Huge Graph Analysis on Your Own Server with WebGraph in Rust

Tommaso Fontana, **Sebastiano Vigna,** Stefano Zacchiroli

Michele Andreata, Lorenzo Cimini, Davide Cologni, Matteo Dell'Acqua, Dario Moschetti, Valentin Tablan, Matteo Zagheno

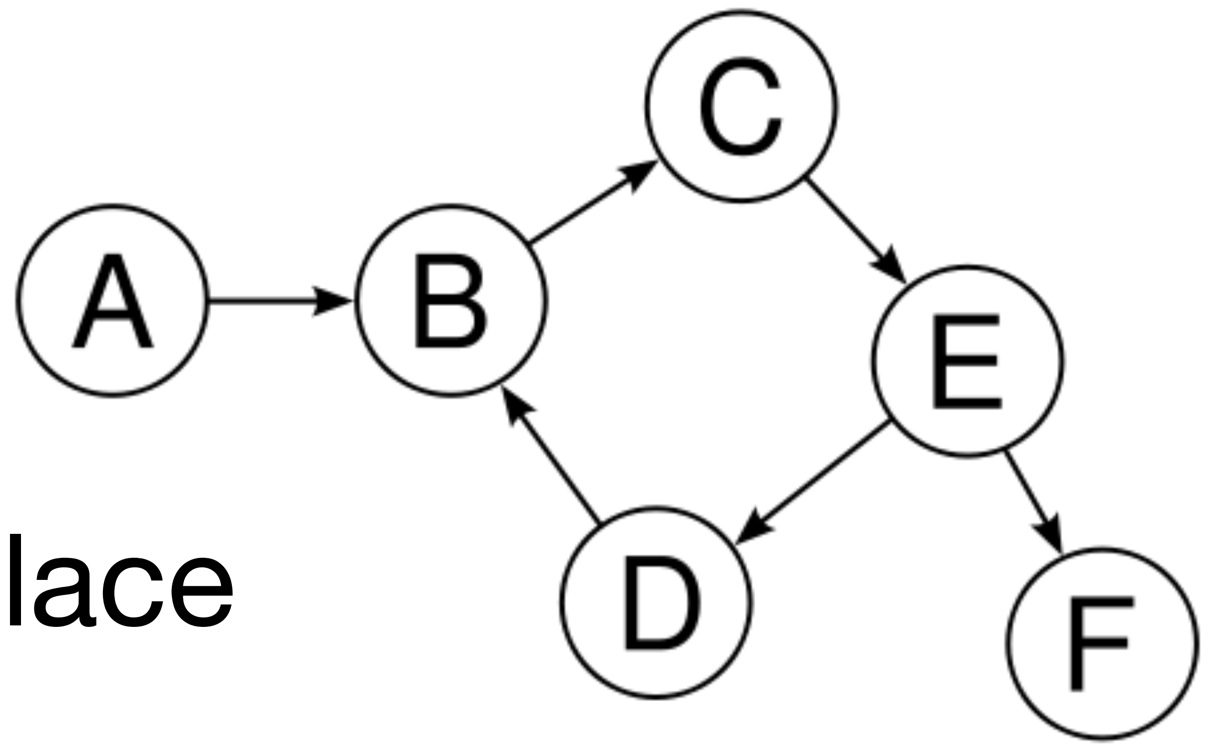
Partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU, and by project ANR COREGRAPHIE, grant ANR-20-CE23-0002 of the French Agence Nationale de la Recherche

Graph Analytics at Scale

Graph Analytics at Scale

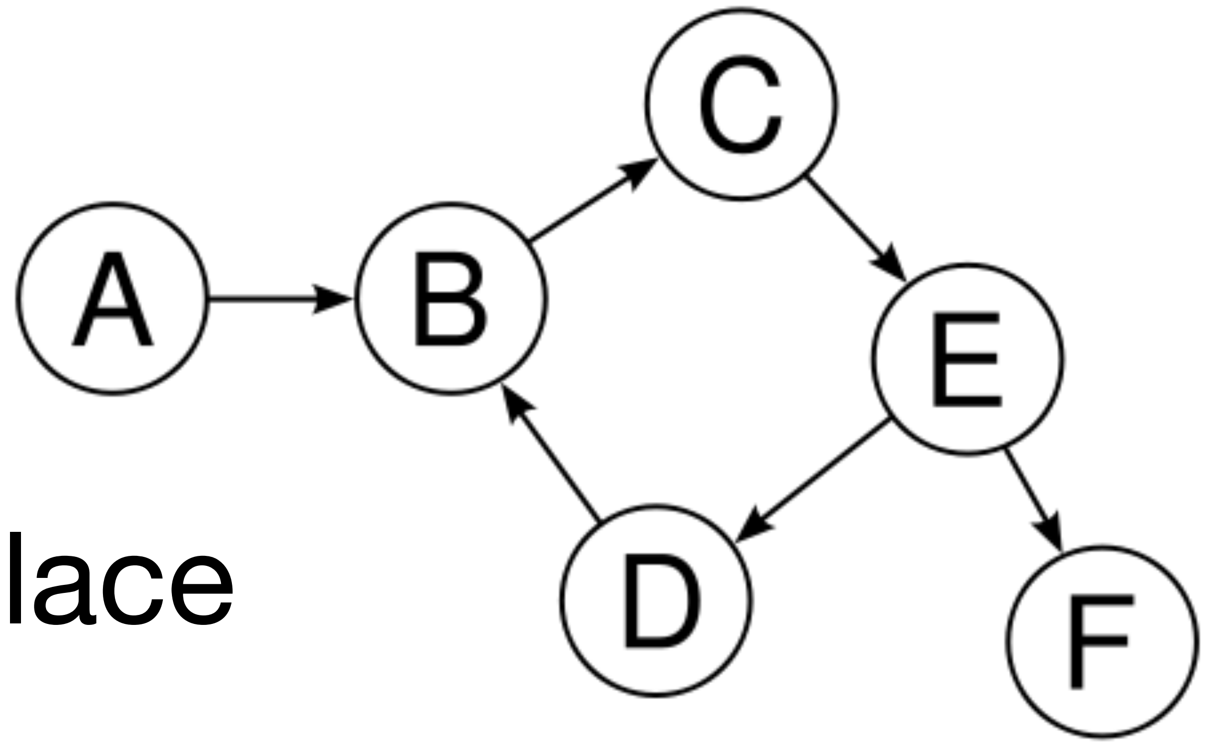
- Today we found huge graphs (>100B arcs) all over the place

Graph Analytics at Scale



- Today we found huge graphs (>100B arcs) all over the place

Graph Analytics at Scale



- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …

Graph Analytics at Scale

- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …

Graph Analytics at Scale

- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …

Graph Analytics at Scale

- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …
- Very large graphs require new approaches

Graph Analytics at Scale

- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …
- Very large graphs require new approaches
- Standard representations in main memory are either impossible (graph too large) or very expensive (many TB of core memory)

Graph Analytics at Scale

- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …
- Very large graphs require new approaches
- Standard representations in main memory are either impossible (graph too large) or very expensive (many TB of core memory)
- Distributed approaches spend a very large amount of time distributing data among nodes

Graph Analytics at Scale

- Today we found huge graphs (>100B arcs) all over the place
- Web snapshots, social networks, biological graphs, …
- Very large graphs require new approaches
- Standard representations in main memory are either impossible (graph too large) or very expensive (many TB of core memory)
- Distributed approaches spend a very large amount of time distributing data among nodes
- What can we do? Compression!

The WebGraph Framework

The WebGraph Framework

- An <u>open source framework</u> for compressed representation of graphs

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)
- Hundreds of publications in major conferences and journals using it (>1500 references)

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)
- Hundreds of publications in major conferences and journals using it (>1500 references)
- In 2011 news went around the world: Facebook had four degrees of separation

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)
- Hundreds of publications in major conferences and journals using it (>1500 references)
- In 2011 news went around the world: Fac
  separation

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)
- Hundreds of publications in major conferences and journals using it (>1500 references)
- In 2011 news went around the world: Facebook had four degrees of separation

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)
- Hundreds of publications in major conferences and journals using it (>1500 references)
- In 2011 news went around the world: Facebook had four degrees of separation
- The measurement was performed at Facebook in collaboration with our group using WebGraph (at that time, 721M nodes, 69B links, just 211GB!)

The WebGraph Framework

- An open source framework for compressed representation of graphs
- One of the most long-lived projects of this kind (>20 years!)
- Hundreds of publications in major conferences and journals using it (>1500 references)
- In 2011 news went around the world: Facebook had four degrees of separation
- The measurement was performed at Facebook in collaboration with our group using WebGraph (at that time, 721M nodes, 69B links, just 211GB!)
- Common Crawl distributes data using WebGraph

Software Heritage History Graph

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

Software Heritage History Graph

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

- The largest public archive of public and git-style version control history

Software Heritage History Graph

- The largest public archive of public and git-style version control history
- Data model: a Merkle direct acyclic graph (intuitively: a single git repository with the development history of all public code)

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

Software Heritage History Graph

- The largest public archive of public and git-style version control history
- Data model: a Merkle direct acyclic graph (intuitively: a single git repository with the development history of all public code)
- One of the largest graphs of human activity available

Software Heritage History Graph

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

- The largest public archive of public and git-style version control history
- Data model: a Merkle direct acyclic graph (intuitively: a single git repository with the development history of all public code)
- One of the largest graphs of human activity available
- 44 billion nodes, 769 billion arcs (December 2024), represented by WebGraph in 251GB instead of >6TB!

Software Heritage History Graph

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

- The largest public archive of public and git-style version control history
- Data model: a Merkle direct acyclic graph (intuitively: a single git repository with the development history of all public code)
- One of the largest graphs of human activity available
- 44 billion nodes, 769 billion arcs (December 2024), represented by WebGraph in 251GB instead of >6TB!
- The previous Java WebGraph-based pipeline for graph analytics was born out of a collaboration between Inria and the Università degli Studi di Milano

Software Heritage History Graph

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

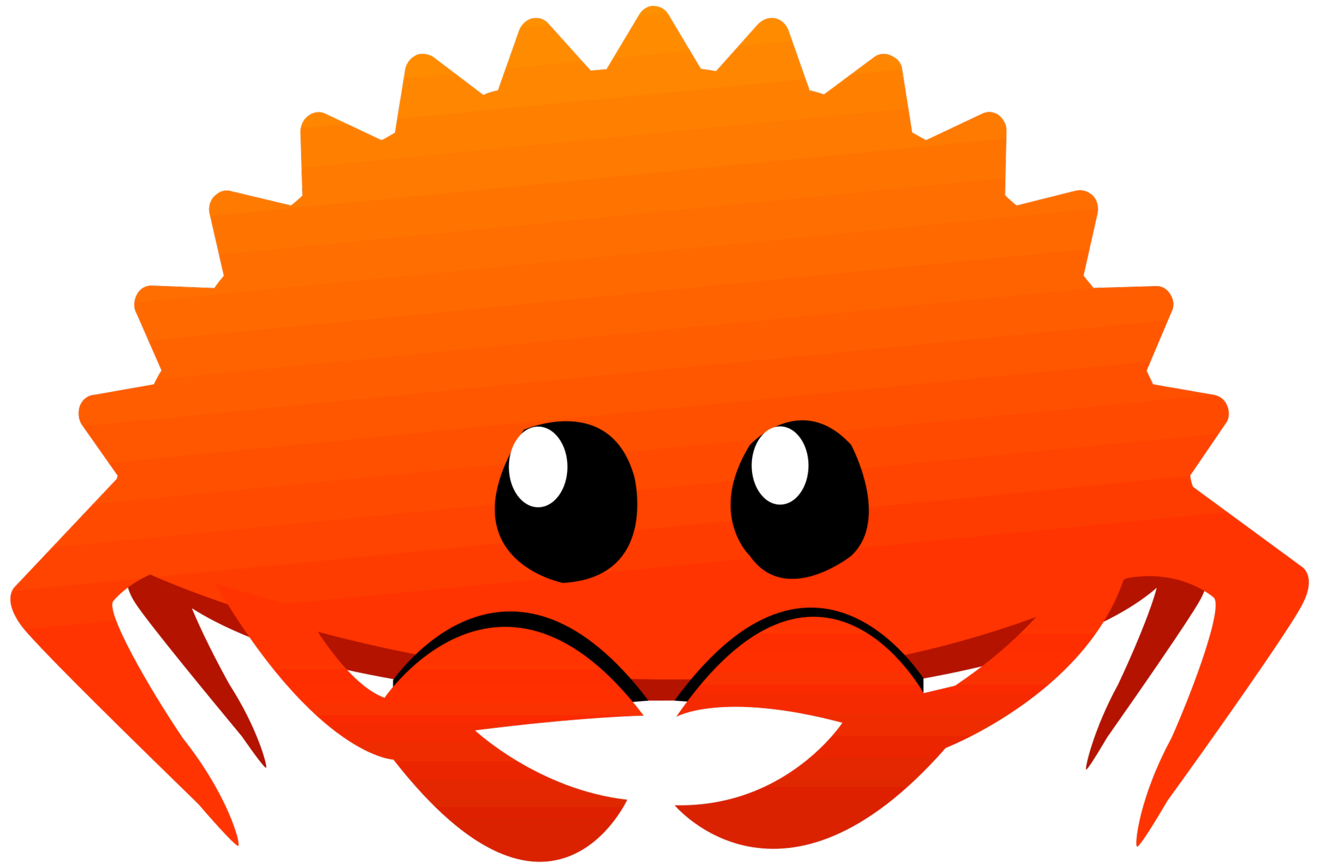
- The largest public archive of public and git-style version control history
- Data model: a Merkle direct acyclic graph (intuitively: a single git repository with the development history of all public code)
- One of the largest graphs of human activity available
- 44 billion nodes, 769 billion arcs (December 2024), represented by WebGraph in 251GB instead of >6TB!
- The previous Java WebGraph-based pipeline for graph analytics was born out of a collaboration between Inria and the Università degli Studi di Milano
- Storing explicitly the graph makes it possible to perform provenance analysis, plagiarism detection, clone detection, etc., at an unprecedented scale

Software Heritage History Graph

Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

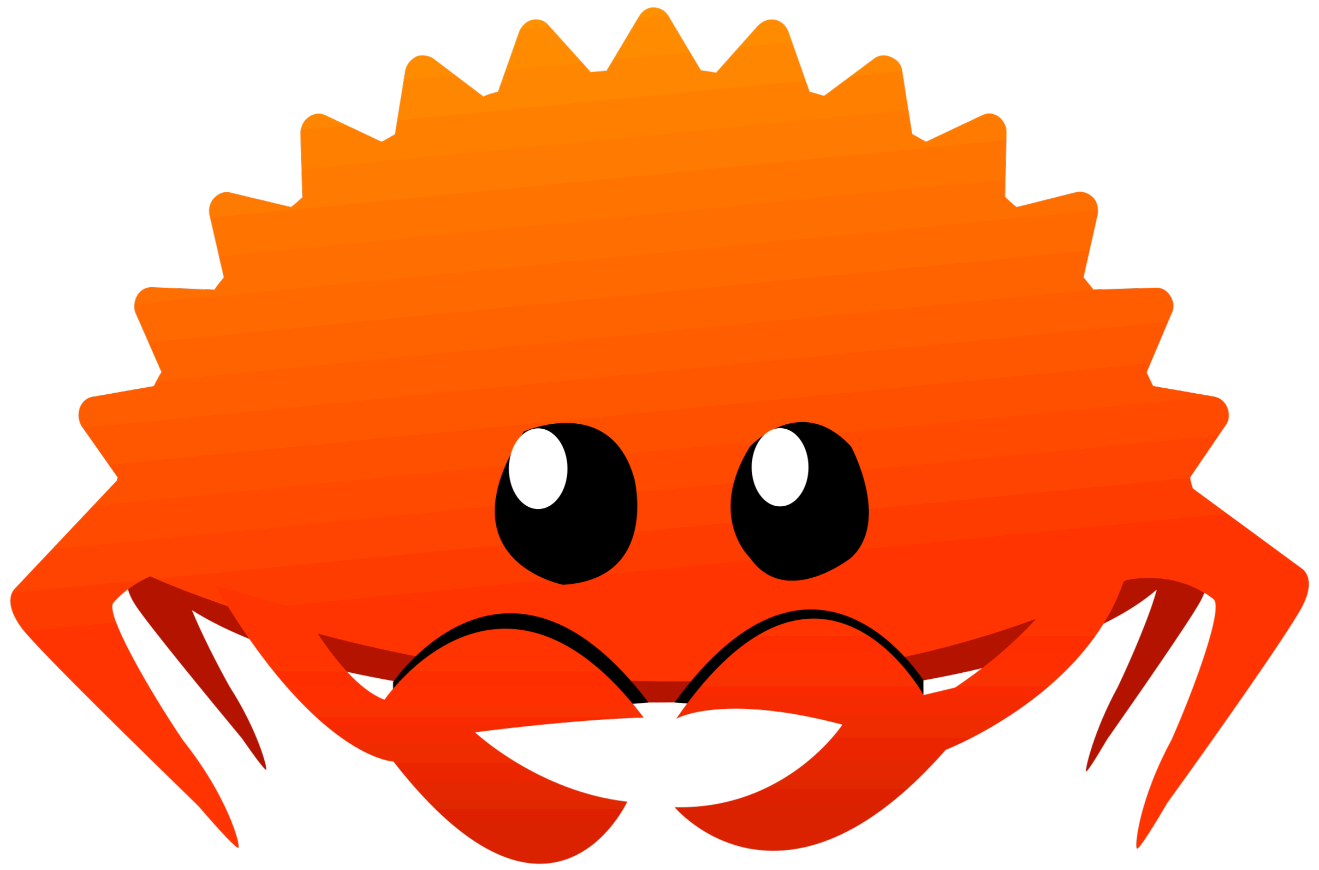
- The largest public archive of public and git-style version control history
- Data model: a Merkle direct acyclic graph (intuitively: a single git repository with the development history of all public code)
- One of the largest graphs of human activity available
- 44 billion nodes, 769 billion arcs (December 2024), represented by WebGraph in 251GB instead of >6TB!
- The previous Java WebGraph-based pipeline for graph analytics was born out of a collaboration between Inria and the Università degli Studi di Milano
- Storing explicitly the graph makes it possible to perform provenance analysis, plagiarism detection, clone detection, etc., at an unprecedented scale
- Still, Java started to get in the way

Moving to Rust

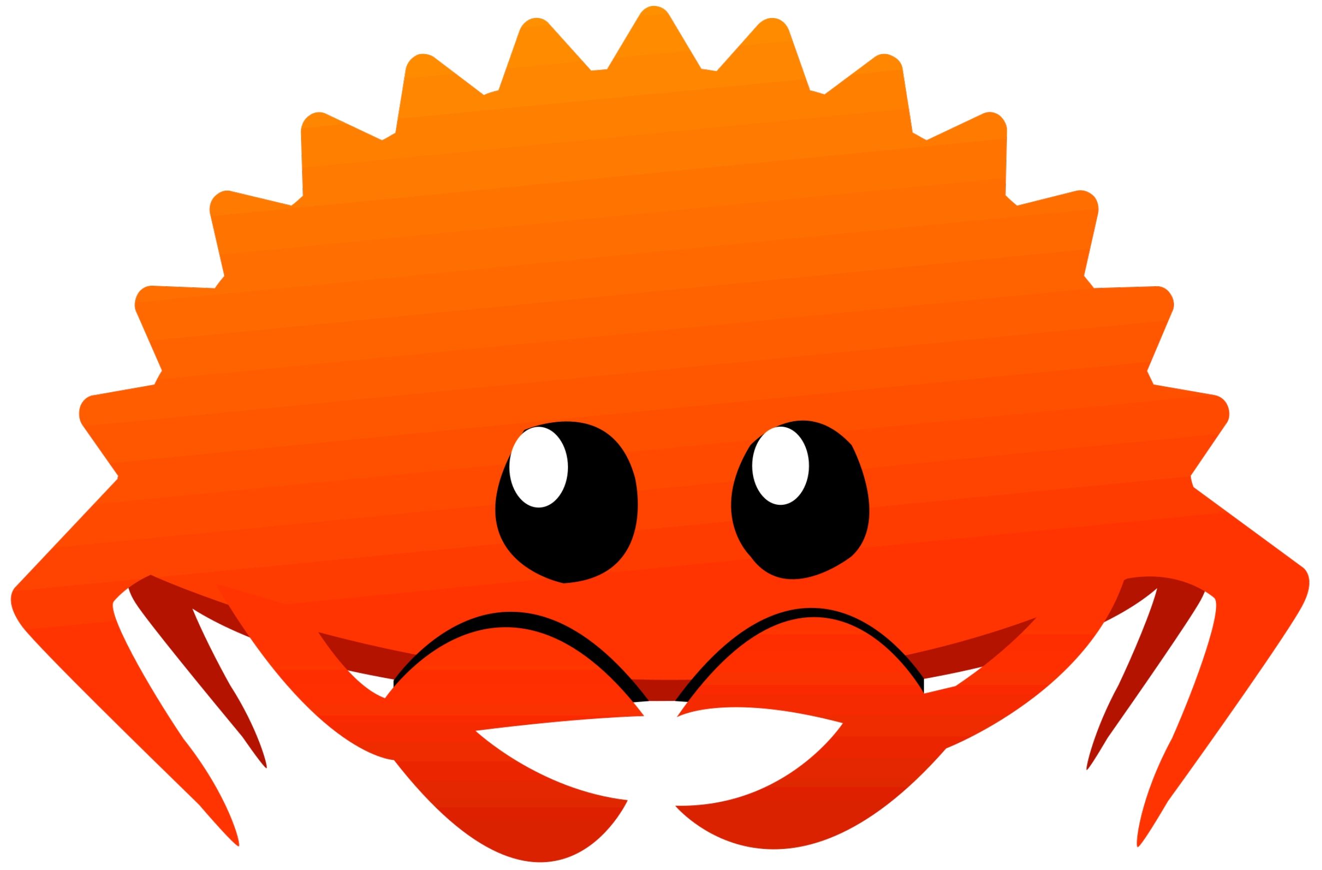


Moving to Rust

- A high-performance, safe language

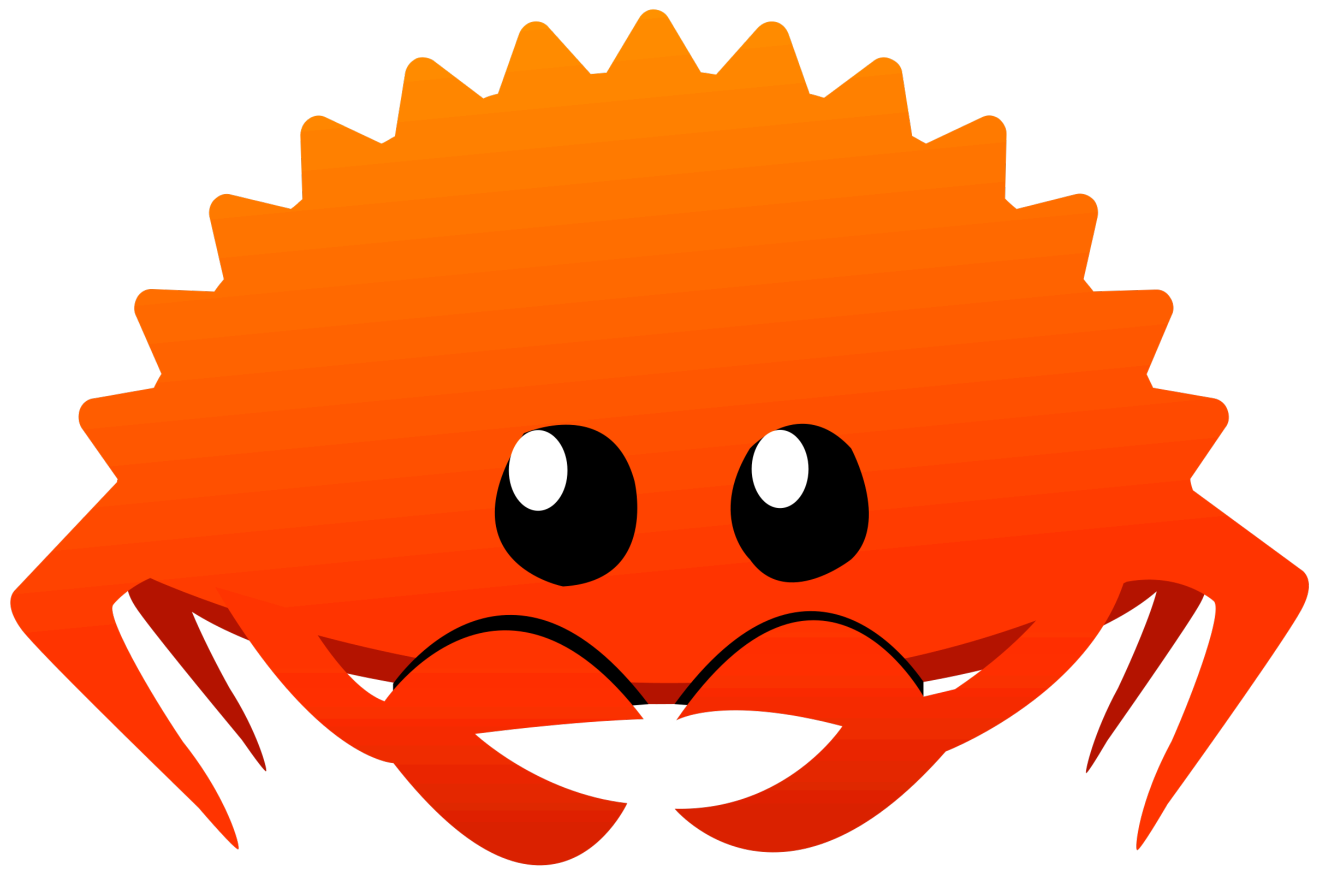


Moving to Rust



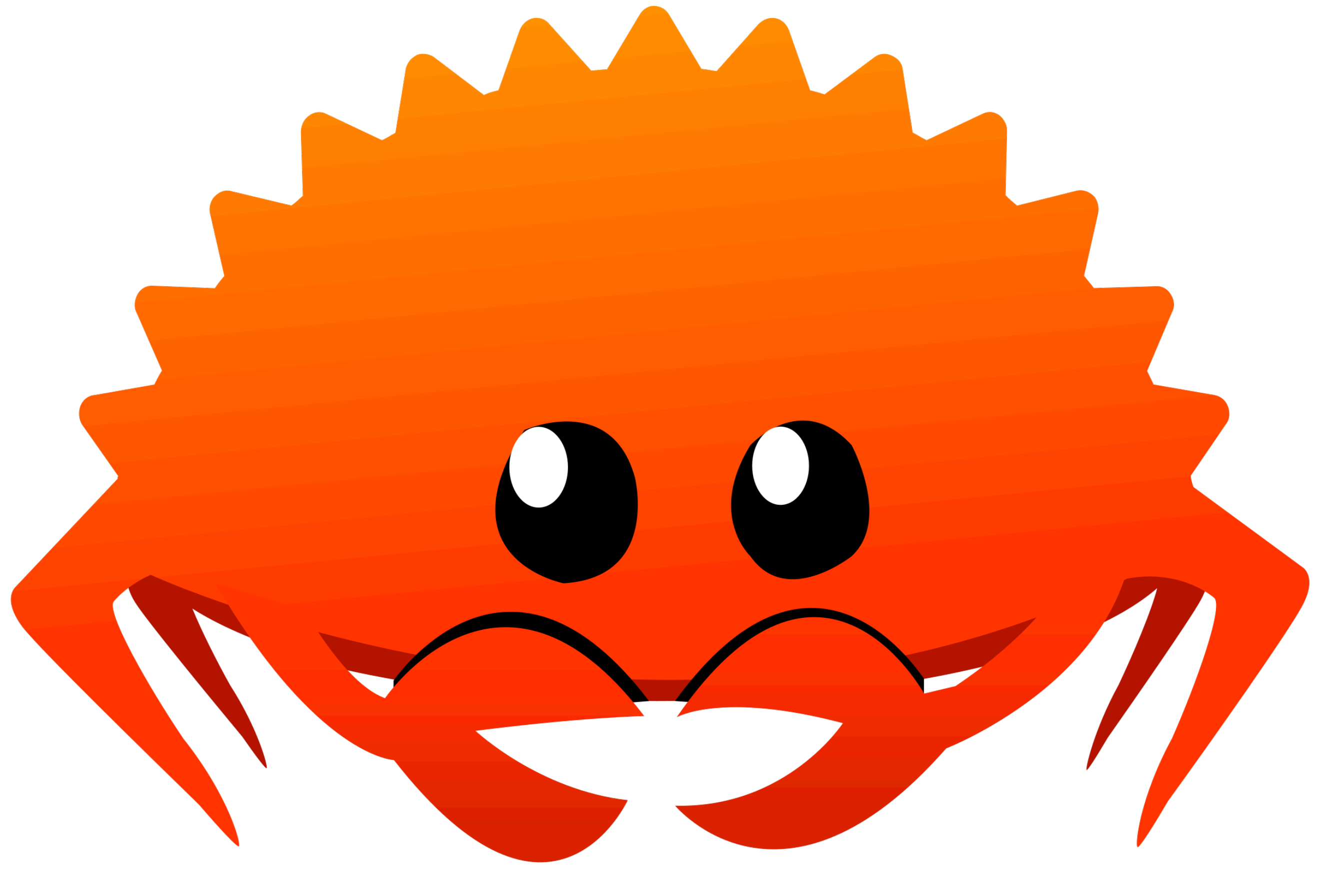
- A high-performance, safe language
- Zero-cost abstractions

Moving to Rust



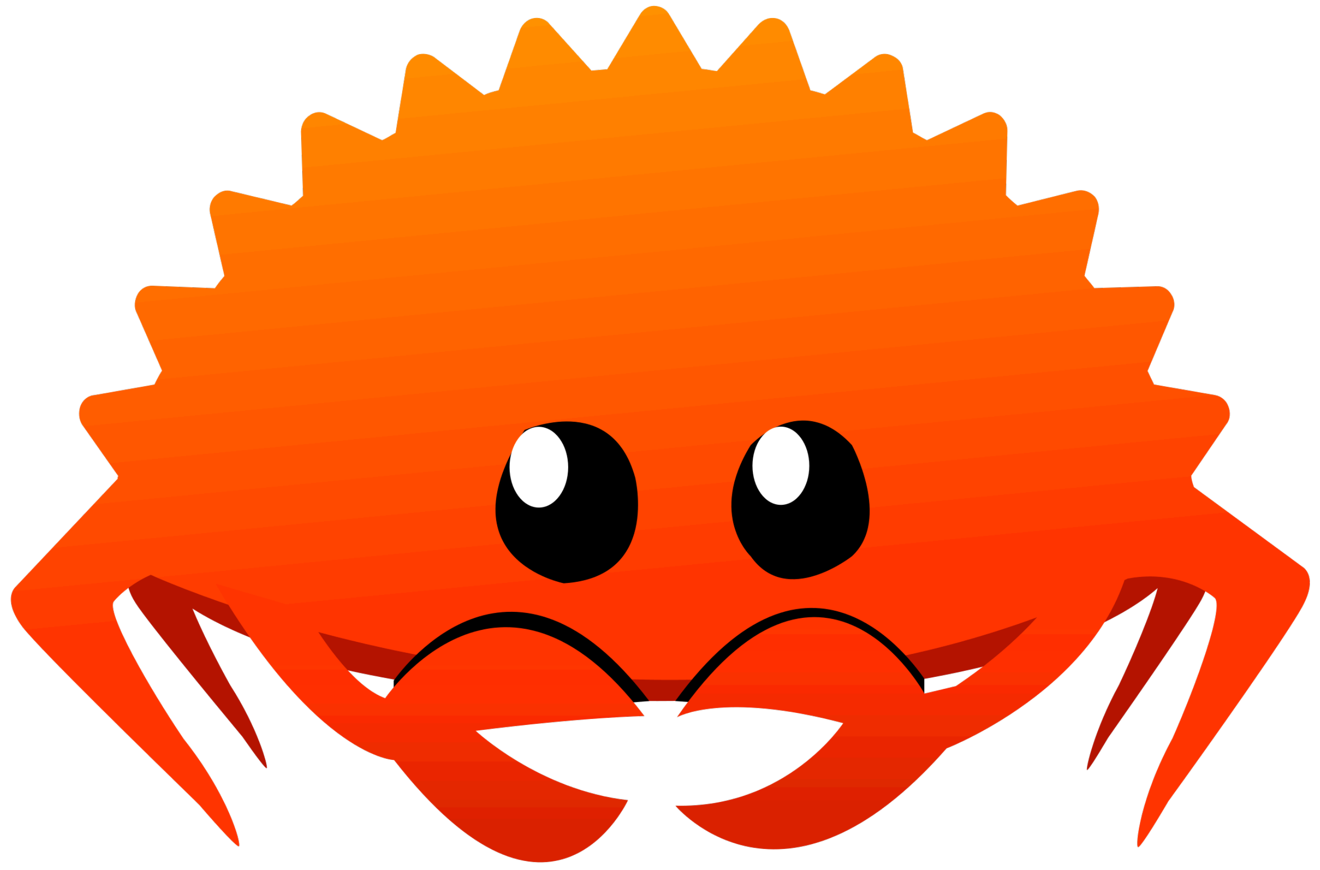
- A high-performance, safe language
- Zero-cost abstractions
- Arrays as large as memory allows

Moving to Rust



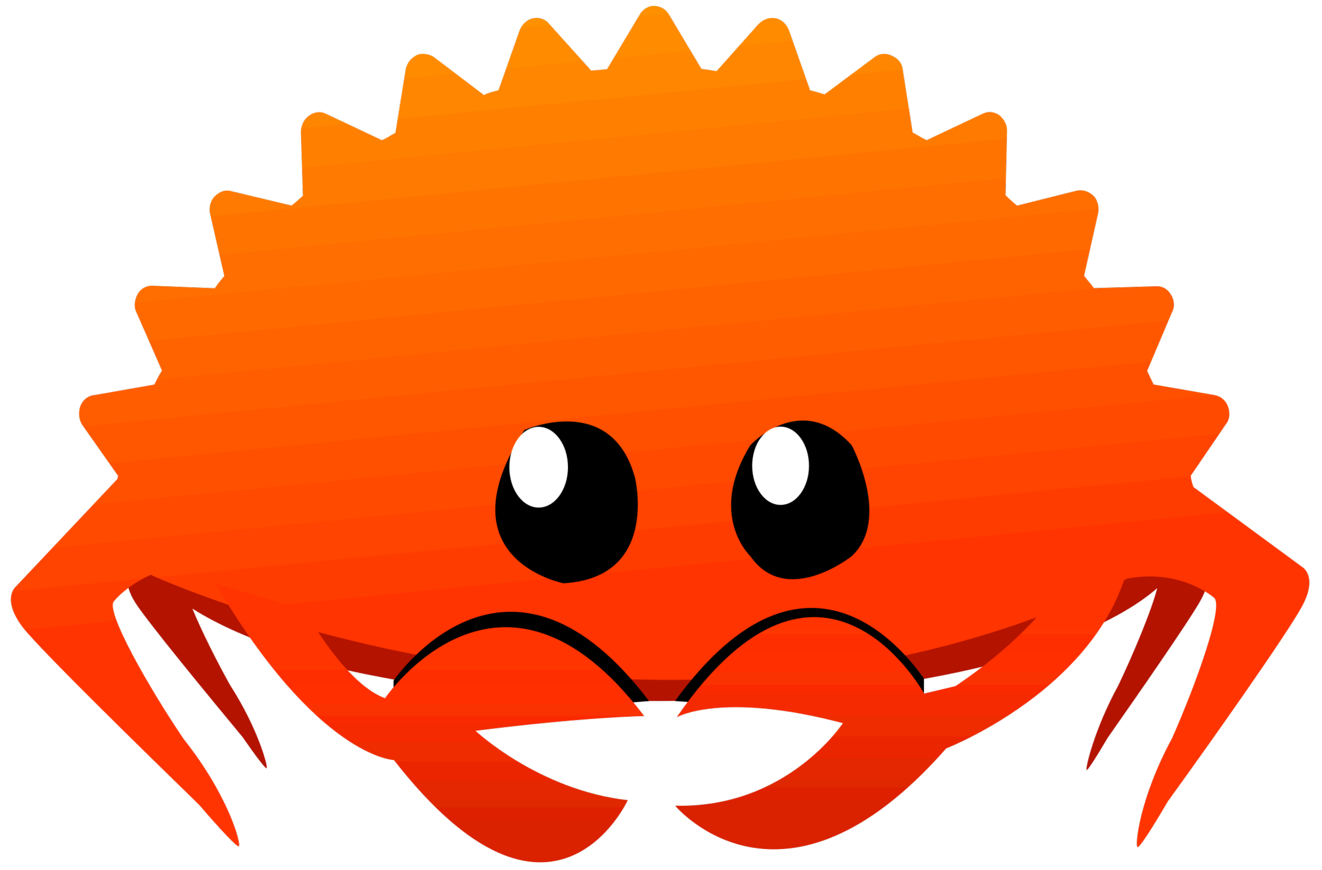
- A high-performance, safe language
- Zero-cost abstractions
- Arrays as large as memory allows
- Fine-grained access to OS facilities (memory mapping)

Moving to Rust



- A high-performance, safe language
- Zero-cost abstractions
- Arrays as large as memory allows
- Fine-grained access to OS facilities (memory mapping)
- Lazy iterators

Moving to Rust



- A high-performance, safe language
- Zero-cost abstractions
- Arrays as large as memory allows
- Fine-grained access to OS facilities (memory mapping)
- Lazy iterators
- Moving to Rust required porting or rethinking several key ideas

Crates

Crates

- ε-serde (epserde): ε-copy serialization/deserialization

Crates

- ε-serde (epserde): ε-copy serialization/deserialization
- mem_dbg: fast memory footprint analysis

Crates

- ε-serde (epserde): ε-copy serialization/deserialization
- mem_dbg: fast memory footprint analysis
- sux: succinct data structures

Crates

- ε-serde (epserde): ε-copy serialization/deserialization
- mem_dbg: fast memory footprint analysis
- sux: succinct data structures
- dsi-bitstream: fast bit streams with support for several types of instantaneous codes

Crates

- ε-serde (epserde): ε-copy serialization/deserialization
- mem_dbg: fast memory footprint analysis
- sux: succinct data structures
- dsi-bitstream: fast bit streams with support for several types of instantaneous codes
- dsi-progress-logger: time-based (concurrent) progress logger

Crates

- ε-serde (epserde): ε-copy serialization/deserialization
- mem_dbg: fast memory footprint analysis
- sux: succinct data structures
- dsi-bitstream: fast bit streams with support for several types of instantaneous codes
- dsi-progress-logger: time-based (concurrent) progress logger
- …and, of course, webgraph

ε-serde

ε-serde

- ε-copy *ser*ialization and *de*serialization

ε-serde

- ε-copy *ser*ialization and *de*serialization
- Like zero-copy, without the limitations (and, of course, with other limitations)

ε-serde

- ε-copy *ser*ialization and *de*serialization
- Like zero-copy, without the limitations (and, of course, with other limitations)
- Unlike abomonation, it does not change the memory (e.g., you can map immutable files into memory)

ε-serde

- ε-copy *ser*ialization and *de*serialization
- Like zero-copy, without the limitations (and, of course, with other limitations)
- Unlike abomonation, it does not change the memory (e.g., you can map immutable files into memory)
- Unlike Zerovec, no impact on performance, and you use standard structures

ε-serde

- ε-copy *ser*ialization and *de*serialization
- Like zero-copy, without the limitations (and, of course, with other limitations)
- Unlike abomonation, it does not change the memory (e.g., you can map immutable files into memory)
- Unlike Zerovec, no impact on performance, and you use standard structures
- Unlike rkiv, the structure you deserialize is the structure you serialize, and no impact on performance

ε-serde

- ε-copy *ser*ialization and *de*serialization
- Like zero-copy, without the limitations (and, of course, with other limitations)
- Unlike abomonation, it does not change the memory (e.g., you can map immutable files into memory)
- Unlike Zerovec, no impact on performance, and you use standard structures
- Unlike rkiv, the structure you deserialize is the structure you serialize, and no impact on performance
- Requires collaboration from the underlying struct: the types you want to ε-copy must be type parameters

ε-serde

ε-serde

- Types are zero-copy (ZeroCopy trait) or deep-copy (DeepCopy trait)

ε-serde

- Types are zero-copy (ZeroCopy trait) or deep-copy (DeepCopy trait)
- Sequences (vectors, boxed slices, etc.) of zero-copy types are replaced by references to slices, without any copying

ε-serde

- Types are zero-copy (ZeroCopy trait) or deep-copy (DeepCopy trait)
- Sequences (vectors, boxed slices, etc.) of zero-copy types are replaced by references to slices, without any copying
- The rest of the structure, usually a small (ε-) fraction of the space occupancy, is allocated normally

ε-serde

- Types are zero-copy (ZeroCopy trait) or deep-copy (DeepCopy trait)
- Sequences (vectors, boxed slices, etc.) of zero-copy types are replaced by references to slices, without any copying
- The rest of the structure, usually a small (ε-) fraction of the space occupancy, is allocated normally
- We use disjoint trait implementation based on an associated type to make the framework behave in a different way for zero-copy and deep-copy types

ε-serde

- Types are zero-copy (ZeroCopy trait) or deep-copy (DeepCopy trait)
- Sequences (vectors, boxed slices, etc.) of zero-copy types are replaced by references to slices, without any copying
- The rest of the structure, usually a small (ε-) fraction of the space occupancy, is allocated normally
- We use disjoint trait implementation based on an associated type to make the framework behave in a different way for zero-copy and deep-copy types
- After deserialization you get a structure containing references to the original memory, and you need to pack it in a MemCase if you need to move it around

ε-serde

- Types are zero-copy (ZeroCopy trait) or deep-copy (DeepCopy trait)
- Sequences (vectors, boxed slices, etc.) of zero-copy types are replaced by references to slices, without any copying
- The rest of the structure, usually a small (ε-) fraction of the space occupancy, is allocated normally
- We use disjoint trait implementation based on an associated type to make the framework behave in a different way for zero-copy and deep-copy types
- After deserialization you get a structure containing references to the original memory, and you need to pack it in a MemCase if you need to move it around
- You cannot have references in the structure

Disjoint Impls

PR #1672
RustyYato trick

Disjoint Impls

PR #1672
RustyYato trick

```
pub struct Zero {}
pub struct Deep {}

pub trait CopyType: Sized {
    type Copy;
}
```

Disjoint Impls

PR #1672
RustyYato trick

```
pub struct Zero {}
pub struct Deep {}

pub trait CopyType: Sized {
    type Copy;
}

pub trait ZeroCopy: CopyType<Copy = Zero> {}

impl<T: CopyType<Copy = Zero>> ZeroCopy for T {}

pub trait DeepCopy: CopyType<Copy = Deep> {}

impl<T: CopyType<Copy = Deep>> DeepCopy for T {}
```

Disjoint Impls

PR #1672
RustyYato trick

```
pub struct Zero {}
pub struct Deep {}

pub trait CopyType: Sized {
    type Copy;
}

pub trait ZeroCopy: CopyType<Copy = Zero> {}

impl<T: CopyType<Copy = Zero>> ZeroCopy for T {}

pub trait DeepCopy: CopyType<Copy = Deep> {}

impl<T: CopyType<Copy = Deep>> DeepCopy for T {}

// This is not possible directly--you need a helper
struct and T: CopyType<Copy = Zero/Deep>
impl<T: ZeroCopy> Deserialize for T { ... }
impl<T: DeepCopy> Deserialize for T { ... }
```

Example

Example

```
#[derive(Epserde, Debug, PartialEq)]
struct MyStruct<A> {
    id: isize,
    data: A,
}
```

Example

```rust
#[derive(Epserde, Debug, PartialEq)]
struct MyStruct<A> {
    id: isize,
    data: A,
}
// Create a structure where A is a Vec<isize>
let s: MyStruct<Vec<isize>> = MyStruct { id: 0, data: vec![0, 1, 2, 3] };
// Serialize it
let mut file = std::env::temp_dir();
file.push("serialized");
s.store(&file);
```

Example

```
#[derive(Epserde, Debug, PartialEq)]
struct MyStruct<A> {
    id: isize,
    data: A,
}
// Create a structure where A is a Vec<isize>
let s: MyStruct<Vec<isize>> = MyStruct { id: 0, data: vec![0, 1, 2, 3] };
// Serialize it
let mut file = std::env::temp_dir();
file.push("serialized");
s.store(&file);
// Load the serialized form in a buffer
let b = std::fs::read(&file)?;
```

Example

```rust
#[derive(Epserde, Debug, PartialEq)]
struct MyStruct<A> {
    id: isize,
    data: A,
}
// Create a structure where A is a Vec<isize>
let s: MyStruct<Vec<isize>> = MyStruct { id: 0, data: vec![0, 1, 2, 3] };
// Serialize it
let mut file = std::env::temp_dir();
file.push("serialized");
s.store(&file);
// Load the serialized form in a buffer
let b = std::fs::read(&file)?;

// The type of t will be inferred--it is shown here only for clarity
let t: MyStruct<&[isize]> =
    <MyStruct<Vec<isize>>>::deserialize_eps(b.as_ref())?;
```

Example

```rust
#[derive(Epserde, Debug, PartialEq)]
struct MyStruct<A> {
    id: isize,
    data: A,
}
// Create a structure where A is a Vec<isize>
let s: MyStruct<Vec<isize>> = MyStruct { id: 0, data: vec![0, 1, 2, 3] };
// Serialize it
let mut file = std::env::temp_dir();
file.push("serialized");
s.store(&file);
// Load the serialized form in a buffer
let b = std::fs::read(&file)?;

// The type of t will be inferred--it is shown here only for clarity
let t: MyStruct<&[isize]> =
    <MyStruct<Vec<isize>>>::deserialize_eps(b.as_ref())?;

// This is a traditional deserialization instead
let t: MyStruct<Vec<isize>> =
    <MyStruct<Vec<isize>>>::load_full(&file)?;
```

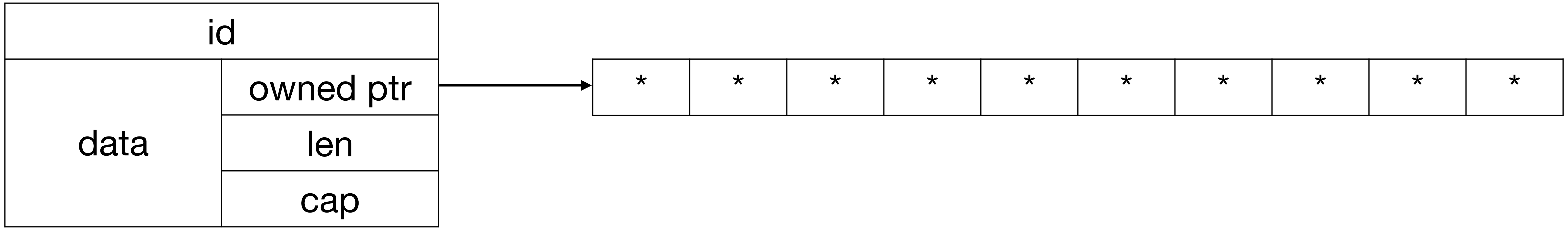
Example

```rust
#[derive(Epserde, Debug, PartialEq)]
struct MyStruct<A> {
    id: isize,
    data: A,
}
// Create a structure where A is a Vec<isize>
let s: MyStruct<Vec<isize>> = MyStruct { id: 0, data: vec![0, 1, 2, 3] };
// Serialize it
let mut file = std::env::temp_dir();
file.push("serialized");
s.store(&file);
// Load the serialized form in a buffer
let b = std::fs::read(&file)?;

// The type of t will be inferred--it is shown here only for clarity
let t: MyStruct<&[isize]> =
    <MyStruct<Vec<isize>>>::deserialize_eps(b.as_ref())?;

// This is a traditional deserialization instead
let t: MyStruct<Vec<isize>> =
    <MyStruct<Vec<isize>>>::load_full(&file)?;

// In this case we map the data structure into memory
let u: MemCase<MyStruct<&[isize]>> =
    <MyStruct<Vec<isize>>>::mmap(&file, Flags::empty())?;
```

Example

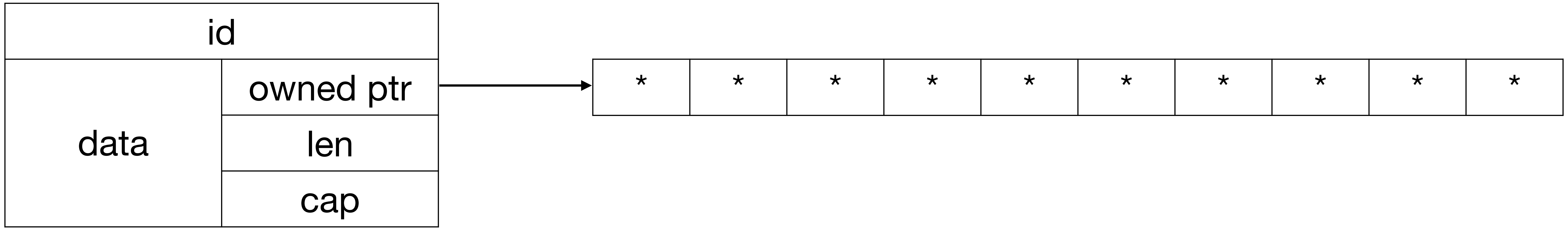
Example

Construction time

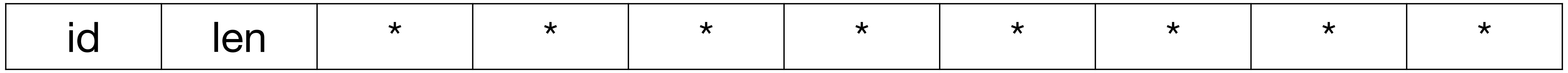


Example

Construction time

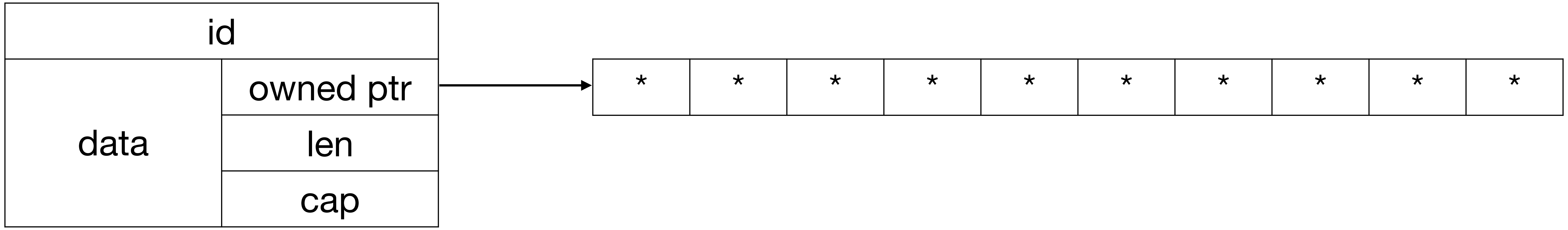


Serialized

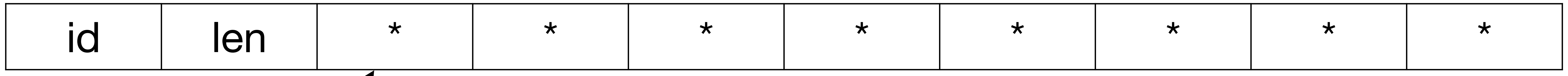


Example

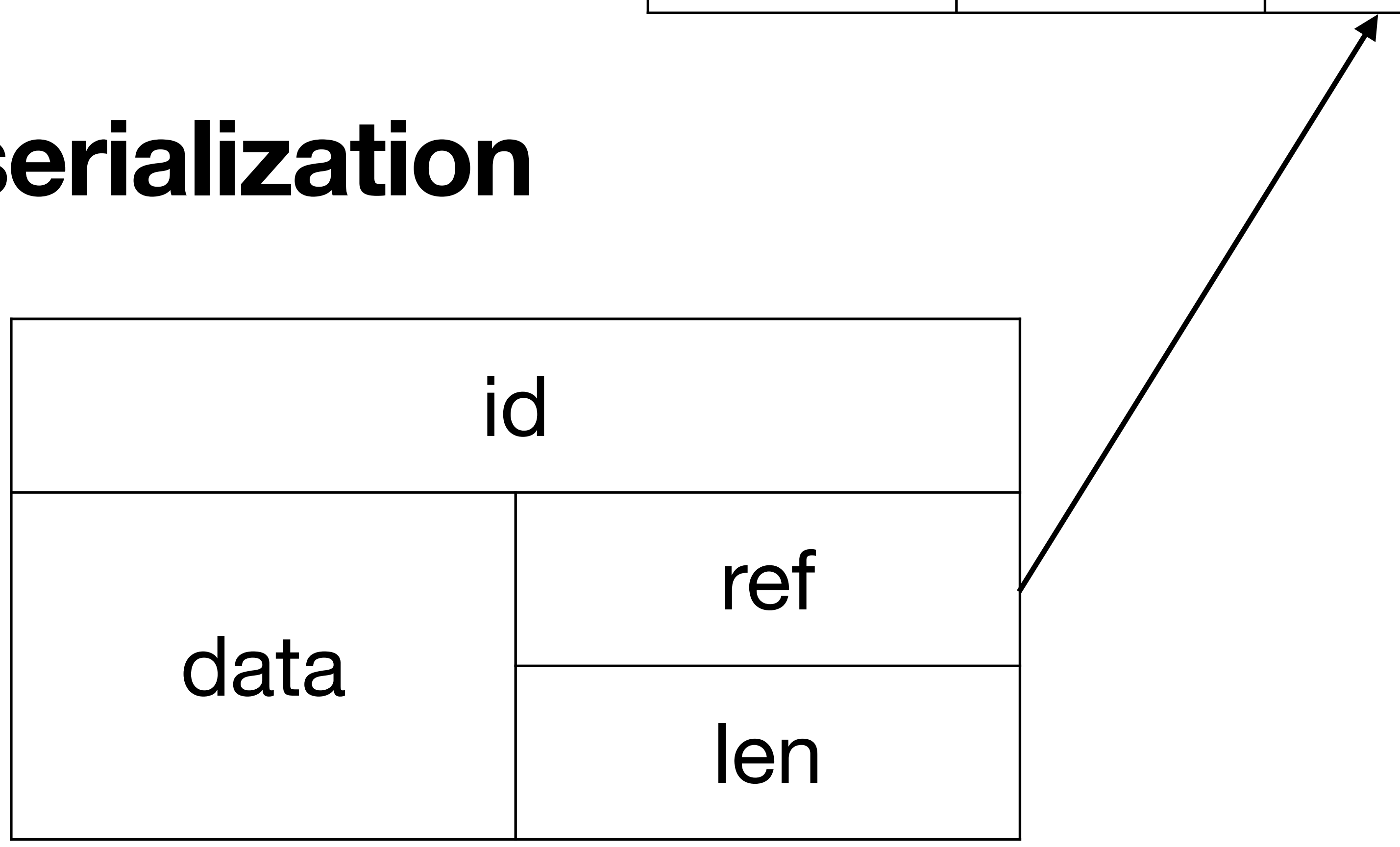
Construction time



Serialized



ε-deserialization



ε-serde

ε-serde

- In the previous example, you are supposed to write all read-only methods for MyStruct using AsRef<&[isize]> as the type parameter: then they will work on the ε-deserialized structure

ε-serde

- In the previous example, you are supposed to write all read-only methods for MyStruct using AsRef<&[isize]> as the type parameter: then they will work on the ε-deserialized structure

- Structures supporting ε-serde can be just used as a type parameter and they will be ε-deserialized recursively

ε-serde

- In the previous example, you are supposed to write all read-only methods for MyStruct using AsRef<&[isize]> as the type parameter: then they will work on the ε-deserialized structure
- Structures supporting ε-serde can be just used as a type parameter and they will be ε-deserialized recursively
- If you have several different such fields, you'll have as many type parameters, which can become a nuisance

ε-serde

- In the previous example, you are supposed to write all read-only methods for MyStruct using AsRef<&[isize]> as the type parameter: then they will work on the ε-deserialized structure
- Structures supporting ε-serde can be just used as a type parameter and they will be ε-deserialized recursively
- If you have several different such fields, you’ll have as many type parameters, which can become a nuisance
- If you think of your structure as a tree, only leaves reachable through a path of ε-serde–supporting type parameters will be zero-copied (given that they can be zero-copied)

mem_dbg

mem_dbg

- High-performance memory-occupancy detector

mem_dbg

- High-performance memory-occupancy detector
- Leverages on ε-serde’s notion of zero-copy data to avoid iterating on collections

mem_dbg

- High-performance memory-occupancy detector
- Leverages on ε-serde’s notion of zero-copy data to avoid iterating on collections

```
Allocated:    2281701509
get_size:     1879048240 152477833 ns
deep_size_of: 1879048240 152482000 ns
size_of:      2281701432 152261958 ns
mem_size:     2281701424 209 ns
```

mem_dbg

- High-performance memory-occupancy detector
- Leverages on ε-serde’s notion of zero-copy data to avoid iterating on collections

```
Allocated:    2281701509
get_size:     1879048240 152477833 ns
deep_size_of: 1879048240 152482000 ns
size_of:      2281701432 152261958 ns
mem_size:     2281701424 209 ns
```

- Additionally, prints memory layouts, including padding

mem_dbg

- High-performance memory-occupancy detector
- Leverages on ε-serde's notion of zero-copy data to avoid iterating on collections

```
Allocated:    2281701509
get_size:     1879048240 152477833 ns
deep_size_of: 1879048240 152482000 ns
size_of:      2281701432 152261958 ns
mem_size:     2281701424 209 ns
```

- Additionally, prints memory layouts, including padding
- By using the nightly offset_of_enum feature we can also display padding in enums

mem_dbg

- High-
- Lever... ons

```
Alloca
get_s
deep_s
size_
mem_s
```

```
1.207 kB 100.00% ●: Struct<TestEnum, Data<alloc::vec::Vec<u8>>>
   16  B   1.33% ├╴a: readme::main::TestEnum
                 │ ├╴Variant: Unnamed
    8  B   0.66% │ ├╴0: usize
    1  B   0.08% │ ╰╴1: u8 [6B]
1.183 kB  98.01% ├╴b: readme::main::Data<alloc::vec::Vec<u8>>
  724  B  59.98% │ ├╴a: alloc::vec::Vec<u8>
  424  B  35.13% │ ├╴b: alloc::vec::Vec<i32>
   35  B   2.90% │ ╰╴c: (u8, alloc::string::String)
    1  B   0.08% │   ├╴0: u8 [7B]
   27  B   2.24% │   ╰╴1: alloc::string::String
    8  B   0.66% ╰╴test: isize
```

- Additi
- By using the nightly offset_of_enum feature we can also display padding in enums

sux

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures

- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures
- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$
- Partial port of sux (C++ project) and Sux4J (Java project)

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures
- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$
- Partial port of sux (C++ project) and Sux4J (Java project)
- There are some existing crates (some porting the projects above)

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures
- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$
- Partial port of sux (C++ project) and Sux4J (Java project)
- There are some existing crates (some porting the projects above)
- Rank and selection

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures
- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$
- Partial port of sux (C++ project) and Sux4J (Java project)
- There are some existing crates (some porting the projects above)
- Rank and selection
- Elias–Fano representation of monotone sequences (e.g., pointers into records)

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures
- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$
- Partial port of sux (C++ project) and Sux4J (Java project)
- There are some existing crates (some porting the projects above)
- Rank and selection
- Elias–Fano representation of monotone sequences (e.g., pointers into records)
- Sorted string compression by prefix omission

sux

- Succinct data structures: data structures using just the space of the information-theoretical lower bound, but with operations asymptotically equivalent to standard structures
- For example, there are 4^n binary trees, so it should be possible to represent a binary tree using $\log 4^n = 2n$ bits (Jacobson) instead of $2n \log n$
- Partial port of sux (C++ project) and Sux4J (Java project)
- There are some existing crates (some porting the projects above)
- Rank and selection
- Elias–Fano representation of monotone sequences (e.g., pointers into records)
- Sorted string compression by prefix omission
- Fast bit vector and slices

sux

sux

- Mix-and-match of arbitrary rank and selection structures

sux

- Mix-and-match of arbitrary rank and selection structures
- Functorial replacement of rank and selection structures (unsafe)

sux

- Mix-and-match of arbitrary rank and selection structures
- Functorial replacement of rank and selection structures (unsafe)
- We use intensively the ambassador crate to delegate all rank, selection, and bit-vector access traits, so you can write

sux

- Mix-and-match of arbitrary rank and selection structures
- Functorial replacement of rank and selection structures (unsafe)
- We use intensively the ambassador crate to delegate all rank, selection, and bit-vector access traits, so you can write

```
let bits = bit_vec![1, 0, 1, 1, 0, 1, 0, 1];
let rank9 = Rank9::new(bits);
let rank9_sel = SelectAdapt::new(rank9, 3);
```

sux

- Mix-and-match of arbitrary rank and selection structures
- Functorial replacement of rank and selection structures (unsafe)
- We use intensively the ambassador crate to delegate all rank, selection, and bit-vector access traits, so you can write

```
let bits = bit_vec![1, 0, 1, 1, 0, 1, 0, 1];
let rank9 = Rank9::new(bits);
let rank9_sel = SelectAdapt::new(rank9, 3);
```

- … and the last structure has also rank methods and access to the underlying bit vector

sux

sux

- Comprehensive set of traits for indexed dictionaries

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms
- Presently, main implementation is Elias–Fano

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms
- Presently, main implementation is Elias–Fano
- Inner workings of the structure are selectable (or functorially modifiable)

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms
- Presently, main implementation is Elias–Fano
- Inner workings of the structure are selectable (or functorially modifiable)
- Also, compact string storage by prefix omission

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms
- Presently, main implementation is Elias–Fano
- Inner workings of the structure are selectable (or functorially modifiable)
- Also, compact string storage by prefix omission
- Main issue: lack of IndexGet or analogous trait makes access cumbersome

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms
- Presently, main implementation is Elias–Fano
- Inner workings of the structure are selectable (or functorially modifiable)
- Also, compact string storage by prefix omission
- Main issue: lack of IndexGet or analogous trait makes access cumbersome
- E.g., a functionally implemented vector that returns i^2 on index i

sux

- Comprehensive set of traits for indexed dictionaries
- Indexing, search, iteration, successor, predecessor, etc. in various forms
- Presently, main implementation is Elias–Fano
- Inner workings of the structure are selectable (or functorially modifiable)
- Also, compact string storage by prefix omission
- Main issue: lack of IndexGet or analogous trait makes access cumbersome
- E.g., a functionally implemented vector that returns i^2 on index i
- Rust and intensional representations do not work very well together ATM

dsi-bitstream

dsi-bitstream

- High-performance bit streams

dsi-bitstream

- High-performance bit streams
- Read/write data by word (settable)

dsi-bitstream

- High-performance bit streams
- Read/write data by word (settable)
- Supports little and big endian files

dsi-bitstream

- High-performance bit streams
- Read/write data by word (settable)
- Supports little and big endian files
- Instantaneous codes for compression: Elias γ, Golomb, etc.

dsi-bitstream

- High-performance bit streams
- Read/write data by word (settable)
- Supports little and big endian files
- Instantaneous codes for compression: Elias γ, Golomb, etc.
- Flexible architecture and benchmarks to tune to your hardware (use decoding tables or not?)

dsi-bitstream

- High-performance bit streams
- Read/write data by word (settable)
- Supports little and big endian files
- Instantaneous codes for compression: Elias γ, Golomb, etc.
- Flexible architecture and benchmarks to tune to your hardware (use decoding tables or not?)
- A γ code read in less than 2ns (for data with the intended distribution)

dsi-bitstream

dsi-bitstream

- Basic traits: BitRead<E> / Bitwrite<E> (E is the endianness)

dsi-bitstream

- Basic traits: BitRead<E> / Bitwrite<E> (E is the endianness)
- Extension traits like GammaRead / GammaWrite add code capabilities

dsi-bitstream

- Basic traits: BitRead<E> / Bitwrite<E> (E is the endianness)
- Extension traits like GammaRead / GammaWrite add code capabilities
- Implementations BufBitReader<E, WR, RP> and BufBitWriter<E, WW, WP> depend on endianness and on the word size used to read or write data

dsi-bitstream

- Basic traits: BitRead<E> / Bitwrite<E> (E is the endianness)
- Extension traits like GammaRead / GammaWrite add code capabilities
- Implementations BufBitReader<E, WR, RP> and BufBitWriter<E, WW, WP> depend on endianness and on the word size used to read or write data
- Moreover, the last parameter is a selector type that chooses whether to use encoding/decoding tables or not for each code

dsi-bitstream

- Basic traits: BitRead<E> / Bitwrite<E> (E is the endianness)
- Extension traits like GammaRead / GammaWrite add code capabilities
- Implementations BufBitReader<E, WR, RP> and BufBitWriter<E, WW, WP> depend on endianness and on the word size used to read or write data
- Moreover, the last parameter is a selector type that chooses whether to use encoding/decoding tables or not for each code
- When reading, the internal bit buffer is twice the read word to make peeking possible (for tables)

dsi-bitstream

- Basic traits: BitRead<E> / Bitwrite<E> (E is the endianness)
- Extension traits like GammaRead / GammaWrite add code capabilities
- Implementations BufBitReader<E, WR, RP> and BufBitWriter<E, WW, WP> depend on endianness and on the word size used to read or write data
- Moreover, the last parameter is a selector type that chooses whether to use encoding/decoding tables or not for each code
- When reading, the internal bit buffer is twice the read word to make peeking possible (for tables)
- Presently WR = u32 and WW = u64 are the best choice

webgraph

webgraph

- Graph are represented by bitstreams

webgraph

- Graph are represented by bitstreams
- Uses dsi-bitstream for instantaneous codes, sux for pointers into the bitstream

webgraph

- Graph are represented by bitstreams
- Uses dsi-bitstream for instantaneous codes, sux for pointers into the bitstream
- On a Software Heritage graph with 34 billion nodes and 517 billion arcs a BFS visit is three time faster than Java (3h)

webgraph

- Graph are represented by bitstreams
- Uses dsi-bitstream for instantaneous codes, sux for pointers into the bitstream
- On a Software Heritage graph with 34 billion nodes and 517 billion arcs a BFS visit is three time faster than Java (3h)
- Unbelievably better ergonomics WRT Java

webgraph

- Graph are represented by bitstreams
- Uses dsi-bitstream for instantaneous codes, sux for pointers into the bitstream
- On a Software Heritage graph with 34 billion nodes and 517 billion arcs a BFS visit is three time faster than Java (3h)
- Unbelievably better ergonomics WRT Java
- Graphs have n nodes numbered in $[0\,.\,.\,n)$.

webgraph

- Graph are represented by bitstreams
- Uses dsi-bitstream for instantaneous codes, sux for pointers into the bitstream
- On a Software Heritage graph with 34 billion nodes and 517 billion arcs a BFS visit is three time faster than Java (3h)
- Unbelievably better ergonomics WRT Java
- Graphs have *n* nodes numbered in [0 . . *n*).
- Access to the graph structure happens by enumerating pairs given by a node (usize) and a (possibly labeled) successor list (IntoIterator<usize>)

webgraph

webgraph

- Compression happens by several techniques:

webgraph

- Compression happens by several techniques:
  - Gap compression: lists are turned into gaps encoded via instantaneous codes

webgraph

- Compression happens by several techniques:
  - Gap compression: lists are turned into gaps encoded via instantaneous codes
  - Reference: lists are partially copied from other nodes with similar successors

webgraph

- Compression happens by several techniques:
  - Gap compression: lists are turned into gaps encoded via instantaneous codes
  - Reference: lists are partially copied from other nodes with similar successors
  - Intervalization: consecutive successors are stored as intervals

webgraph

- Compression happens by several techniques:
  - Gap compression: lists are turned into gaps encoded via instantaneous codes
  - Reference: lists are partially copied from other nodes with similar successors
  - Intervalization: consecutive successors are stored as intervals
- Composition-based labeling

webgraph

- Compression happens by several techniques:
    - Gap compression: lists are turned into gaps encoded via instantaneous codes
    - Reference: lists are partially copied from other nodes with similar successors
    - Intervalization: consecutive successors are stored as intervals
- Composition-based labeling
- Lender- (rather than Iterator-) based architecture, as we need to return items depending on the lender state

webgraph

webgraph

- Permutation algorithms such as LLP provide node indices that improve compression significantly

webgraph

- Permutation algorithms such as LLP provide node indices that improve compression significantly
- Random-access enumeration of successor lists is lazy—lists of referenced nodes are never materialized

webgraph

- Permutation algorithms such as LLP provide node indices that improve compression significantly
- Random-access enumeration of successor lists is lazy—lists of referenced nodes are never materialized
- Compact id space and lack of allocated structures to represent edges makes the framework applicable to very large graphs

webgraph

- Permutation algorithms such as LLP provide node indices that improve compression significantly
- Random-access enumeration of successor lists is lazy—lists of referenced nodes are never materialized
- Compact id space and lack of allocated structures to represent edges makes the framework applicable to very large graphs
- An important change with respect to the Java version is that sequential enumeration of the arcs of a graph has no order guarantee

webgraph

- Permutation algorithms such as LLP provide node indices that improve compression significantly
- Random-access enumeration of successor lists is lazy—lists of referenced nodes are never materialized
- Compact id space and lack of allocated structures to represent edges makes the framework applicable to very large graphs
- An important change with respect to the Java version is that sequential enumeration of the arcs of a graph has no order guarantee
- Though there are marker traits to request that

webgraph

webgraph

- Basic trait: a SequentialLabeling

webgraph

- Basic trait: a SequentialLabeling

```
pub trait SequentialLabeling {
    type Label;
    type Lender<‘node>:
        for<'next> NodeLabelsLender<'next, Label = Self::Label>
    where
        Self: 'node;

    fn num_nodes(&self) -> usize;
    fn iter(&self) -> Self::Lender<'_>;
}
```

webgraph

- Basic trait: a SequentialLabeling

```
pub trait SequentialLabeling {
    type Label;
    type Lender<‘node>:
        for<'next> NodeLabelsLender<'next, Label = Self::Label>
    where
        Self: 'node;

    fn num_nodes(&self) -> usize;
    fn iter(&self) -> Self::Lender<'_>;
}
```

- A sequential graph is a SequentialLabeling with usize labels

webgraph

webgraph

- Random access:

webgraph

- Random access:

```
pub trait RandomAccessLabeling: SequentialLabeling {
    type Labels<'succ>:
        IntoIterator<Item = <Self as SequentialLabeling>::Label>
    where
        Self: 'succ;

   fn num_arcs(&self) -> u64;
   fn labels(&self, node_id: usize) ->
        <Self as RandomAccessLabeling>::Labels<'_>;
   fn outdegree(&self, node_id: usize) -> usize;
}
```

webgraph

- Random access:

```
pub trait RandomAccessLabeling: SequentialLabeling {
    type Labels<'succ>:
        IntoIterator<Item = <Self as SequentialLabeling>::Label>
    where
        Self: 'succ;

   fn num_arcs(&self) -> u64;
   fn labels(&self, node_id: usize) ->
        <Self as RandomAccessLabeling>::Labels<'_>;
   fn outdegree(&self, node_id: usize) -> usize;
}
```

- A random-access graph is a RandomAccessLabeling with usize labels

webgraph

webgraph

- We use lenders based on higher-rank trait bounds (Lender crate, based on Sabrina Jewson’s idea)

webgraph

- We use lenders based on higher-rank trait bounds (Lender crate, based on Sabrina Jewson's idea)
- GAT-based lenders require the lender to be 'static, which is a no-no for us

webgraph

- We use lenders based on higher-rank trait bounds (Lender crate, based on Sabrina Jewson's idea)
- GAT-based lenders require the lender to be 'static, which is a no-no for us

```
pub trait Lender {
    type Item<'this>
    where
        Self: 'a;

    fn next(&mut self) -> Option<Self::Item<'_>>;
}
```

webgraph

- We use lenders based on higher-rank trait bounds (Lender crate, based on Sabrina Jewson’s idea)

- GAT-based lenders require the lender to be 'static, which is a no-no for us

webgraph

- We use lenders based on higher-rank trait bounds (Lender crate, based on Sabrina Jewson’s idea)
- GAT-based lenders require the lender to be 'static, which is a no-no for us

```
pub trait Lending<'a, __ImplBound = &'a Self> {
    type Lend: 'a;
}

pub trait Lender: for<'a /* where Self: 'a */> Lending<'a> {
    fn next(&mut self) -> Option<<Self as Lending<'_>::Lend>;
}
```

webgraph

webgraph

- This not enough for us

webgraph

- This not enough for us
- For example, when iterating over a graph obtained by sorting source/target pairs (e.g., a transpose), we need the iterator on successors to modify the state of the lender

webgraph

- This not enough for us
- For example, when iterating over a graph obtained by sorting source/target pairs (e.g., a transpose), we need the iterator on successors to modify the state of the lender
- Suggestion by quinedot on the Rust Language Forum:

webgraph

- This not enough for us
- For example, when iterating over a graph obtained by sorting source/target pairs (e.g., a transpose), we need the iterator on successors to modify the state of the lender
- Suggestion by quinedot on the Rust Language Forum:

```
pub trait NodeLabelsLender<'a, __ImplBound = &'a Self>:
    Lender + Lending<'a, __ImplBound, Lend = (usize, Self::IntoIterator)>
{
    type Label;
    type IntoIterator: IntoIterator<Item = Self::Label>;
}
```

Performance

Graph	Nodes	Arcs	Avg. Degree	b/arc	Size (comp.)
dblp-2010	326K	1.6M	4.95	6.78	1.4MB
hollywood-2011	2M	229M	105.00	4.89	140MB
enwiki-2023	4.2M	101M	24.93	13.55	267MB
in-2004	41M	1.1G	27.87	1.41	250MB
webbase-2001	118M	1G	8.63	2.78	399MB
twitter-2010	41M	1.4G	35.25	13.90	2.5GB
eu-2015	1G	92G	85.74	1.19	13GB
swh-2023	34G	491G	14.38	3.07	176GB

	Java	**Rust**	**speedup**	**Java**	**Rust**	**speedup**
Graph	*Random access* (ns/arc)			*BFS visit* (ns/node)		
dblp-2010	96	50	× 1.92	604	220	× 2.75
hollywood-2011	51	27	× 1.88	7520	2620	× 2.87
enwiki-2023	61	31	× 1.97	1450	734	× 1.98
in-2004	70	37	× 1.89	735	369	× 1.99
webbase-2001	114	73	× 1.56	665	322	× 2.07
twitter-2010	73	38	× 1.92	2650	1270	× 2.09
eu-2015	24	17	× 1.41	1580	971	× 1.63
swh-2023	104	47	× 2.21	1140	359	× 3.18

Conclusion

Conclusion

- This journey started in April 2023 and it has been a blast

Conclusion

- This journey started in April 2023 and it has been a blast
- It was very surprising for me that we were able to port and, in fact, massively improve the design and implementation of a codebase that had been accumulating for 20 years

Conclusion

- This journey started in April 2023 and it has been a blast
- It was very surprising for me that we were able to port and, in fact, massively improve the design and implementation of a codebase that had been accumulating for 20 years
- Graph traversals, graph algorithms, etc., are on their way

Conclusion

- This journey started in April 2023 and it has been a blast
- It was very surprising for me that we were able to port and, in fact, massively improve the design and implementation of a codebase that had been accumulating for 20 years
- Graph traversals, graph algorithms, etc., are on their way
- We would like to thank Valentin Lorentz at Software Heritage for a lot of improvements and suggestions (and for being our guinea pig)

Conclusion

- This journey started in April 2023 and it has been a blast
- It was very surprising for me that we were able to port and, in fact, massively improve the design and implementation of a codebase that had been accumulating for 20 years
- Graph traversals, graph algorithms, etc., are on their way
- We would like to thank Valentin Lorentz at Software Heritage for a lot of improvements and suggestions (and for being our guinea pig)
- And we would like to thank the community of the Rust Language Forum, without which none of this would have ever happened

Conclusion

- This journey started in April 2023 and it has been a blast
- It was very surprising for me that we were able to port and, in fact, massively improve the design and implementation of a codebase that had been accumulating for 20 years
- Graph traversals, graph algorithms, etc., are on their way
- We would like to thank Valentin Lorentz at Software Heritage for a lot of improvements and suggestions (and for being our guinea pig)
- And we would like to thank the community of the Rust Language Forum, without which none of this would have ever happened
- And my students, without whom we would be far behind schedule

Questions?