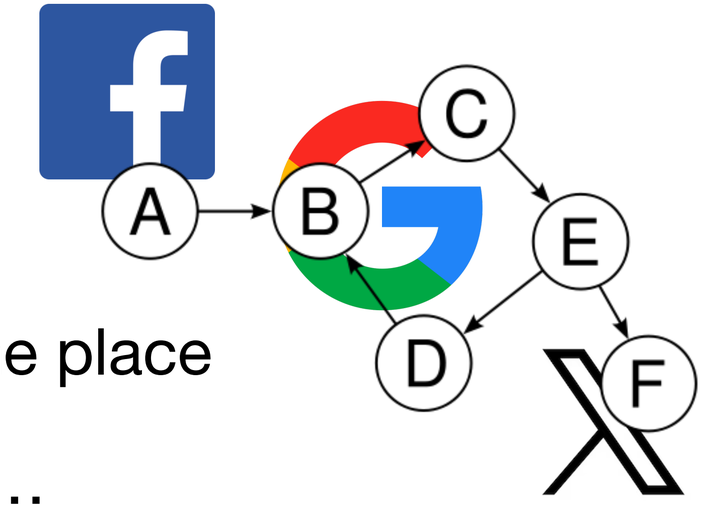
Stuff in Rust

Sebastiano Vigna
Università degli Studi di Milano

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

- 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 (>1700 references)
- In 2011 news went around the world: Fac
  separation
- The measurement was performed at Fac
  using WebGraph (at that time, 721M nod
- Common Crawl distributes data using W

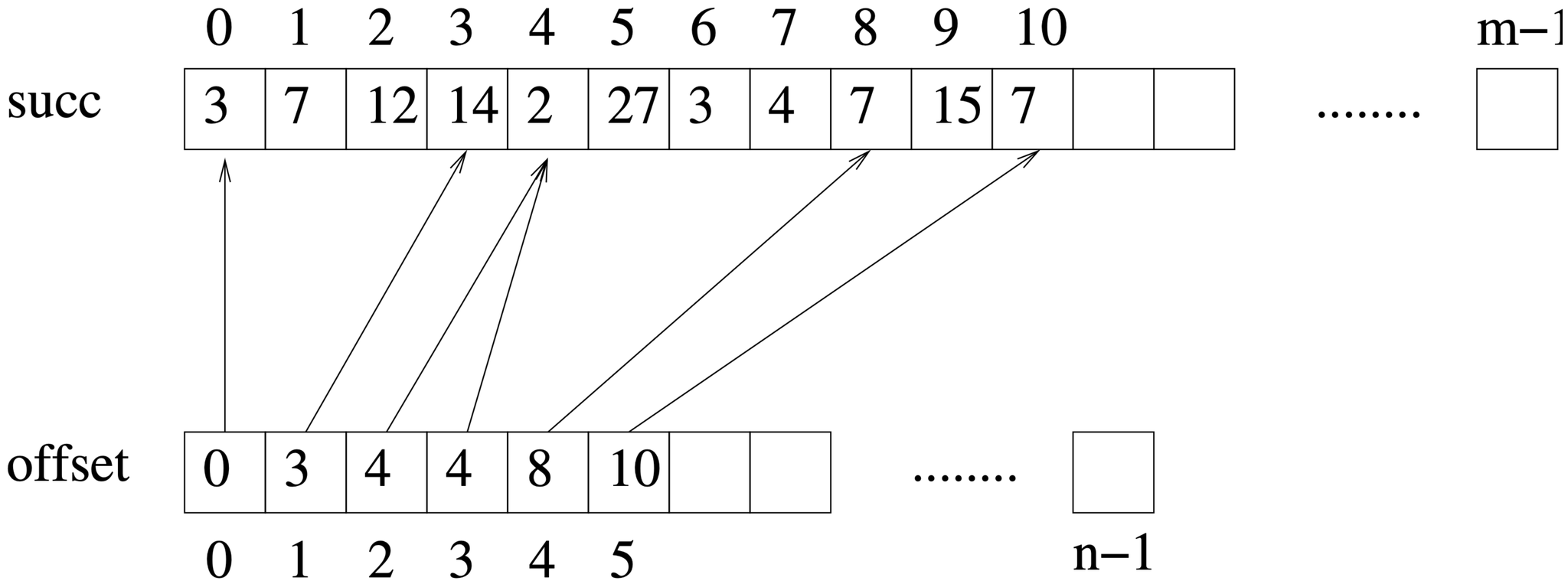
Software Heritage
THE GREAT LIBRARY OF SOURCE CODE

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
- 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

(Web)Graph Compression

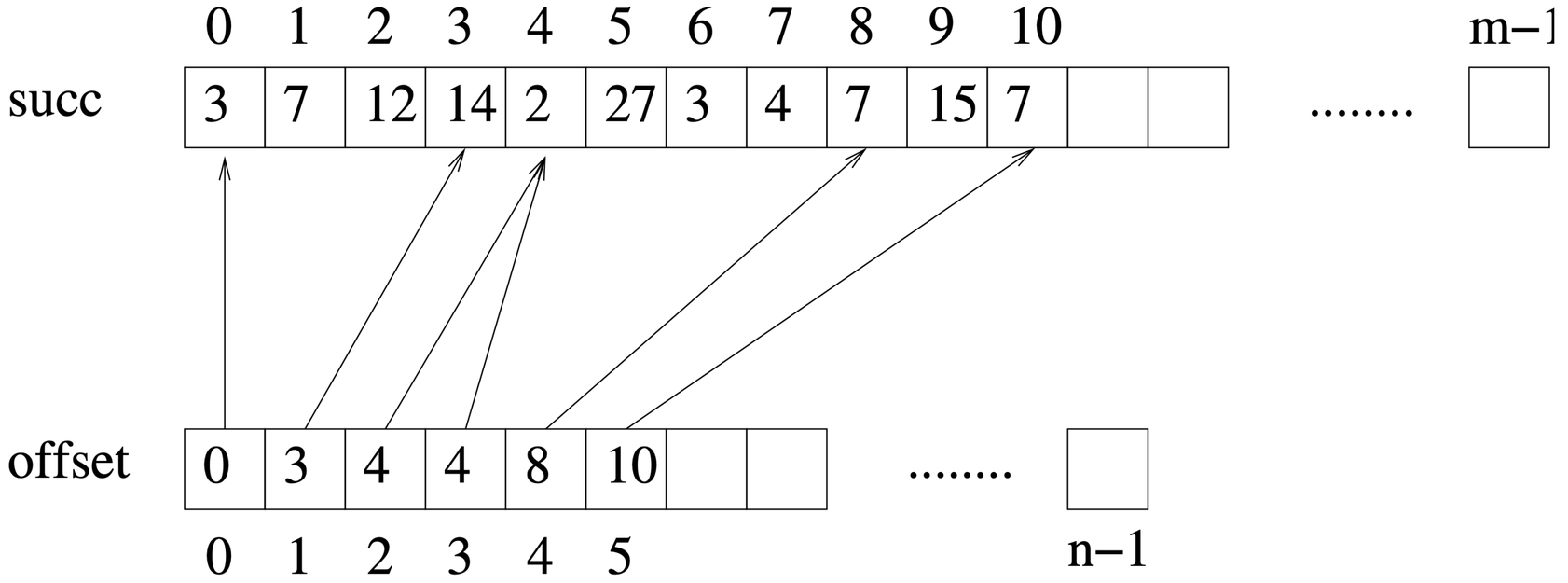
- Immutable storage for large graphs
- Trivial: CRS (Compressed Sparse Row)



Superfast Primer on Instantaneous Codes

- Unary: 1, 01, 001, 0001, 00001, …
- Elias ɣ: write $x > 0$ in binary, strip the highest bit, prefix with length in unary
  - $1_{10} \rightarrow 1_2 \rightarrow \varepsilon \rightarrow 1{\cdot}\varepsilon \rightarrow 1$
  - $2_{10} \rightarrow 10_2 \rightarrow 0 \rightarrow 01{\cdot}0 \rightarrow 010$
  - $3_{10} \rightarrow 11_2 \rightarrow 1 \rightarrow 01{\cdot}1 \rightarrow 011$
  - $4_{10} \rightarrow 100_2 \rightarrow 00 \rightarrow 001{\cdot}00 \rightarrow 00100$
- Elias δ: same as above but length in ɣ

Gap Compression



- 3, 7, 12 → 3 , 7 – 3, 12 – 7 → 011 · 00100 · 00101
- Offsets are now into a bit stream

Reference Compression

- Pages can have a significant overlap in their link
- E.g., from the same site
- We can copy some successors from a previous node in a small window and just write the remaining successors
- How maximize the compression?
- URL sorted order
- Clustering (LLP)
- This also improves gaps
- Dense web graphs: ≈ 1b / arc (!)

WebGraph in Rust

- 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

Random Access

- 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

Labels on Arcs

- We have a zip operator for labelings
- If you zip a graph and a labeling, you get a labeled graph
- Wrapper type UnitGraph: given a graph, gives you a labeled graph where every arc is labeled with the unit type ()
- Projections: Left and Right give you the sides of the zip
- Left(UnitGraph(G)) = G (i.e., $G \times 1 \cong G$ via left projection)
- The compiler understands this—working on G or Left(UnitGraph(G)) has the same performance!
- You can zip labels and get composite labels

Lenders

- Lenders are iterators whose returned item take an exclusive reference to the emitter
- Essential for stateful iterators returning (part of) the inner state

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

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

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

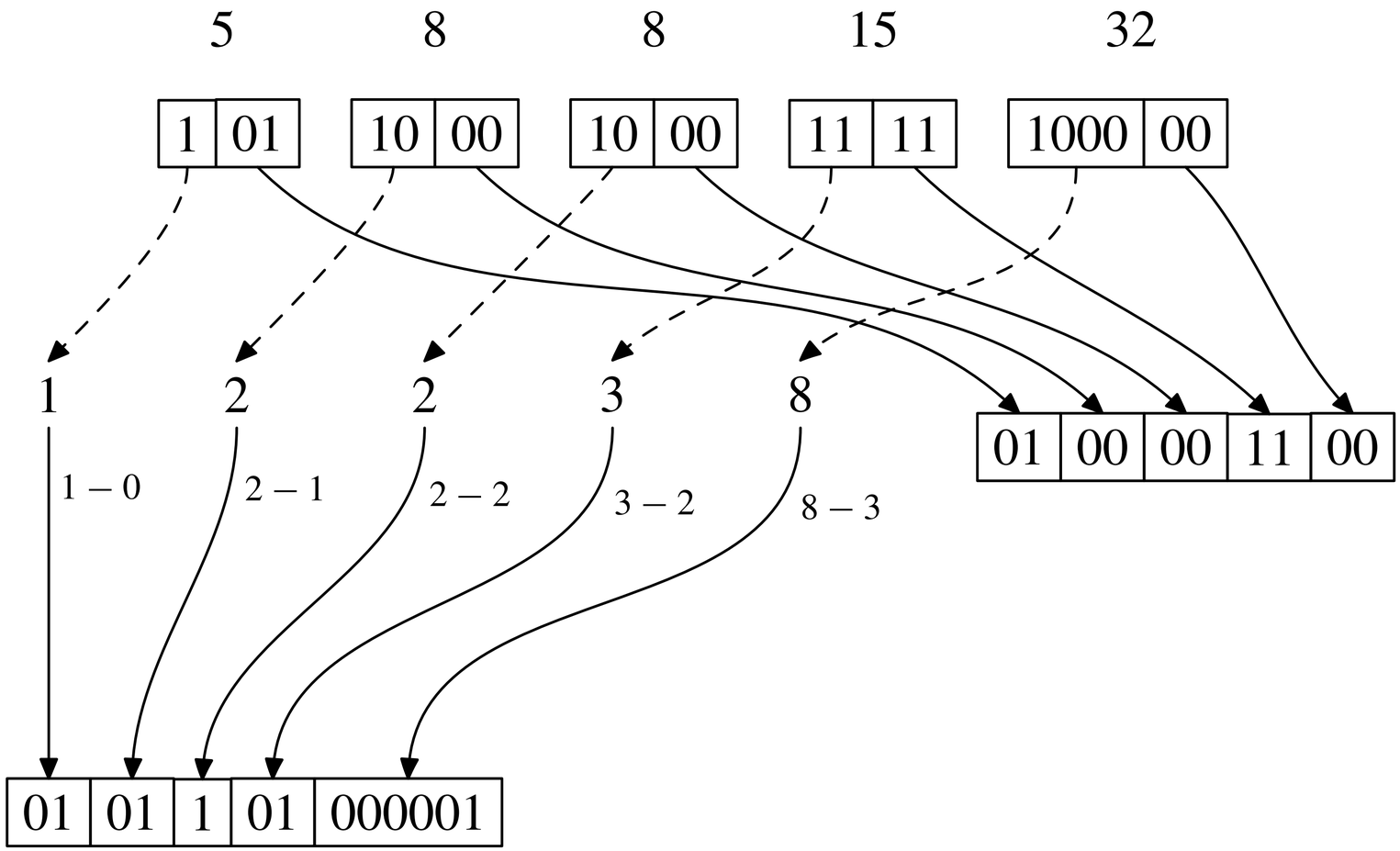
- Mix-and-match of arbitrary rank and selection structures
- We use intensively the ambassador crate to delegate all rank, selection, and bit-vector access traits, so you can write

```
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

Elias–Fano

- Given n and u we have a monotone sequence $0 \leq x_0, x_1, x_2, \ldots, x_{n-1} < u$
- Store the lower $\ell = \log(u / n)$ bits explicitly
- Store the upper bits as a sequence of unary coded gaps (0^k1 represents k)
- We use at most $2 + \log(u / n)$ bits per element
- Close to the succinct bound: quasi-succinct!
- (Less than half a bit away, as Elias proves)



$5, 8, 8, 15, 32 < u = 36, \ell = 2$

Some Very Personal Thoughts


Rust when I have an atom of difference between my type and the expected type
Python when I cast a float into an unsigned Toyota Yaris 2023

Your Relationship With the Compiler

Jarrod Overson

- Programming in Rust is like being in an emotionally abusive relationship. Rust screams at you all day, every day, often about things that you would have considered perfectly normal in another life. Eventually, you get used to the tantrums. They become routine. You learn to walk the tightrope to avoid triggering the compiler's temper. And just like in real life, those behaviors changes stick with you forever.
- Emotional abuse is not generally considered a healthy way to encourage change, but it does effect change nonetheless.
- I can't write code in other languages without feeling uncomfortable when lines are out of order or when return values are unchecked. I also now get irrationally upset when I experience a runtime error.

But…

- Some people find error messages they can't ignore more annoying than wrong results, and, when judging the relative merits of programming languages, some still seem to equate “the ease of programming” with the ease of making undetected mistakes.
- Edsger W. Dijkstra, 1978, On the foolishness of "natural language programming” (EWD 667)

Rust is Strict

- Rust forces an in-world-vs.-out-world philosophy
- Inside, everything is beautiful: no UB (e.g., integer operations)
- Outside, *hic sunt leones*
- There are controlled membranes at the entrance and exit, and opt-outs
- Once you're in, it's paradise
- But the strictness *has a cognitive cost*
- Example: many, many types of strings: String, str, Box<str>, OsStr, OsString, CString, Cstr, …
- unsafe { } is the entrance / exit of the pointer world

Rust Controls Data Access

- In Rust, you share many references (&), or you have an exclusive reference (&mut)
- Frank McSherry (very intuitive) view: you have a read-write lock on every piece of data
- The SharingXorMutation opens the way to great optimizations
- Compulsory move semantics frees you from memory management
- At the same time, there’s a cognitive cost
- SharingXorMutation & reference validity are enforced by the borrow checker
- Many levels of opt-out: Cell (just give up on SharingXorMutation), RefCell (an actual read/write lock!), UnsafeCell, Rc, etc.

Rust is Wonderful

- The borrow checker is a wonder of programming languages
- The Rust compiler is essentially the first compiler using a theorem prover to show your programs do work
- (essentially because, e.g., some languages detect uninitialized vars)
- This strictness makes your program work
- But borrow checking is Turing-complete, and obviously Rust prefers correctness (and no false positives) to completeness (and no false negatives)
- There are also shady areas: where the BC doesn’t go, and it knows; where the BC doesn’t go, but doesn’t know; and where the BC can’t go
- Also, very difficult to formalize (one compiler!)

Rust Has Special Needs

- Ownership is essential to Rust, and uniquely embodied in the BC
- Since it’s so unique, sometimes concepts are confusing
- Example: Box<[T]> and &[T] *have exactly the same memory representation*: a pointer and an integer
- Their only difference is ownership
- This was for me initially a problem: many construct in Rust exist *because* Rust does things in a different way
- The fact the Box::new is type-dependent doesn’t help
- Rust literature is very mathematical in that it tends to hide the motivation (“rabbit out of a hat”)

Why Did I Come to Rust

- Maybe surprising, my only reason was to get rid of Java (GC, in particular)
- Our software manipulates very large data structure and 20 years ago 2B-element arrays were kinda ok ,today it’s ridiculous
- The garbage collector gets in the way and bends your software
- So my way in was move semantics and automatic memory management
- Little I knew I would have met my dream language
- As Ryan Levick put it introducing Felix Clock at RustFest 2016:

  “Something very unique about Rust is the ability to really have your head in the cloud and thinking of things like category theory and advanced type systems, but at the same time having your toes wrestle around inside the bits.”

The Only Correct Syntax Choices

- Rust syntax is amazingly well-designed and orthogonal (for a mathematician)
- (More correctly: amazingly aligned to the syntax *I'd like*)
- a..b is [a..b) (Knuth's notation for intervals / EWD 831)
- Projections of tuples starts at zero (EWD 831)
- If you know some category theory, the terminal set is the set towards which there's a unique function, and that's the singleton
- By tradition people uses the empty pair () for the element of the singleton, because $\prod_{\varnothing} = \{ () \}$ and Rust does the same
- Everything as a value, even loops
- const _ : () = assert!(…)
- But: https://github.com/dtolnay/rust-quiz

Stunning Zero-Cost Abstraction Capabilities

- The type system and compiler have a stunning combinatorial understanding of types
- You can reason very mathematically about types
- My favorite example from WebGraph: the unit type ()
- If you know some category theory, the terminal set is the set towards which there's a single function, and that's the singleton
- By tradition people uses the empty pair () for the element of the singleton, because $\prod_{\varnothing} = \{ () \}$ and Rust does the same
- A graph is (V, A) with $A \subseteq V \times V$. A graph labeled on L is (V, A) with $A \subseteq V \times L \times V$.

Stunning Zero-Cost Abstraction Capabilities

- We can see a graph G = (V, A) as a ()-labelled graph G’ = (V, B) with $B \subseteq V \times \{ () \} \times V$.
- So if we have an algorithm on *labeled* graphs we can run it on G’
- E.g., transposition: (x, l, y) → (y, l, x)
- So now we have a transposed graph $(G')^T = (V, T)$ with $T \subseteq V \times \{ () \} \times V$.
- If we project away the label, you get $U \subseteq V \times V$ and (V, U) is the transpose of G
- The Rust compiler understands all this
- So we have just a labelled version of everthing; to apply to normal graphs, we first “paint” everything with (), run the algorithm, and project it away from the result
- Same speed (everything cancels away)
- Cognitive cost: very complicated types sometimes

Conditional Trait Implementations

- The type system is based on Haskell *type classes*
- This is a further level of abstraction and flexibility from Eiffel (1986) or Go (2008) interfaces
- Conditional implementation does wonders in organizing logically features
- impl<T: bound> Trait for T {} implement Trait given that the bound is true
- Even better: impl<T: bound> Trait for Struct<T> {}
- In Java you have to resort to runtime reflection or “optional” methods
- Cognitive load: sometimes it is really difficult to understand why you can’t call a certain method

Sequences Done Right

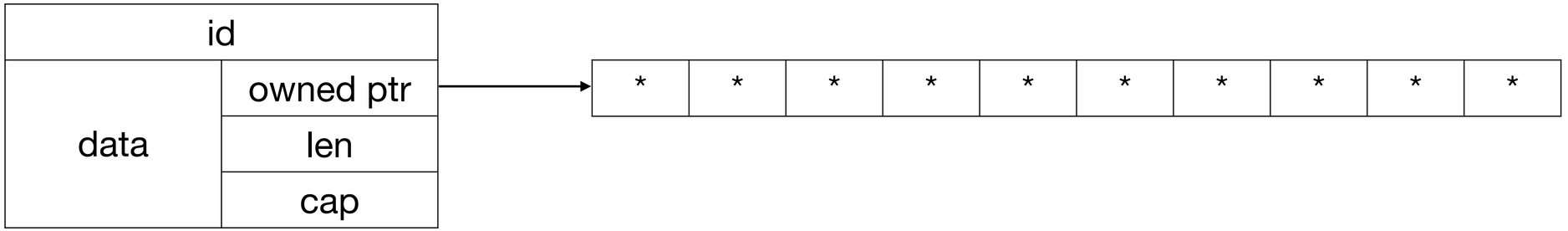
- The sequence is the most used structure in computer science
- Not, generically lists, but sequences of contiguous elements
- Rust has an incredibly well thought out sequence ecosystem with several different flexible types with different purposes
- Arrays are fixed-length and sized
- Slices are variable-length and unsized
- Reference to slices access sequences by reference
- Vectors hold (own) variable-length sequences
- Boxed slices own fixed-length sequences (save one word WRT vectors, no excess space)
- But *all of them* are AsRef<[T]>

Sequences + Conditional Implementation

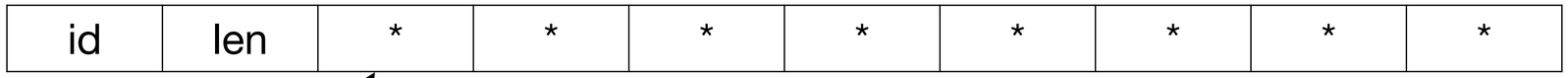
- You can write methods and implement trait based on AsRef<[T]> instead of a specific sequence implementation
- AsRef<[T]> act as “the idea of a sequence”
- It’s the typesafe version of pointer + length of C programming
- For example: Struct<A: AsRef<[usize]>>
- We can impl<A: AsRef<[usize]>> Struct<A> { … } and this will work for Struct<Vec<usize>>, Struct<[usize; n]>, etc.
- And for Struct<&[usize]>! The structure might refer to preallocated memory (zero-copy deserialization)

Example

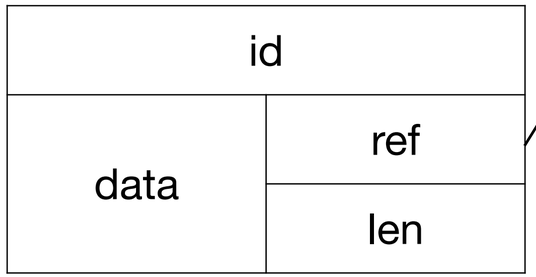
Construction time



Serialized



ε-deserialization



Iterators Done Right

- If you used iterators in other languages, in particular Java and C#, you know how many things can go wrong when designing iterators
- Rust iterators are fully lazy—no hasNext crap and no boolean checks for exhaustion
- More importantly, by contract they give undefined results after the first None—that's the key to efficient lazy implementations
- If you want idempotence (once None, always None) just .fuse() the iterator
- If you want peekability, just .peekable() the iterator, etc.
- You pay for what you use

Conclusions

- I wasn’t having this fun programming since decades
- Coalescing 50 years of research in programming languages (ML ≈ early ‘70s) in a language with tight control on resources was definitely a brilliant idea (not Hoare’s original idea, tho, probably)
- And, yes, sometimes it’s frustrating, but once you’re in the mindset you can be incredibly productive
- Great match with AI and “vibe coding” as the compiler can catch problems at an early stage
- Most common issue: the rabbit hole of abstraction
- It’s so easy and has no cost, so you end up exaggerating a bit