How Rust views tradeoffs

Steve Klabnik • 03.04.2019

Overview

What is a tradeoff?

“Bending the Curve”

Design is about values

Case Studies

- BDFL vs Design By Committee
- Stability Without Stagnation
- Acceptable levels of complexity
- Green vs System threads

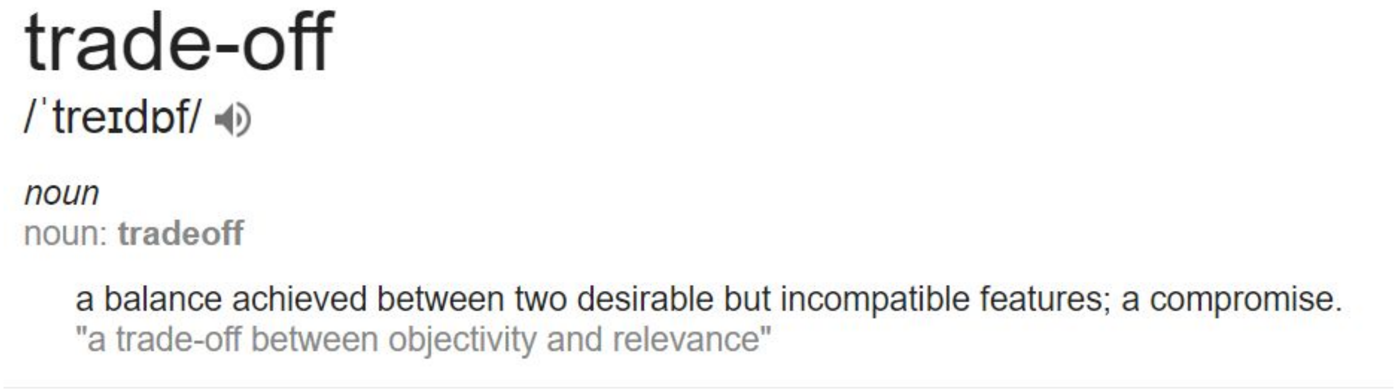
Thanks, Bryan!

“Platform as a Reflection of Values”

Node Summit 2017

What is a tradeoff?

What is a tradeoff?



trade-off

/ˈtreɪdɒf/

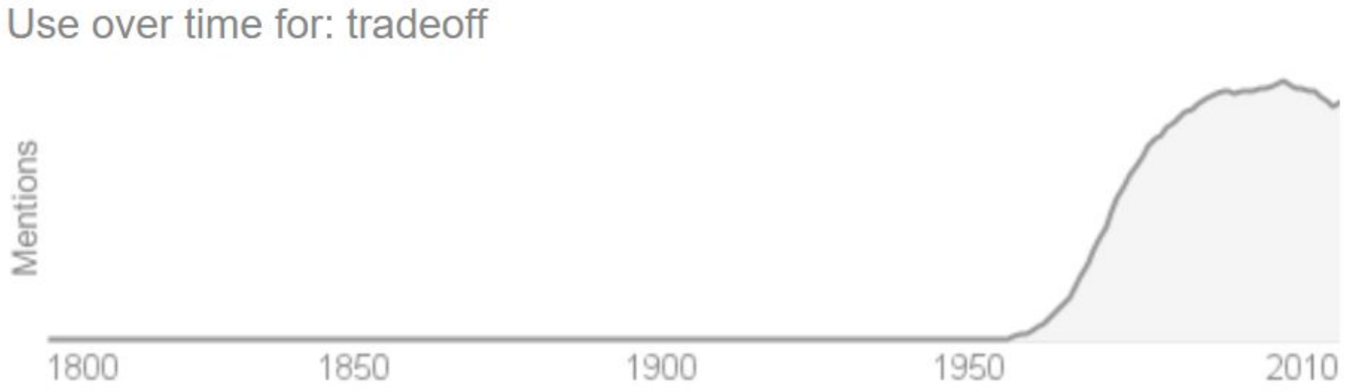
noun

noun: **tradeoff**

a balance achieved between two desirable but incompatible features; a compromise.

"a trade-off between objectivity and relevance"

What is a tradeoff?



What is a tradeoff?

Space vs time

You can make it fast, **or** you can make it small

Throughput vs Latency

Two different measurements, often at odds

Dynamic Types vs Static Types

Ruby, or Haskell?

“Bending the curve”

Bending the Curve

One of these things is not like the others:

- Space vs Time
- Throughput vs Latency
- Dynamic Types vs Static Types

Bending the Curve

One of these things is not like the others:

- Space vs Time
- **Throughput vs Latency**
- Dynamic Types vs Static Types

Bending the Curve

A trade-off (or tradeoff) is a situational decision that involves diminishing or losing one quality, quantity or property of a set or design in return for gains in other aspects. In simple terms, a tradeoff is where one thing increases and another must decrease. -Wikipedia

Bending the Curve

Space vs time

Sometimes, smaller **is** faster

Dynamic Types vs Static Types

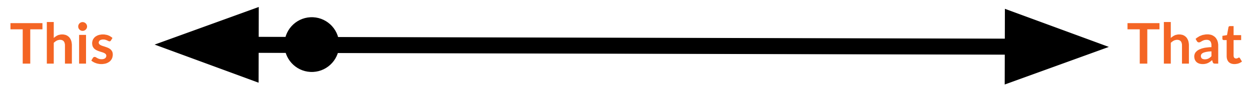
Gradual typing, anyone? (Don’t say “unityped”...)

Throughput vs Latency

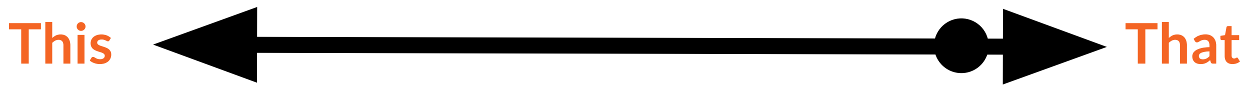
Not **inherently** against each other, though they often are

Bending the Curve

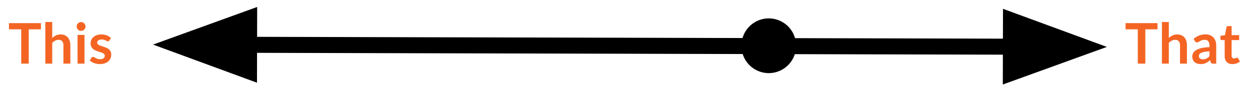
Bending the Curve



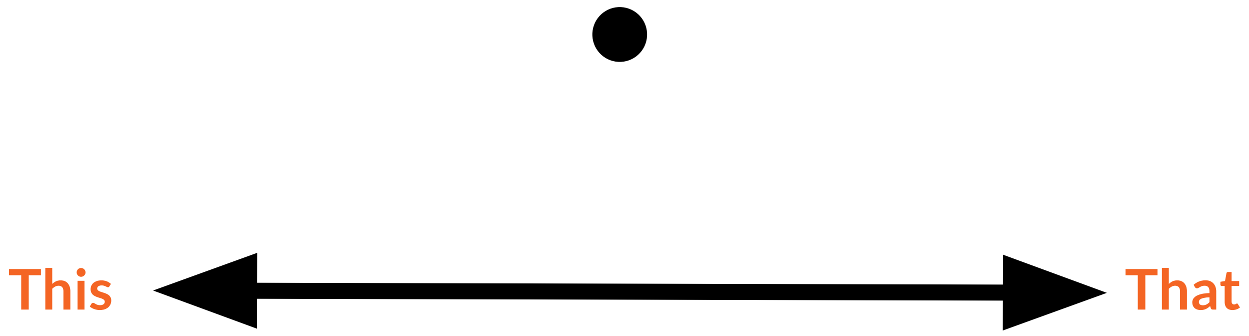
Bending the Curve



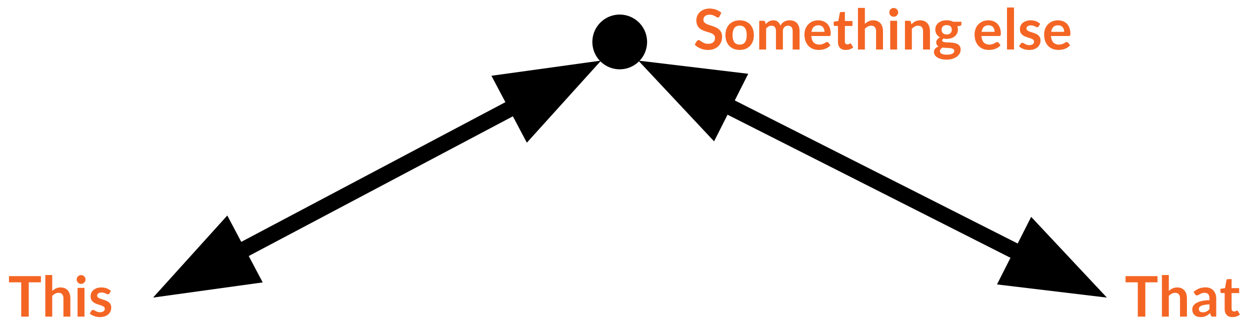
Bending the Curve



Bending the Curve



Bending the Curve



Bending the Curve

A trade-off (or tradeoff) is a situational decision that involves diminishing or losing one quality, quantity or property of a set or design in return for gains in other aspects. In simple terms, a tradeoff is where **one thing increases and another must decrease**. -Wikipedia

Bending the Curve

In game theory and economic theory, a **zero-sum game** is a mathematical representation of a situation in which each participant's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants. If the total gains of the participants are added up and the total losses are subtracted, they will sum to zero. - Wikipedia

Bending the Curve

A **win–win game** is game theory which is designed in a way that all participants can profit from the game in one way or the other. In conflict resolution, a **win–win strategy** is a collaborative strategy and conflict resolution process that aims to accommodate all participants. - Wikipedia

Bending the Curve

When designing Rust, we ask ourselves:

Is this tradeoff fundamental, or through some creativity, can we bend the curve into a win-win situation?

The answer isn’t always yes, but it is yes surprisingly often.

Design is about Values

Design is about Values

What are your core values?

What things do you **refuse** to compromise on?

What are your secondary values?

What stuff would you *like* to have, but don’t *need* to have?

What do you not care about?

What things do others care about that you couldn’t care less for?

Design is about Values

Rust's core values:

In a very particular order:

- Memory Safety
- Speed
- Productivity

Design is about Values

Rust’s secondary values:

- Ergonomics
- Compile times
- Correctness

Design is about Values

Rust doesn’t care about:

- Blazing a new trail in PLT
- Worse is Better
- Supporting certain kinds of old hardware

Design is about Values

Users have values too!

As a developer, you should do some introspection!

Use the tools that align with your values

Otherwise, you’re going to have a bad time

A mismatch causes problems

A lot of internet arguments are really about differing values

Design is about Values

So when you should use Rust?

Rust is ideal when you need a system that’s both reliable and performant.

Sometimes, you don’t need those things! But sometimes, you do.

Sometimes, you don’t need those things at first, but then you do later. We’ll leave the light on for you.

Case Study:
BDFL vs Design by Committee

BDFL vs Design by Committee

Benevolent Dictator For Life

Rules over projects with a velvet fist.

Design by Committee

“A camel is a horse designed by committee”

Can we do things differently?

BDFL vs Design by Committee

The Graydon years

Someone has to start this thing.

The Core Team years

More than one person should be in charge

The RFC Process + subteams

More than one team should be in charge

BDFL vs Design by Committee

This is a problem of scale

As things grow, you hit limits.

Only one team held us back

The Core Team was a bottleneck

New solutions come with new problems

More people and more distributed-ness means less cohesion

Case Study:
Stability Without Stagnation

Stability Without Stagnation

Stability

Things don’t change

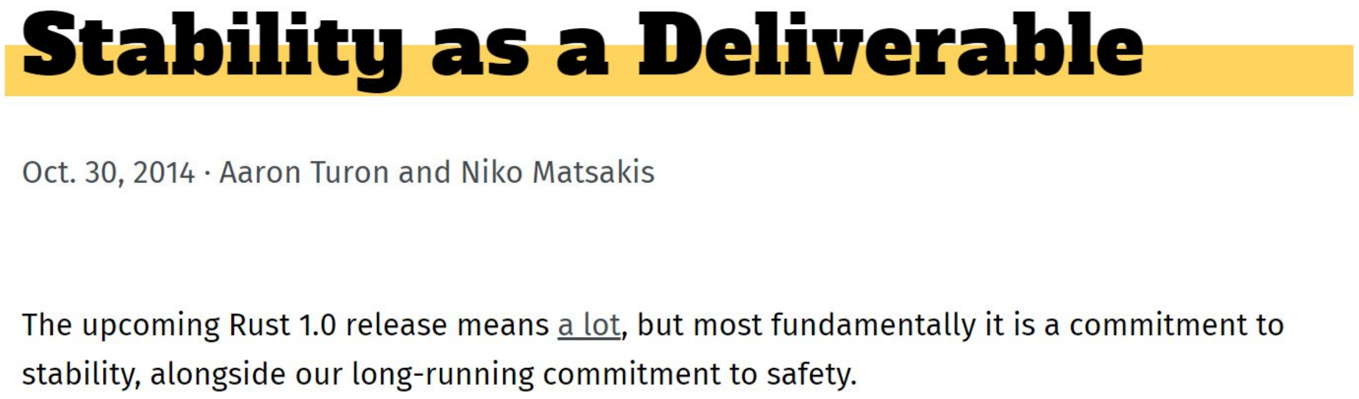
Stagnation

Without change, growth is **hard**

Is this tradeoff fundamental?

We don’t believe so!

Stability Without Stagnation



Stability as a Deliverable

Oct. 30, 2014 · Aaron Turon and Niko Matsakis

The upcoming Rust 1.0 release means a lot, but most fundamentally it is a commitment to stability, alongside our long-running commitment to safety.

https://blog.rust-lang.org/2014/10/30/Stability.html

Stability Without Stagnation

Stability

It's important to be clear about what we mean by stable. We don't mean that Rust will stop evolving. We will release new versions of Rust on a regular, frequent basis, and we hope that people will upgrade just as regularly. But for that to happen, those upgrades need to be painless.

To put it simply, our responsibility is to ensure that you never dread upgrading Rust. If your code compiles on Rust stable 1.0, it should compile with Rust stable 1.x with a minimum of hassle.

Stability Without Stagnation

The Plan

We will use a variation of the train model, first introduced in web browsers and now widely used to provide stability without stagnation:

- New work lands directly in the master branch.
- Each day, the last successful build from master becomes the new nightly release.
- Every six weeks, a beta branch is created from the current state of master, and the previous beta is promoted to be the new stable release.
- In short, there are three release channels -- nightly, beta, and stable -- with regular, frequent promotions from one channel to the next.

Stability Without Stagnation

The Plan

New features and new APIs will be flagged as unstable via feature gates and stability attributes respectively. Unstable features and standard library APIs will only be available on the nightly branch, and only if you explicitly "opt in" to the instability.

The beta and stable releases, on the other hand, will only include features and APIs deemed *stable*, which represents a commitment to avoid breaking code that uses those features or APIs.

Stability Without Stagnation

This is a lot of work!

It’s worth it, though

We have a lot of bots to help

Without change, growth is **hard**

The tradeoff here is us vs users

We spend more time so our users don’t have to

Case Study:
Acceptable Levels of Complexity

Acceptable Complexity Level

Inherent complexity

Some kinds of complexity just exist

Incidental complexity

Some kinds of complexity are your fault

Different designs are differently complex

Fundamental choices can add or remove inherent complexity

Acceptable Complexity Level

Rust’s values mean large inherent complexity

7. It is easier to write an incorrect program than understand a correct one.
8. A programming language is low level when its programs require attention to the irrelevant.
23. To understand a program you must become both the machine and the program.

Epigrams in Programming - Alan Perlis

Acceptable Complexity Level

Rust’s values mean large inherent complexity

Rust wants to help you write correct software. This means that a lot of error handling gets exposed. It’s harder to handle more errors than fewer errors, and it’s harder to handle Result<T, E> than ignore an exception.

Things like the ? operator help reduce this, but it’s still present.

Acceptable Complexity Level

Rust's values mean large inherent complexity

One way in which Rust provides safety and speed at the same time is by using a strong static type system. Types are checked at compile time, but free at runtime!

Strong static type systems have a lot of complexity.

Acceptable Complexity Level

Rust’s values mean large inherent complexity

Rust has a strong commitment to performance. This means that we cannot rely on a garbage collector, and we have to expose, or at least give access to, low-level details of the machine.

This inherently has more complexity than a language that doesn’t prioritize these things.

Case Study:
Green vs System Threads

Green vs System threads

Two competing models

We won’t get *too* in the weeds here

System threads

An API offered by your operating system

Green threads

An API offered by your runtime; N system threads run M “green” ones

Green vs System threads

Two competing models

We won’t get *too* in the weeds here

System threads

An API offered by your operating system

Green threads

An API offered by your runtime; N system threads run M “green” ones

Green vs System threads

System Threads

- Requires calling into the kernel
- Fixed stack size (8MB default on x86_64 Linux)

Green Threads

- No system calls
- Much smaller stack size (8kb default for a goroutine)

Green vs System threads

Sometimes, values change over time

Rust was pre-1.0 for five years

Originally Rust had a runtime

This runtime provided green threads

Rust got lower-level over time

Were green threads still the right choice?

Green vs System threads

System Threads

- Shouldn’t a systems language provide access to the system API?

Green Threads

- Different stack means you have to switch when calling into C
- This creates overhead unacceptable for a performance-first systems language

Green vs System threads

What about a unified API?

Could we bend the curve by having both?

libnative vs libgreen

Two implementations of one API; pick the one you want

The downsides of both and advantage of neither

Were green threads still the right choice?

Green vs System threads

Forced co-evolution. With today's design, the green and native threading models must provide the same I/O API at all times. But there is functionality that is only appropriate or efficient in one of the threading models.

Overhead. The current Rust model allows runtime mixtures of the green and native models. Unfortunately, this flexibility has several downsides: *Binary sizes*, *Task-local storage*, *Allocation and dynamic dispatch*.

Problematic I/O interactions. As the documentation for libgreen explains, only some I/O and synchronization methods work seamlessly across native and green tasks.

Embedding Rust. When embedding Rust code into other contexts -- whether calling from C code or embedding in high-level languages -- there is a fair amount of setup needed

Maintenance burden. Finally, libstd is made somewhat more complex by providing such a flexible threading model

https://github.com/rust-lang/rfcs/blob/master/text/0230-remove-runtime.md

Green vs System threads

As values change, so do the answers

Neither side of a tradeoff is illegitimate

Take time to re-evaluate your values

Have your values changed since this decision was made?

In the end, this API was removed

Rust provides only system threads in its standard library

Green vs System threads

No runtime means bring your own!

Not everything has to be provided by the language

Need green threads? There are packages!

Rayon for CPU-bound tasks, Tokio for IO-bound tasks

There's tradeoffs here too

Fragmentation is a real possibility; the community needs to pick a winner and rally around it for this strategy to work well

Thanks! <3

1. Tradeoffs are inherent
2. Think outside the box
3. Rust shines when robustness and speed are paramount