ICU4X 1.0: Bringing Internationalization to Rust and Beyond

Shane Carr (sffc@google.com)

Presenter Introduction

Shane F. Carr

Shane is the chair of the ICU4X subcommittee in the Unicode Consortium and lead for Google's efforts to build next-generation internationalization solutions. Shane is also chair of the ECMA-402 task group for ECMAScript internationalization.

Agenda

01 Internationalization

What is i18n and why is it important?

02 ICU4X Overview

How did the project come about? What are the goals?

03 Data Management

How ICU4X is, at its core, a pipeline for locale data.

04 Modularity

How did we make ICU4X lightweight?

05 Learnings

What can be learned from building a core library for the Rust ecosystem?

06 Closing Thoughts

Main takeaways from this presentation.

Internationalization

What is i18n and why is it important?

INTERNATIONALIZATION

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

I18N

i18n Is More Than Globalization

Non-English Languages in the U.S.

In 2019, approximately 78% **(241 million)** of all 308.8 million people ages five and older reported speaking only English at home regardless of their nativity. The remaining 22% **(67.8 million)** reported speaking a language other than English at home.

Based on this data, Mandarin and Cantonese were the most common non-English, non-Spanish languages spoken in the U.S., with more than **3.4 million** speakers across the country.

Here is a list of the most common languages spoken at home in the U.S., outside of English:

Language	Population Estimate	Share of Foreign Language Speakers
Spanish	41,757,000	61.6%
Cantonese and Mandarin	3,495,000	5.2%
Tagalog	1,764,000	2.6%
Vietnamese	1,571,000	2.3%
Arabic	1,260,000	1.9%
French and Louisiana French	1,172,000	1.7%
Korean	1,075,000	1.6%
Russian	941,000	1.4%
Haitian Creole	925,000	1.4%
German	895,000	1.3%

i18n Is More Than Translation

Are the lights currently “on” or “off” at this office in Japan?

i18n Is Deep

- Every language, region, and culture has its own unique challenges
- What works for one locale may not work for another
- Always use official i18n solutions

Evolving Needs for International Software

- More Users in More Languages
- Smaller Devices
- Client-Side / Edge Computing

ICU4X Overview

How did the project come about?

What are the goals?

ICU4X is a modular internationalization library in Rust

ICU4X is a modular internationalization library in Rust

- ICU = International Components for Unicode
- ICU4J & ICU4C (aka libicu) are industry standards, also maintained by Google i18n and others

ICU4X is a modular internationalization library in Rust

- Formatting of dates, times, and numbers
- Grammatical feature selection (plurals)
- Unicode text processing (properties, segmentation, collation)
- Time zones & non-gregorian calendars
- ...and more!

ICU4X is a modular internationalization library in Rust

- Small code size & low memory usage
- Only include what you need
- Pluggable data
- Usable on embedded systems!

ICU4X is a modular internationalization library in Rust

- Extensible FFI support via Diplomat
- C++, C, and TypeScript/JS/WASM APIs
- Future: Dart, Java, ..?

ICU4X is a modular internationalization library in Rust

- Safe by default
- Blazing fast zero-copy deserialization
- But usable from other languages!
- Benefit from Rust's ecosystem

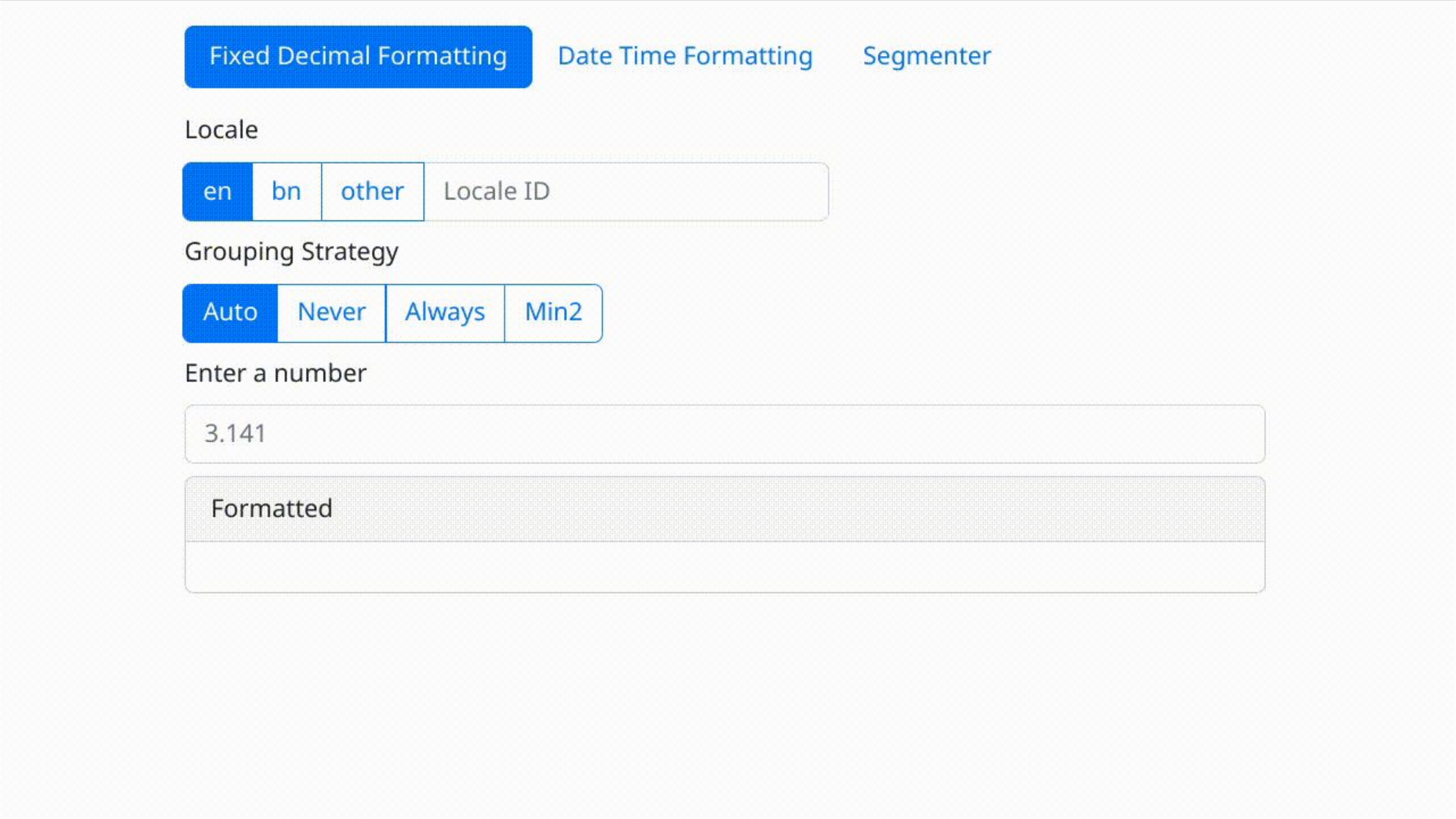
Key Value Proposition

ICU4X is:

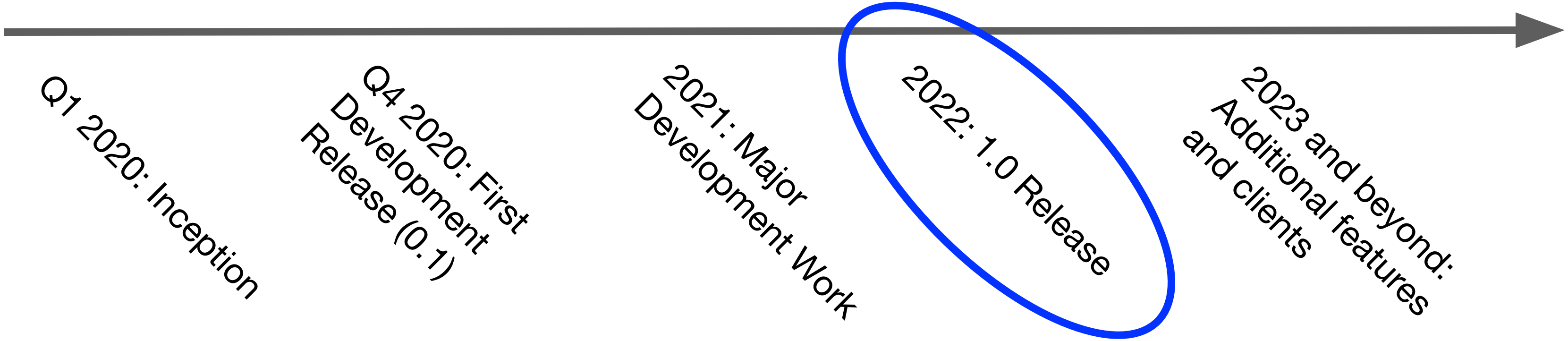
- ❖ Lightweight
- ❖ Portable
- ❖ Secure

Who are we?

ICU4X WebAssembly Demo



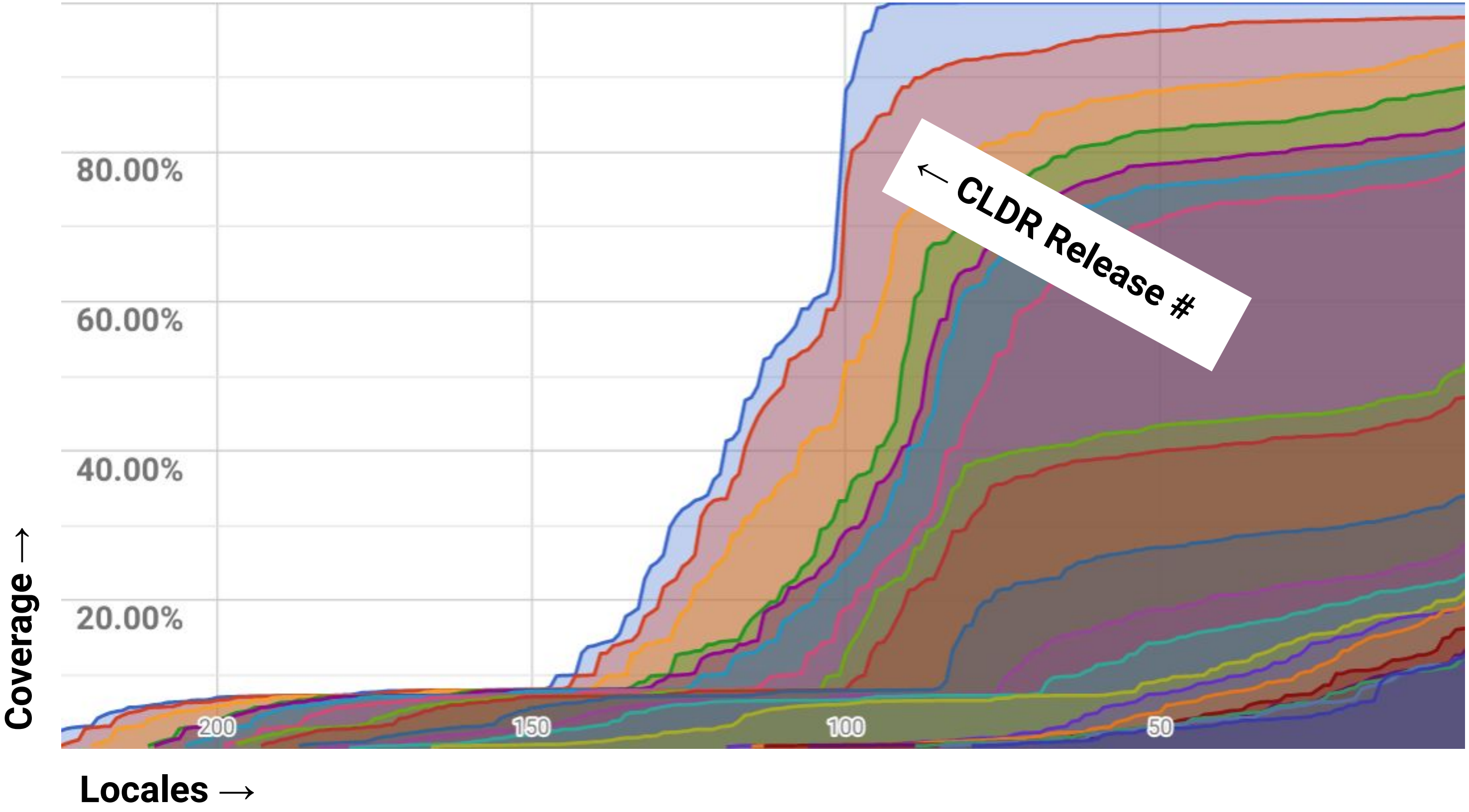
ICU4X Project Status



Data Management

How ICU4X is, at its core, a pipeline
for locale data.

Common Locale Data Repository (CLDR)



Locale Data Is Challenging

- Locale data is Big
  - Quadratic growth:
    # of features * # of locales
- Locale data is Heterogeneous
  - Every piece of data is different
- Locale data is Algorithm-Heavy
  - Need complex code to process it

Multi-Pronged Approach

1
Zero Copy
No memory allocations

2
Stable Files
Backward and forward compatible

3
Pluggable Pipeline
Multiple data sources & overlays

4
Dynamic Loading
Download on demand

5
Static Slicing
Pay for what you use

6
Live Refresh
Unload data without app restart

Modularity

How did we make ICU4X lightweight?

Optimizing for Tree Shaking and Code Size

- Smaller crates
  - Makes dependencies more explicit
- Smaller functions
  - Avoid large conditional code paths that could be dead-code eliminated
- Make error types Copy
  - Eliminates invocations of Drop
- Use traits wisely
  - Generics duplicate code
  - Trait objects (*dyn*) hard to optimize
  - Solution: use generics for APIs and delegate into a single low-level fn

Static Analysis for Data Slicing

- **Annotate fns with data requirements**
- **Explicit data provider argument**
- **Analyze DCE'd code for data it needs**
  - Build optimal data files

```rust
impl TimeFormatter {

pub fn try_new_with_length_unstable<D>(
    data_provider: &D,
    locale: &DataLocale,
    length: Time
) -> Result<TimeFormatter, DateTimeError>
where
    D: DataProvider<TimeLengthsV1Marker> +
       DataProvider<TimeSymbolsV1Marker> +
       DataProvider<DecimalSymbolsV1Marker> +
       ?Sized,
```

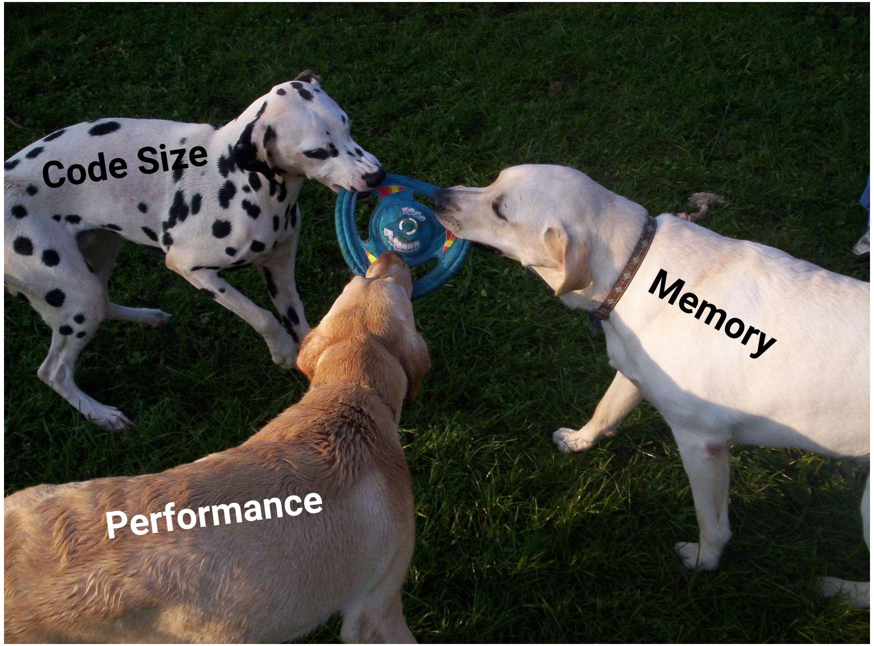
Only these 3 data impls get linked!

Postcard and zero-copy deserialization

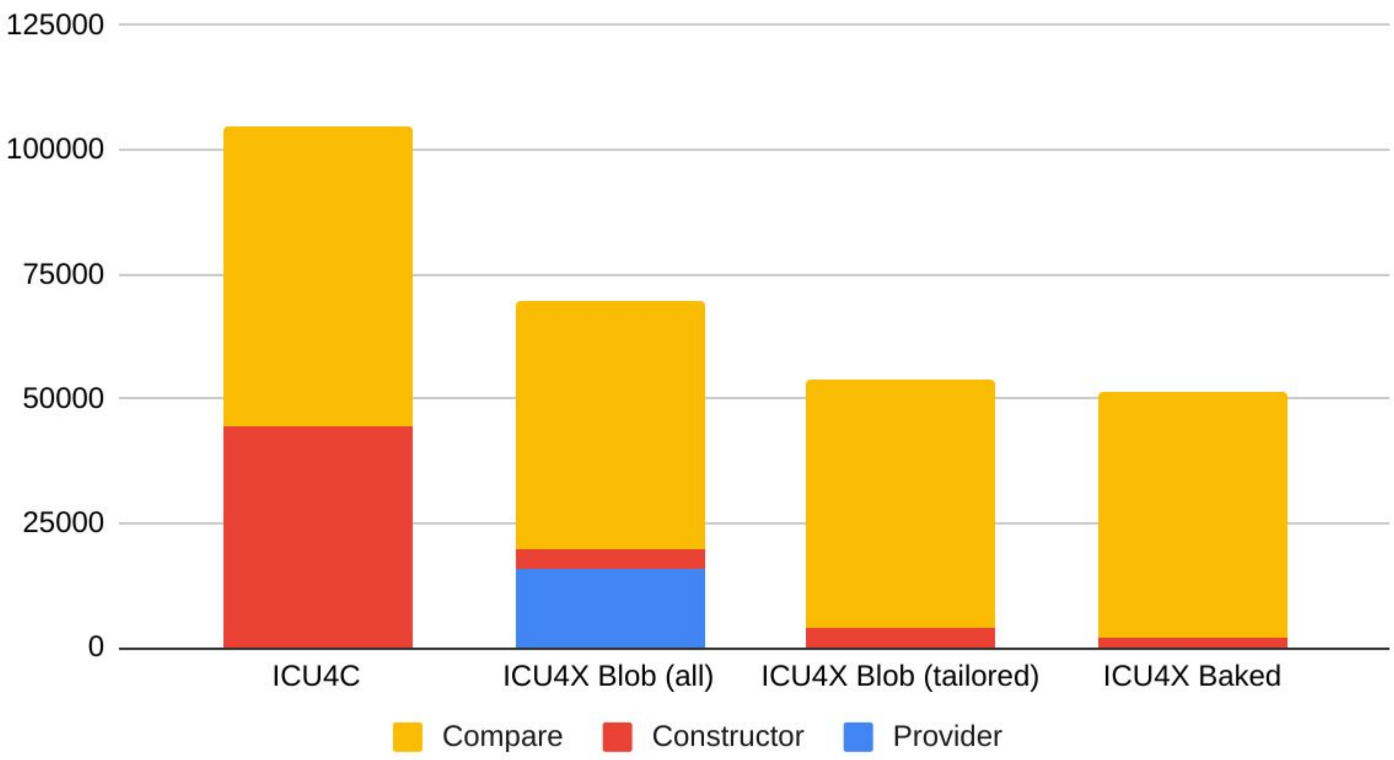
- ICU4X supports dynamically loaded data
  - Download more locales when needed
  - Load from the environment or OS
- The data needs to be serialized for interchange
- But normally, deserialization is expensive!
  - Slow, lots of memory, lots of code
- Enter Postcard with ZeroVec
  - Zero-copy deserialization with small code and no memory allocations!
  - #[no_std] compatible
  - More on ZeroVec later!

Balancing Performance

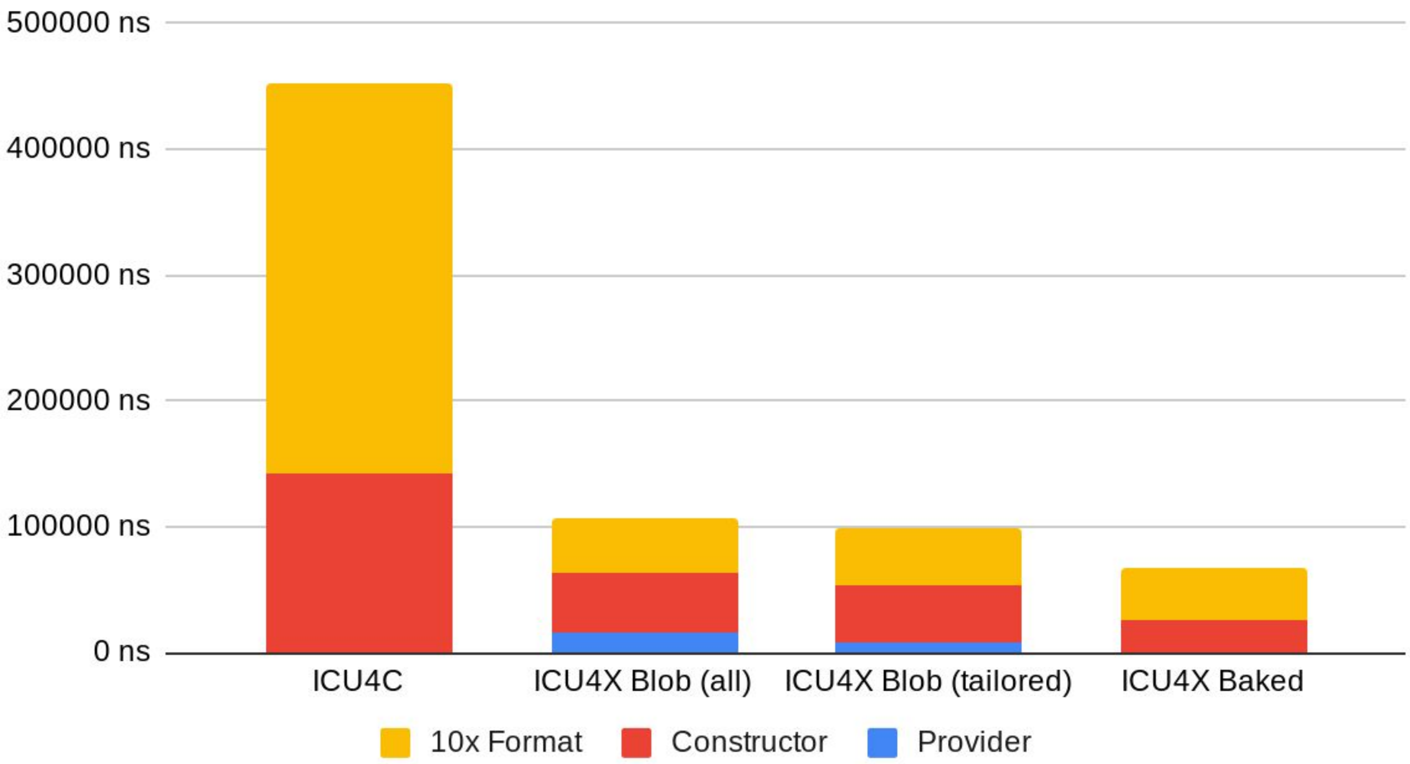
- Sometimes, changes that improve performance also reduce memory and code size
  - Example: zero-copy deserialization
- But, sometimes the goals compete
  - Example: binary search vs hash table
- ICU4X tries to take a holistic view: big wins on one metric could come at smaller losses on another
  - Example: character properties



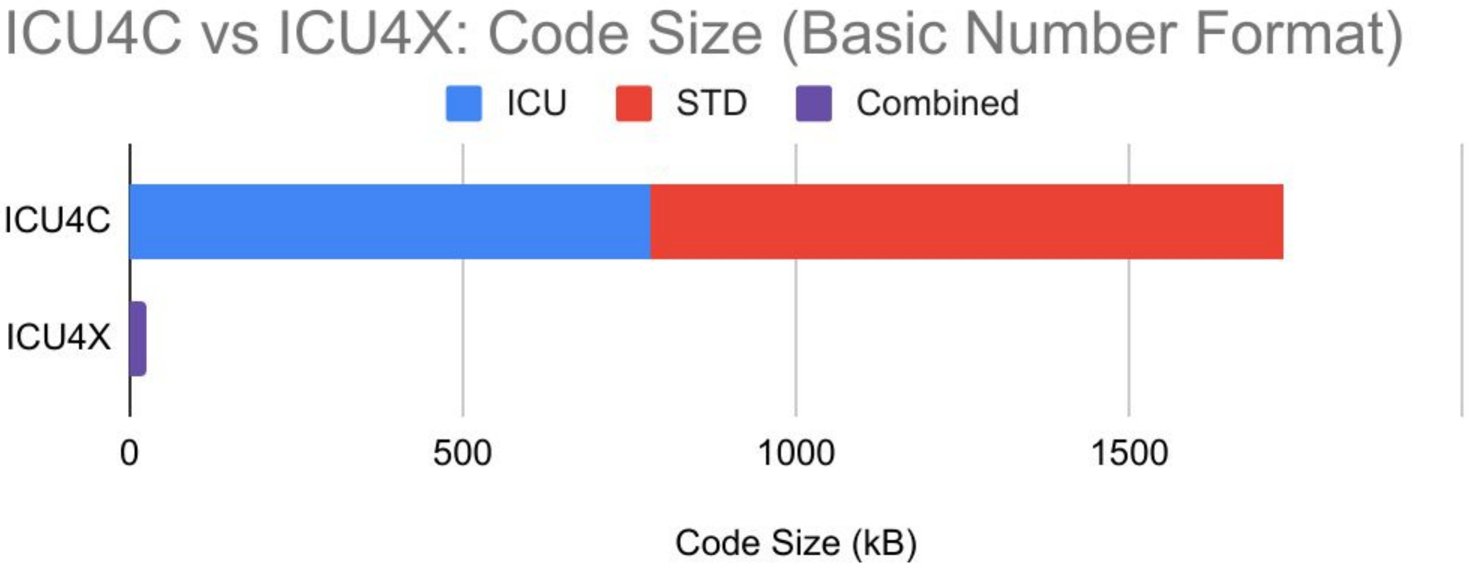
ICU4C 72 vs ICU4X 1.0 PluralRules



ICU4C 72 vs ICU4X 1.0 DateTimeFormatter



Code size for program that prints 1,000,007 in Bangla digits: "১০,০০,০০৭"



Learnings

What can be learned from building a core library for the Rust ecosystem?

no_std*

- Abstract data storage when possible
  - Example: Filesystem data provider
- Embedded Devices
  - Internet-of-Things, etc.
- Good for WASM and portability
  - Fewer dependencies when linking ICU4X
  - Easier and smaller when built to WASM
- * But you require the alloc module?
  - ICU4X is designed such that we can fully remove runtime allocations if necessary. Interested in a fully no-alloc ICU4X? Let us know!

Deliberate Dependencies

- cargo tree in icu metacrate has almost 200 lines?! 🧐
- However, most deps are:
  - Duplicates (lots of those)
  - Sub-components of ICU4X
  - Unsafe utilities
  - Build-time deps
- tl;dr, Small crates are good, but they make dependencies look bigger than they really are

```
│   ├── tinystr v0.6.2 (/usr/local/googl
│   ├── writeable v0.4.1 (/usr/local/goo
│   └── zerovec v0.8.1 (/usr/local/googl
├── icu_collator v1.0.0-beta1 (/usr/loca
│   ├── displaydoc v0.2.3 (proc-macro) (
│   ├── icu_collections v1.0.0-beta1 (/u
│   │   ├── displaydoc v0.2.3 (proc-macr
│   │   ├── yoke v0.6.1 (/usr/local/goog
│   │   ├── zerofrom v0.1.1 (/usr/local/
│   │   └── zerovec v0.8.1 (/usr/local/g
│   ├── icu_locid v1.0.0-beta1 (/usr/loc
│   ├── icu_normalizer v1.0.0-beta1 (/us
│   │   ├── displaydoc v0.2.3 (proc-macr
│   │   ├── icu_collections v1.0.0-beta1
│   │   ├── icu_properties v1.0.0-beta1
│   │   │   ├── displaydoc v0.2.3 (proc-
│   │   │   ├── icu_collections v1.0.0-b
│   │   │   ├── icu_provider v1.0.0-beta
│   │   │   └── zerovec v0.8.1 (/usr/loc
│   │   ├── icu_provider v1.0.0-beta1 (/
│   │   ├── smallvec v1.9.0
│   │   ├── utf16_iter v1.0.3
│   │   ├── utf8_iter v1.0.3
│   │   ├── write16 v1.0.0
```

Be Friendly

- No implicit panics
  - Clippy-enforced checks:
    - indexing_slicing
    - unwrap_used
    - …
- Data-driven algorithms increase the potential error space
  - Validate when it's cheap
  - When it's expensive: log::warn, debug_assert, and/or best effort (handle errors gracefully)

Closing Thoughts

Key Takeaways

★ i18n consists of subtle, data-driven algorithms

★ Use ICU4X for your i18n needs in Rust and beyond

★ You can build a large library that is no_std, doesn't panic, and remains suitable for slicing and other compiler optimizations

Thank you for your time!

icu4x.unicode.org