is called thereafter, the consumer can assume that the memory for the indicated number of items has been initialized. If the calling code is faulty, this can lead to undefined behavior, making the *process_consumed* function *unsafe* in the Rust sense of the word, i.e., it can trigger undefined behavior when its contract is not upheld. There is no such problem with the bulk producer API. Thankfully, the *bulk_consume* helper function fully insulates from this source of errors.

5 Summary

This concludes our main arguments and designs. Figure 1 lists our final APIs. Our main points of departure from current mainstream designs are the following:

- Full symmetry between producers and consumers.
- Equivalent APIs irrespective of effects such as asynchrony or fallability.
- A dedicated type for the last sequence item, drastically increasing expressivity.
- Slurping producers.
- Bulk processing for items other than raw bytes.
- Subtyping relation between bulk processors and regular processors.
- Zero-copy bulk APIs.
- Dedicated consumer syntax as a counterpart to generators.
- Conducer syntax to automatically derive adapters for both consumers and producers simultaneously.

```
API Producer<P, I, F>
    produce: P -> (I, P) | F

API BufferedProducer<P, I, F>
extends Producer<P, I, F>
    slurp: P -> P

API BulkProducer<P, I, F>
extends BufferedProducer<P, I, F>
    producer_slots: (P) -> &r[I] | F
    process_produced: (P, Nat) -> P
```

```
API Consumer<C, I, F>
    consume: (I, C) -> C
    close: (F, C) -> ()

API BufferedConsumer<C, I, F>
extends Consumer<C, I, F>
    flush: C -> C

API BulkConsumer<C, I, F>
extends BufferedConsumer<C, I, F>
    consumer_slots: (C) -> &w[I]
    process_consumed: (P, Nat) -> P
```

Figure 1. Our API designs in a single place.

6 Onward!

We have proposed and argued for some simple designs, but there is still plenty of engineering and research left to be done.

What is up with conducers? Is the introduction of dedicated syntax really the best way of deriving consumer *and* producer adapters from a single specification? Is there a nicer API design that captures the same degree of composability without requiring this split? If dedicated syntax is the way to go, should there be dedicated syntax for bulk producers, bulk consumers, and bulk conducers? What would it look like? What about vectored I/O[15]?

Concerning the dedicated syntax, we took a lot of shortcuts, not least of all the deliberately horrible syntax for consumers. On the more formal side, what should be the proper — say, denotational — semantics of a `conducer` syntactic element be? Given such formal semantics, what is a translation of the syntax into "normal" syntactic components of equivalent semantics? Which "normal" constructs are particularly helpful — coroutines, continuations? Can you elegantly avoid such fancy constructs altogether?

Is the fact that conducers generalize coroutines a coincidence, or do conducers deserve study as a control-flow mechanism in their own right? Coroutines are as expressive as one-shot continuations, but strictly less expressive than general continuations [MI09]. Where do conducers fall in this spectrum?

What is up with the symmetry between producers and consumers? Is there a general, formal setting for expressing APIs with a general, precise notion of duality, in which producers and consumers are dual in a formal sense? Did we simply not find it yet, or is this impossible? For infinite, homogeneous sequences, producers and consumers are actually dual. Why, and where *exactly* do things go wrong when adding dedicated final elements or effects such as irrecoverable errors?

How far can we take our unsatisfying substitutes for proper duality — symmetry and inverse-like composition? There is plenty of literature on proving iterators correct, see [BHMS22] for a recent example. How much of such literature carries over to consumers, and how much has to be redeveloped from scratch? This question should serve as a powerful motivation for finding a framing in which producers and consumers are fully dual. Similar thoughts apply to optimization techniques [KBPS17] or code synthesis[RML+12].

Session types [DCD10][HLV+16] aim to statically type communication patterns in a way that guarantees, for example, deadlock-freedom. Our explicit final item type allows us to also accurately type certain classes of communication patterns. How much overlap is there between our work and session types, can they benefit from each other?

Regarding more direct concerns of software engineering, which adaptors or combinators should make up the standard toolbox for composing sequence APIs? Which algebraic laws must they fulfil? What is a good technique for implementing combinators only once and then automatically deriving bulk versions? Conducers provide a good framing for unary combinators, but what about other combinators (say, a binary concatenation combinator)?

Producers and consumers strictly limit *where* they interact with a sequence. Aside from optimization details such as functions for providing estimates of the minimum and maximum number of items that can still be processed, the most obvious extension of our APIs is that of random-access. Readers and writers originate from the Unix notion of *files*, and *seeking* in a file is a core concern of this perspective. What do good APIs for seeking look like? Support for infinity sequences mandates relative offsets rather than absolute indexing. Does this mean that all such generalizations amount to Turing-machine APIs with a movable read/write head? Should writing do overwrites exclusively, or is there

15 https://en.wikipedia.org/wiki/Vectored_I/O

design space for elastic bands that support insertion of new items in-between older items (as well as proper deletion)? Can and should these two modes be captured in the same API, or do they require separate abstractions? What does a lattice of (sub-) APIs look like that provides a more nuanced yet practically useful version of "everything is a file"?

Another avenue for generalization is provided by the expressive power of the APIs. Our producers and consumers correspond to the (ω-) regular languages. Are there elegant APIs that capture the context-free languages? If you squint a lot, (sets of) producer types look quite similar to left-regular grammars — which should not be too surprising, given the relation with regular languages. What is the formal version of "squinting a lot"? Does it have an inverse? Which computational interpretation do you obtain by "unsquinting", say, the grammars in Chomsky normal form?

Yet another (arguably more practically relevant) generalization is from sequences to other graphs. What are appropriate APIs for consuming or producing trees? How do different traversal orders (breadth-first, depth-first, etc) factor into the API designs? What about APIs for exploring only a single path through a tree? Will there be a link between APIs for tree processing and grammars of context-free languages? How far can we take sensible APIs for traversing more complex graphs like DAGs or even arbitrary digraphs?

Finally, APIs with support for seeking in sequences or more complex graphs open up the question of *who* performs the seeking. In a traditional file system API for seeking in and reading from a file, it is the user code that invokes the seeking. But consider instead a texteditor that feeds changes to a text buffer to some plugin. Here, the user code (i.e., the plugin) reads data, but it does not control *where* in the sequence it reads. The same kind of inverted seeking can happen for more complex data structures: a text editor might update a plugin about changes to a (higher-order) syntax tree, for example. We are not aware of any principled investigation into such APIs.

7 Further Reading

In this final section, we want to share some references that could be of interest to anyone wishing to pursue those open questions or to implement a library of sequence abstractions.

We have primarily presented our API designs by deriving them from first principles, instead of relating them to existing designs. While there are plenty of languages and libraries to choose from for documentation of existing APIs, there is much less available material on the reasoning behind those APIs. A notable exception are Oleg Kiselyov's *iteratees* [Kis12] and the resulting streamlined and well-documented `iterIO` Haskell library[16]. Their expressivity and rich algebraic structures are remarkable, as is the viewpoint of iteratees as communicating sequential processes. Yet, the design differs significantly from ours, the inherent asymmetry is striking: enumerators and iteratees are not at all dual. Particularly interesting is the notion of `Inums` in the `iterIO` library: they fulfil the same role as our naïve conducers, while being completely asymmetric (and hence avoiding the problems that require us to move from naïve to actual conducers).

Kiselyov's treasure trove of a website[17] contains several[18] collections[19] of writing[20] that pertain to sequence APIs. The writing focuses almost exclusively on producers, with barely a word on consumers or any notions of symmetry or duality. We find it quite exciting that there is such a deep take on the same material that reaches such different conclusions.

Functional reactive programming (FRP) is concerned with APIs for building systems on event streams, a good overview is given in [PBN16]. Whereas a sequence can be interpreted as a value evolving over discrete timesteps, FRP tackles the challenges of building abstractions (and efficient implementations) for values varying over a continuous notion of time. Discussion of FRP invariably turns to restricting the treatment of time to that of discrete event steps; this notion of FRP is all about what we called producers, discussing efficient implementation techniques, adapters, and combinators. A prominent example of this brand of FRP is the Elm language [CC13]. Appendix A contains a dozen popular javascript libraries for such FRP.

FRP stands on the shoulders of stream processing. An instructive survey by Stephens [Ste97] provides a good introduction. Like us, Stephens laments the lack of a unified theory underlying disparate API design efforts. The theory that Stephens then proposes is a mathematical one rather than one of API designs.

The implementation of iterators (and hence, producers and the symmetric consumers) in imperative langages is typically a highly stateful business. In many cases, particularly when no side-effects are involved, there exist purely functional alternatives [Bak93]. Gibbons and Oliveira [GO09] give a particularly thorough account that incorporates effect handling in a functional setting. The reader who has not spent years obtaining intimite familiarity with the Haskell standard library should be warned that reading this paper is a lot like reading the Silmarillion, in that a startling fraction of words past the introduction are made-up.

In discussing algebraic datatypes together with homogeneous array types as a representation for strict sequences in memory, we glossed over the fact that such representations do not allow numeric indexing. Such representations are also possible, even while maintaining static typing [KLS04]. The degree to which the strictly limited access provided by

[16] https://hackage.haskell.org/package/iterIO-0.2.2/docs/Data-IterIO.html

[17] https://okmij.org/ftp/

[18] https://okmij.org/ftp/Haskell/Iteratee/index.html

[19] https://okmij.org/ftp/Streams.html

[20] https://okmij.org/ftp/Scheme/enumerators-callcc.html

producers and consumers simplifies typing compared to a random-access collection is remarkable.

We finish with a few pieces of literature on iterators that arguably did not stand the test of time, but which provide some creative input to the design space.

Interruptible Iterators [LKM06] provide an alternative to generator syntax for implementing iterators. *Interrupts* aim to allow for easy implementation of internal state changes between or during iteration steps.

Segmented iterators [Aus00] address efficiency concerns when working with segmented data structures such as hash tables that consist of several, disjoint arrays of items.

Iterators in the *swapping paradigm* [WEHL94] tackle difficulties in formally verifying properties of iterators. They reimagine iterator for programming laguages that do not *copy* values by default, but *swap* them instead. This programming model anticipates the linear-type-like move semantics of languages like Rust.

References

[AP21] Danel Ahman and Matija Pretnar. Asynchronous effects. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[Aus00] Matthew H Austern. Segmented iterators and hierarchical algorithms. In *Generic Programming: International Seminar on Generic Programming Dagstuhl Castle, Germany, April 27–May 1, 1998 Selected Papers*, pages 80–90. Springer, 2000.

[Bak93] Henry G Baker. Iterators: Signs of weakness in object-oriented languages. *ACM SIGPLAN OOPS Messenger*, 4(3):18–25, 1993.

[BHMS22] Aurel Bílý, Jonas Hansen, Peter Müller, and Alexander J Summers. Compositional reasoning for side-effectful iterators and iterator adapters. *arXiv preprint arXiv:2210.09857*, 2022.

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. *ACM SIGPLAN Notices*, 48(6):411–422, 2013.

[DCD10] Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. Sessions and session types: An overview. In *Web Services and Formal Methods: 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers 6*, pages 1–28. Springer, 2010.

[GO09] Jeremy Gibbons and Bruno C d S Oliveira. The essence of the iterator pattern. *Journal of functional programming*, 19(3-4):377–402, 2009.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, et al. Foundations of session types and behavioural contracts. *ACM Computing Surveys (CSUR)*, 49(1):1–36, 2016.

[HU69] John E Hopcroft and Jeffrey D Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[Ier06] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[KBPS17] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 285–299, 2017.

[Kis12] Oleg Kiselyov. Iteratees. In *International Symposium on Functional and Logic Programming*, pages 166–181. Springer, 2012.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 96–107, 2004.

[Lei17] Daan Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, pages 16–29, 2017.

[LH18] Erick Lavoie and Laurie Hendren. A formalization for specifying and implementing correct pull-stream modules. *arXiv preprint arXiv:1801.06144*, 2018.

[LKM06] Jed Liu, Aaron Kimball, and Andrew C Myers. Interruptible iterators. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 283–294, 2006.

[MI09] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.

[MMB+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429. IEEE, 2013.

[PBN16] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. *ACM SIGPLAN Notices*, 51(12):33–44, 2016.

[RML+12] Derek Rayside, Vajihollah Montaghami, Francesca Leung, Albert Yuen, Kevin Xu, and Daniel Jackson. Synthesizing iterators from abstraction functions. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, pages 31–40, 2012.

[Ste97] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34:491–541, 1997.

[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.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

[WEHL94] Bruce W. Weide, Stephen H. Edwards, Douglas E. Harms, and David Alex Lamb. Design and specification of iterators using the swapping paradigm. *IEEE Transactions on Software Engineering*, 20(8):631–643, 1994.

[ZBL20] Tian Zhao, Adam Berger, and Yonglun Li. Asynchronous monad for reactive iot programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 25–37, 2020.

A Appendix: 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
- 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

All 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

Of these roughly 30 competing designs, the pull-streams API is the only one for which we are aware of any academic treatment [LH18].