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, 3 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.

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 orthogonality, symmetry, and expressivity.

Orthogonality asks that no aspect of the API design can be expressed through other aspects of the design. The design must be minimal in that regard.

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, orthogonality 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 ω-regular languages — making us quite confident about the appropriate level of expressivity.

1.2 Case Study: Strict Sequences

Finite case: isomorphism to regular expressions.

2 Related Work

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

3 Evaluating Abstractions

Notation for (the core design behind) APIs.

Disclaimer on choosing the APIs to present here.

Iterator: js and rust.

Co-Iterator: nobody? Java string builder, rust fmt, clojure sequence. rust for-loop.

Stream: rust, java reactivex. Recoverable vs irrecoverable errors.

Sink: lack of symmetry.

4 Deriving Our Design

5 Evaluating Our Design

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

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

Lengths and limits.

Asynchrony.

Buffered.

Slices.

Trees. DAGs? Digraphs?

Coroutines/generators generalized.

Turing machines.

Elastic Turing machines.

Outside movement.

8 Conclusion

This is the conclusion [MTZ03].

On a meta note, this paper provides 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 (discreetly) 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

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

[MTZ03] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.