a different serde

serde:
SERialize/DEserialize

two parts:

frontend:
for rust types

backend:
the wire format

everyone uses serde

postcard uses serde

YOU should keep using serde

serde has some problems

sometimes I wonder if I could do better...

...at least for what postcard needs.

how serde works:

1

The serde data model: 29 different “types”

primitive types like `u8`, `i16`, `f32`, `...`

arrays like

`&str`, `[T]`, `[u8]`, ...

composite types like
struct, tuples

1

enums and their variants

1

the frontend turns Rust types into Data Model types

the backend turns Data Model types into bytes

the visitor pattern

types are given a
Serializer

each Data Model Type has a serializing method

```
s.serialize_u8(u8)
s.serialize_str(&str)
...
```

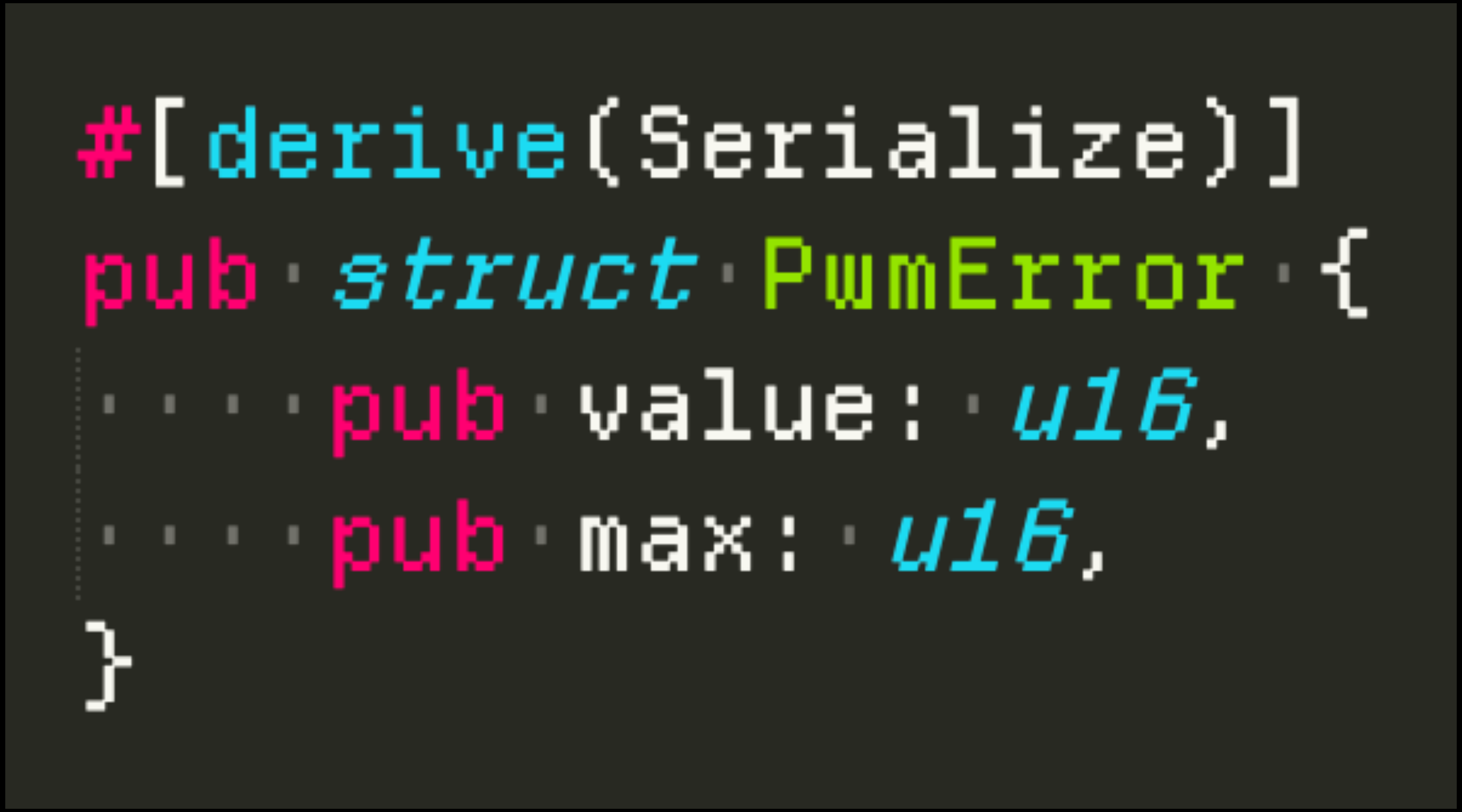
this drives the backend

this code is *usually* derived

3

```
#[derive(Serialize)]
```

3



```rust
#[derive(Serialize)]
pub struct PwmError {
    pub value: u16,
    pub max: u16,
}
```

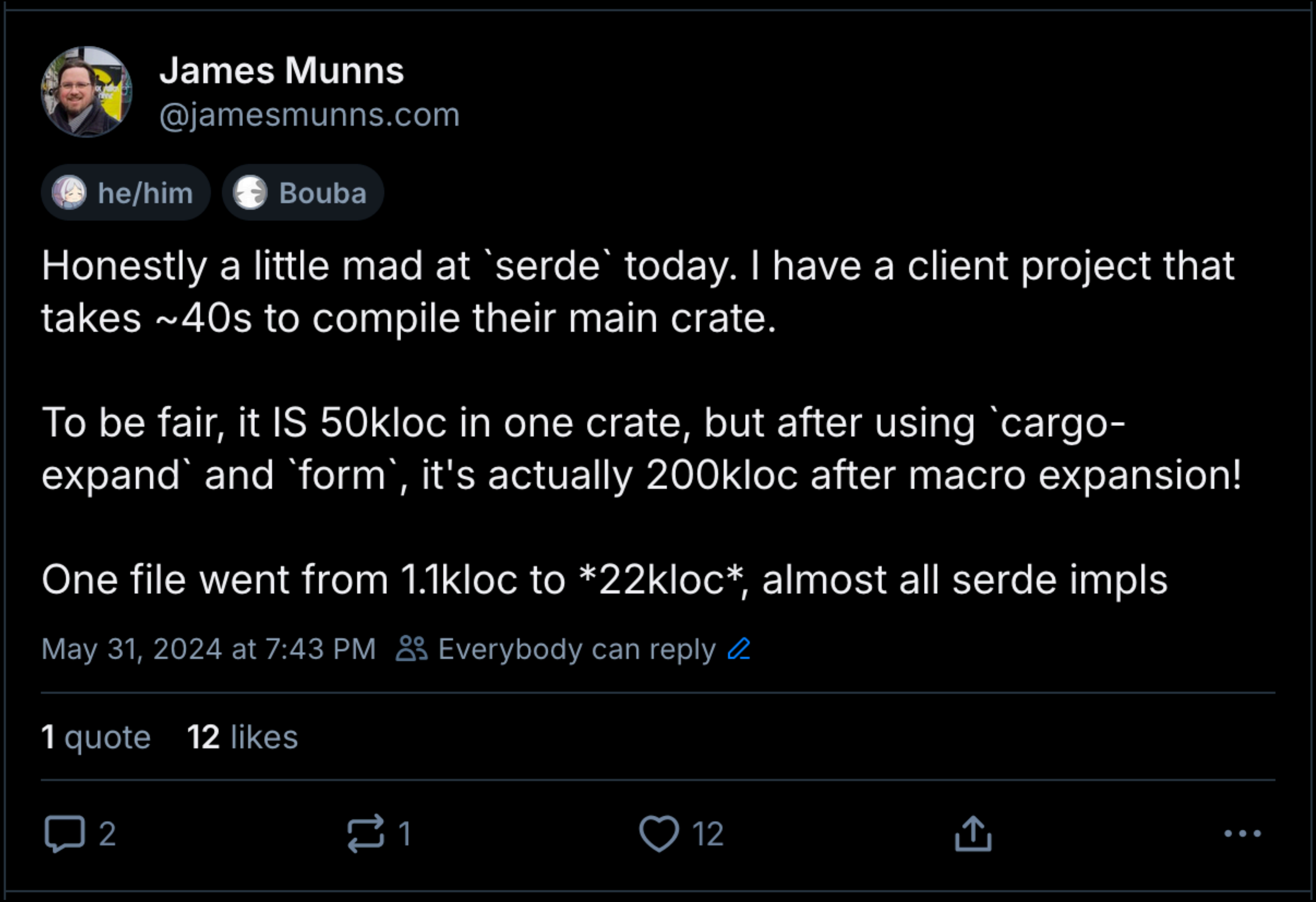
3

```
#[doc(hidden)]
#[allow(non_upper_case_globals, unused_attributes, unused_qualifications)]
const _: () = {
    #[allow(unused_extern_crates, clippy::useless_attribute)]
    extern crate serde as _serde;
    #[automatically_derived]
    impl _serde::Serialize for PwmError {
        fn serialize<__S>(
            &self,
            __serializer: __S,
        ) -> _serde::__private::Result<__S::Ok, __S::Error>
        where
            __S: _serde::Serializer,
        {
            let mut __serde_state = _serde::Serializer::serialize_struct(
                __serializer,
                "PwmError",
                false as usize + 1 + 1,
            )?;
            _serde::ser::SerializeStruct::serialize_field(
                &mut __serde_state,
                "value",
                &self.value,
            )?;
            _serde::ser::SerializeStruct::serialize_field(
                &mut __serde_state,
                "max",
                &self.max,
            )?;
            _serde::ser::SerializeStruct::end(__serde_state)
        }
    }
};
```

there are some problems

this generates a LOT of code

Self-Directed Research



James Munns
@jamesmunns.com

he/him Bouba

Honestly a little mad at `serde` today. I have a client project that takes ~40s to compile their main crate.

To be fair, it IS 50kloc in one crate, but after using `cargo-expand` and `form`, it's actually 200kloc after macro expansion!

One file went from 1.1kloc to *22kloc*, almost all serde impls

May 31, 2024 at 7:43 PM Everybody can reply

1 quote **12** likes

2 1 12

there’s a LOT of monomorphization and inlining

the visitor pattern is recursive

for deserialization: data returned by value for postcard: more flexibility than we need how else do we approach this problem?

postcard-forth

https://github.com/jamesmunns/postcard-forth/

EXPERIMENTAL

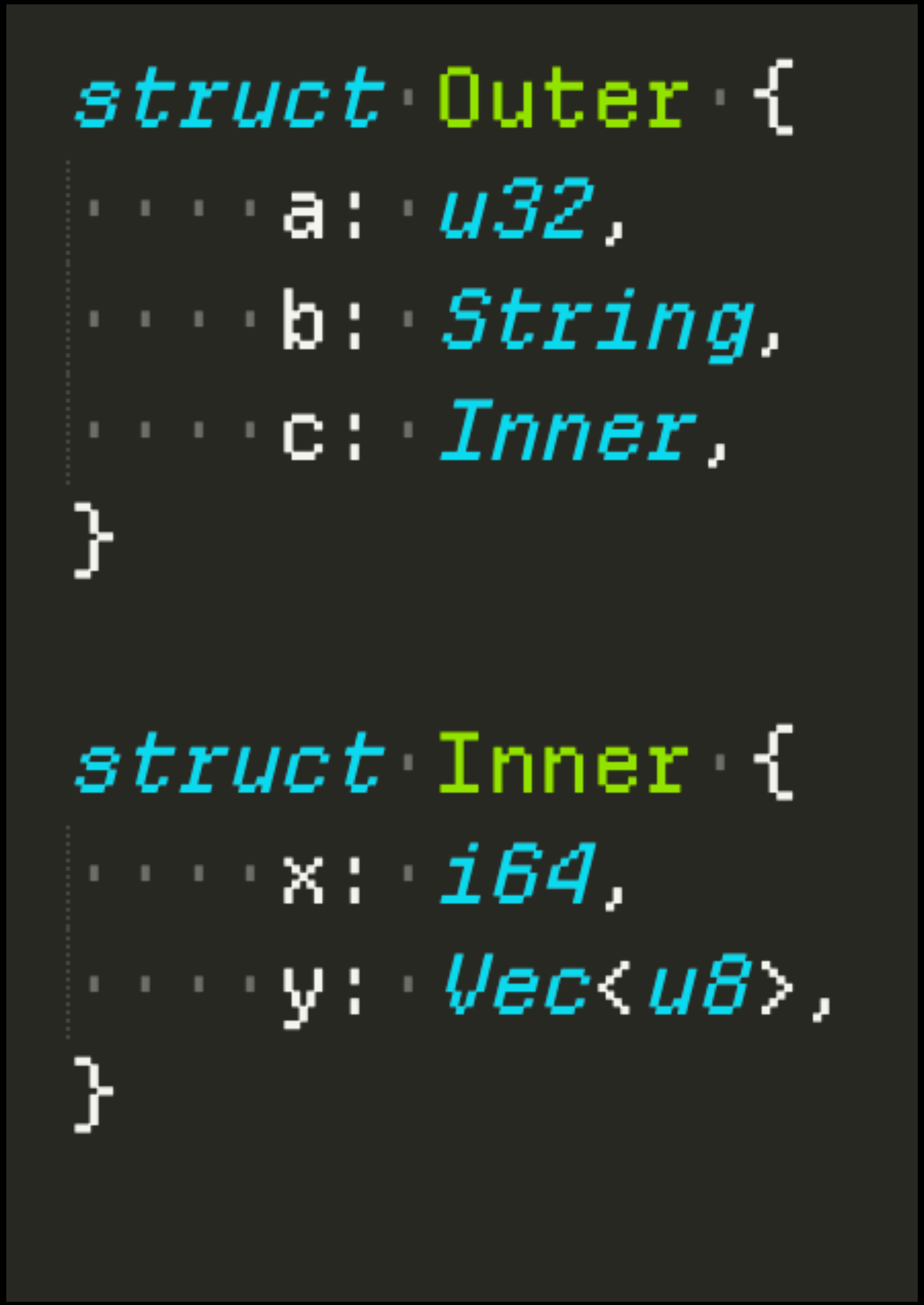
how postcard-forth works:

1

keep (mostly) the same data model

a much simpler derive macro

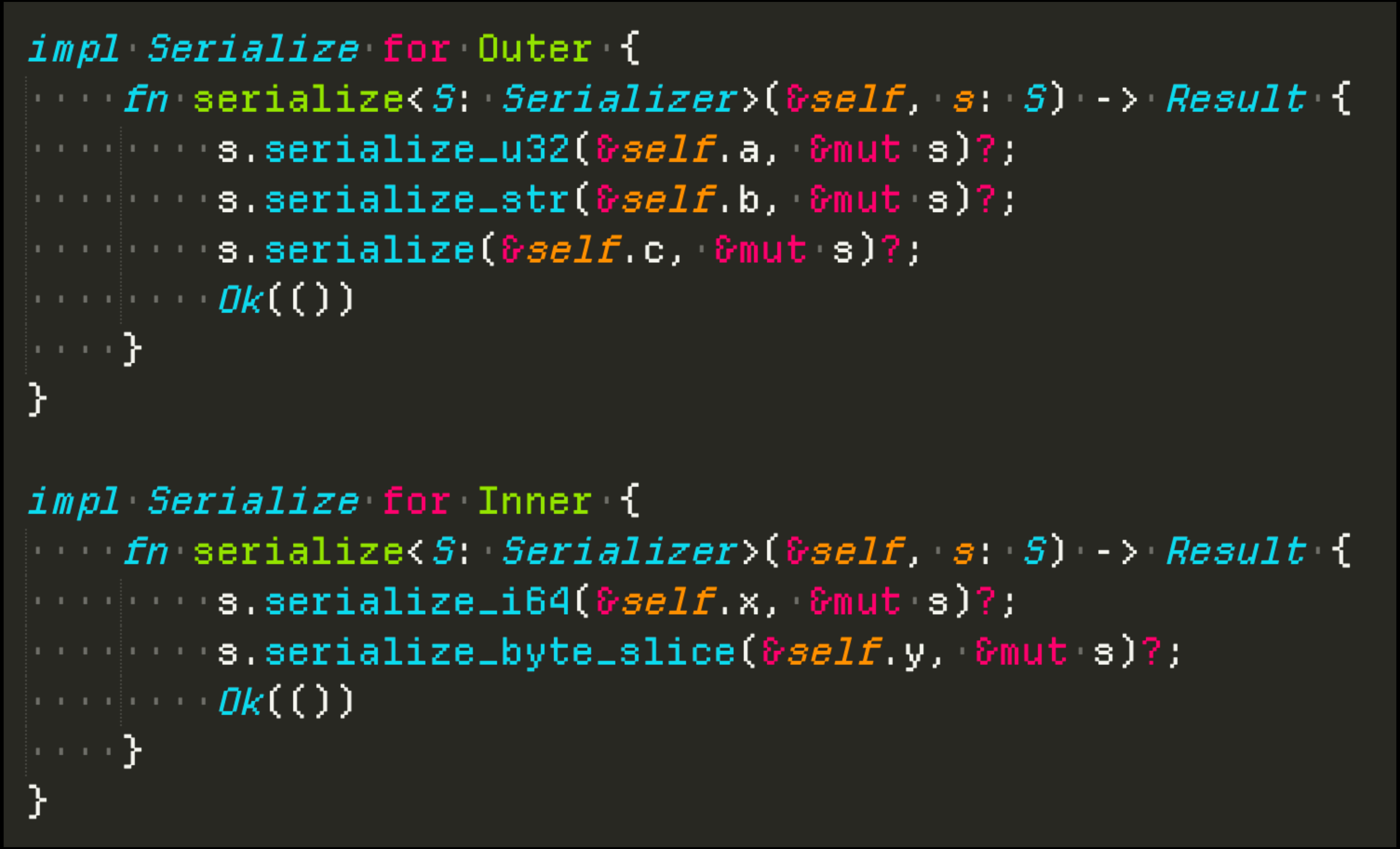
ONLY generate a list of field offsets and function pointers for this data:



```
struct Outer {
    a: u32,
    b: String,
    c: Inner,
}

struct Inner {
    x: i64,
    y: Vec<u8>,
}
```

instead of ~this:



```rust
impl Serialize for Outer {
    fn serialize<S: Serializer>(&self, s: S) -> Result {
        s.serialize_u32(&self.a, &mut s)?;
        s.serialize_str(&self.b, &mut s)?;
        s.serialize(&self.c, &mut s)?;
        Ok(())
    }
}

impl Serialize for Inner {
    fn serialize<S: Serializer>(&self, s: S) -> Result {
        s.serialize_i64(&self.x, &mut s)?;
        s.serialize_byte_slice(&self.y, &mut s)?;
        Ok(())
    }
}
```

generate ~this:

2

```rust
impl Inner {
    const SER_LIST: &[(usize, fn(*const (), &mut OutStream))] = &[
        (offset_of!(Inner, x), ser_i64),
        (offset_of!(Inner, y), ser_deref_slice_u8::<Vec<u8>>),
    ];
}

impl Outer {
    const SER_LIST: &[(usize, fn(*const (), &mut OutStream))] = &[
        (offset_of!(Outer, a), ser_u32),
        (offset_of!(Outer, b), ser_string),
        (offset_of!(Outer, c), ser_ty::<Inner>),
    ];
}
```

or maybe even ~this:

```
impl Inner {
    const SER_LIST: &[(usize, fn(*const (), &mut OutStream))] = &[
        (offset_of!(Inner, x), ser_i64),
        (offset_of!(Inner, y), ser_deref_slice_u8::<Vec<u8>>),
    ];
}

impl Outer {
    const SER_LIST: &[(usize, fn(*const (), &mut OutStream))] = &[
        (offset_of!(Outer, a), ser_u32),
        (offset_of!(Outer, b), ser_string),
        (offset_of!(Outer, c) + offset_of!(Inner, x), ser_i64),
        (offset_of!(Outer, c) + offset_of!(Inner, y), ser_deref_slice_u8::<Vec<u8>>),
    ];
}
```

3

turn ser/de into a "stack machine"

the input is the list of offsets and functions

the output is the stream of serialized bytes

this is basically an interpreter

and forth says "data is code"

why is this good?

there is only ever
ONE ser/de machine

it's an explicit stack machine: bounded depth initial testing shows it's USUALLY faster

Serialize / deserialize speed and size

Crate	Serialize	Deserialize	Size	Zlib	Zstd	Zstd Time
postcard 1.0.8	67.71%	90.57%	100.00%	100.00%	100.00%	100.00%
postcard_forth 0.1.0	100.00%	100.00%	100.00%	100.00%	100.00%	97.91%

Serialize / deserialize speed and size

Crate	Serialize	Deserialize	Size	Zlib	Zstd	Zstd Time
postcard 1.0.8	100.00%	56.44%	100.00%	100.00%	100.00%	99.42%
postcard_forth 0.1.0	84.07%	100.00%	100.00%	100.00%	100.00%	100.00%

Serialize / deserialize speed and size

Crate	Serialize	Deserialize	Size	Zlib	Zstd	Zstd Time
postcard 1.0.8	67.11%	86.19%	100.00%	100.00%	100.00%	99.91%
postcard_forth 0.1.0	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

Serialize / deserialize speed and size

Crate	Serialize	Deserialize	Size	Zlib	Zstd	Zstd Time
postcard 1.0.8	58.05%	67.48%	100.00%	100.00%	100.00%	100.00%
postcard_forth 0.1.0	100.00%	100.00%	100.00%	100.00%	100.00%	99.82%

initial testing shows it's USUALLY smaller (code and binary)

case	types	text size	input lines	expanded lines	ttl time	crate time
baseline	0	10584	0	53	13.3s	0.33s
postcard-serde	128	164000	2664	47240	20.1s	7.03s
postcard-forth	128	111220	2664	15931	16.4s	3.14s
postcard-serde	512	640456	10244	181583	39.7s	25.95s
postcard-forth	512	395712	10244	60232	24.6s	11.82s
postcard-forth (inlined)	512	389944	10244	79471	24.8s	11.78s
postcard-serde (no enums)	512	550012	8248	72610	33.1s	19.7s
postcard-forth (no enums)	512	223492	8248	20594	19.6s	6.82s
postcard-serde (onlyprims)	512	610800	10248	177647	45.6s	32.4s
postcard-forth (onlyprims)	512	295704	10248	59645	22.3s	9.63s

for deser:
we could do it
totally *in-place*

why is this NOT good?

requires ANOTHER derive trait for every type

the ser/de engine is
WILDLY UNSAFE

manually implementing the traits is WILDLY UNSAFE we still need SOME code to get to Data Model Types especially for iterators, or non-deref types

we need a whole RFC for new enum abilities

https://github.com/rust-lang/rfcs/pull/3727 it's **not** quite a linear program...

...we need "branching" for enums

...we need "loops"
for slices and iters

is it worth it?

no idea yet.