

Abusing the Type System for Fun and for Profit



Buh!

By Conrad Ludgate

Featuring very scary software mistakes, and even scarier type signatures…

Why do we use Rust?

Is it because of the cute crab?

“Modern APIs, high-level features, and C-speed”

“safety by default, generally not a pain to write”

“It makes my friends think I’m smart”

“The crab is kinda cute tho…”

WRONG

You use Rust because of the type system.

But what is a type system?

The Oxford English Dictionary defines it as…

Thank you for visiting Oxford English Dictionary

To continue reading, please sign in below or purchase a subscription

But what is a type system?

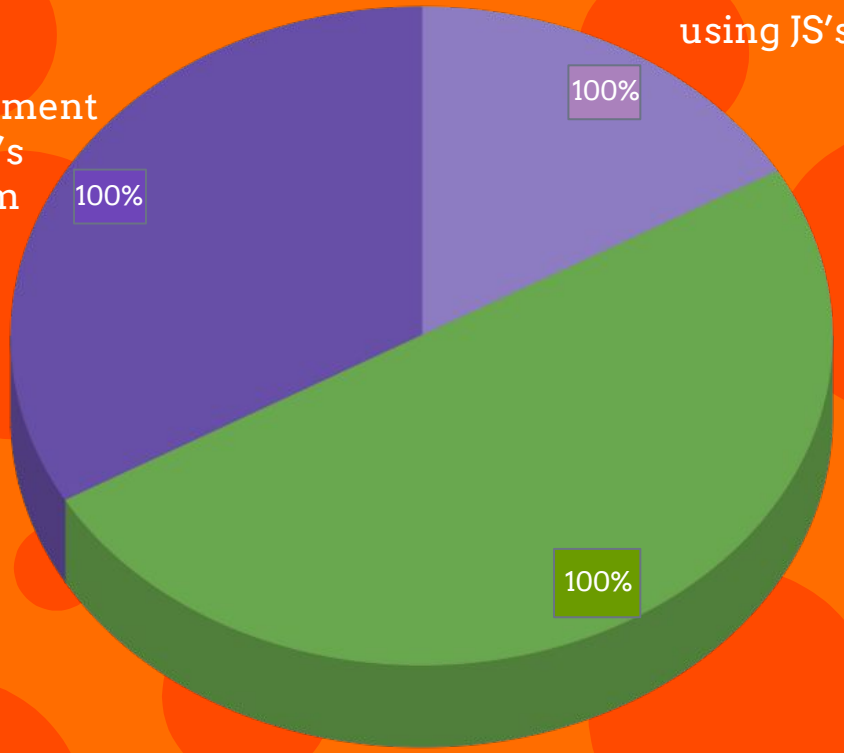
The type system categorises data.

The type system should not make use sad.

They often do.

Your sadness using JS's

Your enjoyment using Rust's type system



Your sadness looking at this pie chart

89,526,124

Types were diagnosed with [object Object] last year.

Payment handling

Are you sure your payments are only executed once, and that you don't double-charge your customers?

Payment lifecycle

Send request

Your customer requests to purchase a product

Process request

You handle the request, and send the notification to the bank

Funds acquired

You take your customers money. Happy days

Because stuff always breaks, you stick some retries here… oops… you just charged your customer twice…

Naive solution: state machine

```rust
pub enum PaymentState {
    Initiated,
    Processed,
    Completed,
}

impl PaymentState {
    pub fn update(&mut self, event: Event) -> Result<()> {
        match (self, event) {
            (Self::Initiated, Event::Processed) => {
                *self = Self::Processed;
                acquire_funds();
            }
            (Self::Processed, Event::Funds) => {
                *self = Self::Completed,
            }
            (_, _) => bail!("invalid event")
        }
    }
}
```

```rust
pub enum PaymentState {
    Initiated,
    Processed,
    Completed,
}

impl PaymentState {
    pub fn update(&mut self, event: Event) -> Result<()> {
        match (self, event) {
            (Self::Processed, Event::Processed) => {
                *self = Self::Processed;
                acquire_funds();
            }
            (Self::Initiated, Event::Processed) => {
                // out of order sequence, that's ok
                Ok(())
            }
            (Self::Processed, Event::Funds) => {
                *self = Self::Completed,
            }
            (_, _) => bail!("invalid event")
        }
    }
}
```

The solution is simple

Affine types → Can only be used → At most once

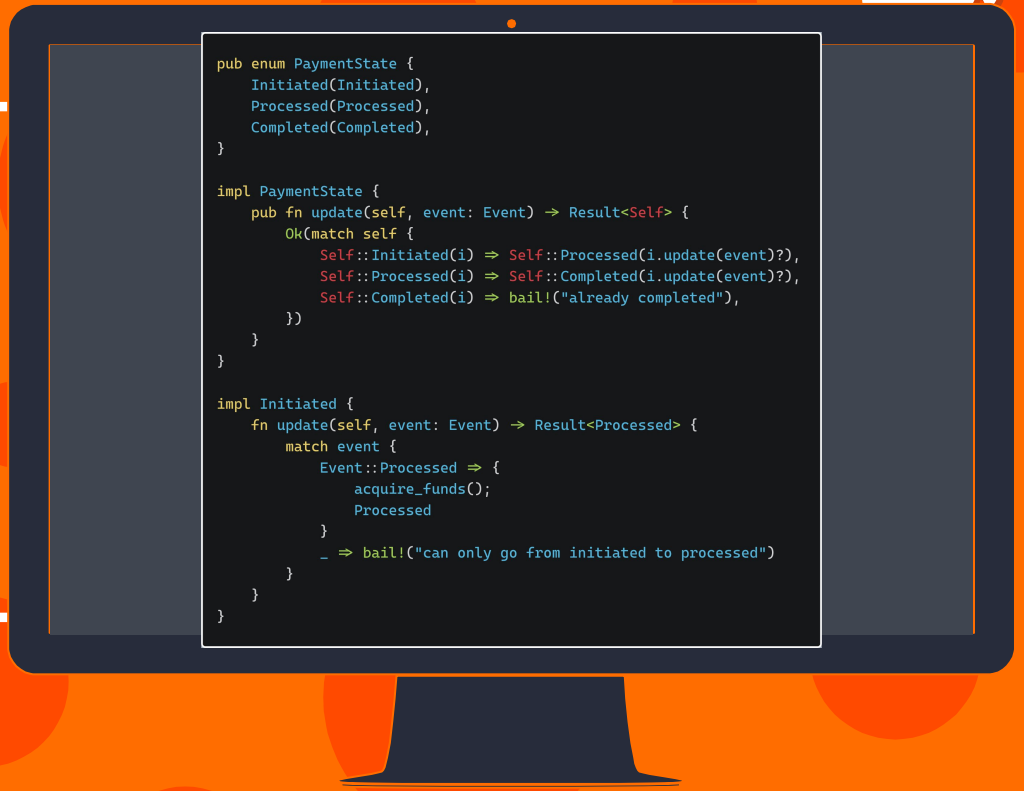
```rust
pub enum PaymentState {
    Initiated(Initiated),
    Processed(Processed),
    Completed(Completed),
}

impl PaymentState {
    pub fn update(self, event: Event) -> Result<Self> {
        Ok(match self {
            Self::Initiated(i) => Self::Processed(i.update(event)?),
            Self::Processed(i) => Self::Completed(i.update(event)?),
            Self::Completed(i) => bail!("already completed"),
        })
    }
}

impl Initiated {
    fn update(self, event: Event) -> Result<Processed> {
        match event {
            Event::Processed => {
                acquire_funds();
                Processed
            }
            _ => bail!("can only go from initiated to processed")
        }
    }
}
```

Use ownership

Make each state a custom type, and use the type system to make sure it only progresses forward



```rust
pub enum PaymentState {
    Initiated(Initiated),
    Processed(Processed),
    Completed(Completed),
}

impl PaymentState {
    pub fn update(self, event: Event) -> Result<Self> {
        Ok(match self {
            Self::Initiated(i) => Self::Processed(i.update(event)?),
            Self::Processed(i) => Self::Completed(i.update(event)?),
            Self::Completed(i) => bail!("already completed"),
        })
    }
}

impl Initiated {
    fn update(self, event: Event) -> Result<Processed> {
        match event {
            Event::Processed => {
                acquire_funds();
                Processed
            }
            _ => bail!("can only go from initiated to processed")
        }
    }
}
```

Case study: rustls

goto fail

This is the name of a vulnerability in Apple Secure Transport CVE-2014-1266. This boiled down to the following code, which validates the server's signature on the key exchange:

```
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
>       goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
```

The marked line was duplicated, likely accidentally during a merge. This meant the remaining part of the function (including the actual signature validation) was unconditionally skipped.

Case study: rustls

```rust
impl State<ClientConnectionData> for ExpectCertificate {
    fn handle(mut self: Box<Self>, cx: &mut ClientContext<'_>, m: Message) -> hs::NextStateOrError {
        let cert_chain = require_handshake_msg!(
            m,
            HandshakeType::Certificate,
            HandshakePayload::CertificateTls13
        )?;
        self.transcript.add_message(&m);

        let server_cert =
            ServerCertDetails::new(cert_chain.convert(), cert_chain.get_end_entity_ocsp());

        Ok(Box::new(ExpectCertificateVerify {
            config: self.config,
            server_name: self.server_name,
            randoms: self.randoms,
            suite: self.suite,
            transcript: self.transcript,
            key_schedule: self.key_schedule,
            server_cert,
            client_auth: self.client_auth,
        }))
    }
}
```

Algorithmic Lucidity

Are you safe from the alg:none vulnerability?

Not even Microsoft is safe

It has been 36 days since the last alg:none JWT vulnerability.

An unauthenticated attacker could impersonate any user in SharePoint 2019 by using an alg:none JWT for OAuth authentication.

Introducing PASETO

Algorithm Lucidity

Algorithm Lucidity refers to resilience against algorithm confusion attacks.

This document aims to make it easy for PASETO implementations to achieve this property.

PASETO Cryptography Key Requirements

Cryptography keys in PASETO are defined as **both the raw key material and its parameter choices, not just the raw key material.**

PASETO implementations **MUST** enforce some logical separation between different key types; especially when the raw key material is the same (i.e. a 256-bit opaque blob).

Arbitrary strings (or byte arrays, or equivalent language constructs) **MUST NOT** be accepted as a key in any PASETO library, **UNLESS** it's an application-specific encoding that encapsulates both the key and an algorithm identifier. (For example, a `k2.local` PASERK.)

In order to allow for key interoperability between different PASETO libraries, any PASETO library **SHOULD** support the `local`, `public` and `secret` types from PASERK.

Introducing PASETO

Algorithm Lucidity

Algorithm Lucidity refers to resilience against algorithm confusion attacks.

This document aims to make it easy for PASETO implementations to achieve this property.

PASETO Cryptography Key Requirements

Cryptography keys in PASETO are defined as **both the raw key material and its parameter choices, not just the raw key material.**

PASETO implementations **MUST** enforce some logical separation between different key types; especially when the raw key material is the same (i.e. a 256-bit opaque blob).

Arbitrary strings (or byte arrays, or equivalent language constructs) **MUST NOT** be accepted as a key in any PASETO library, **UNLESS** it's an application-specific encoding that encapsulates both the key and an algorithm identifier. (For example, a `k2.local` PASERK.)

In order to allow for key interoperability between different PASETO libraries, any PASETO library **SHOULD** support the `local`, `public` and `secret` types from PASERK.

Introducing PASETO

```
class SymmetricKey extends Key {
    public SymmetricKey(byte[] keyMaterial, Version version) {
        super(keyMaterial, version);
    }

    public bool isKeyValidFor(Version v, Purpose p) {
        return v == this.version && p == Purpose.PURPOSE_LOCAL;
    }
}
```

Introducing PASETO

```php
/*
 * PASETO Version 4 - Encrypt - Step 1
 *
 * We already have a type-safety check on local/public. The following check
 * constrains the key to v4 tokens only.
 */
if (!($key->getProtocol() instanceof Version4)) {
    throw new InvalidVersionException(
        'The given key is not intended for this version of PASETO.',
        ExceptionCode::WRONG_KEY_FOR_VERSION
    );
}
```

Introducing pasta-tokens

```rust
pub struct V3;
pub struct V4;

impl<V, M> UnencryptedToken<V, M> {
    pub fn encrypt(self, key: &SymmetricKey<V>)
      -> Result<EncryptedToken<V>, PasetoError>
    {
        // the type system has checked that our key version is correct :)
    }
}
```

Introducing pasta-tokens

```rust
impl<V: version::Version> FromStr for EncryptedToken<V> {
    type Err = PasetoError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        // ensure token starts with `v3.local.` or `v4.local.`
        let s = s.strip_prefix(V::PASETO_HEADER).ok_or(PasetoError::InvalidToken)?;
        let s = s.strip_prefix(".local.").ok_or(PasetoError::InvalidToken)?;

        // ...
    }
}
```

Turing completeness

The type system is Turing complete… I'm sorry…


Typenum.
Trick or treat?

What is typenum?

```
use std::ops::Add;
use typenum::{Integer, P3, P4};

type X = <P3 as Add<P4>>::Output;
assert_eq!(<X as Integer>::to_i32(), 7);
```

How is typenum?

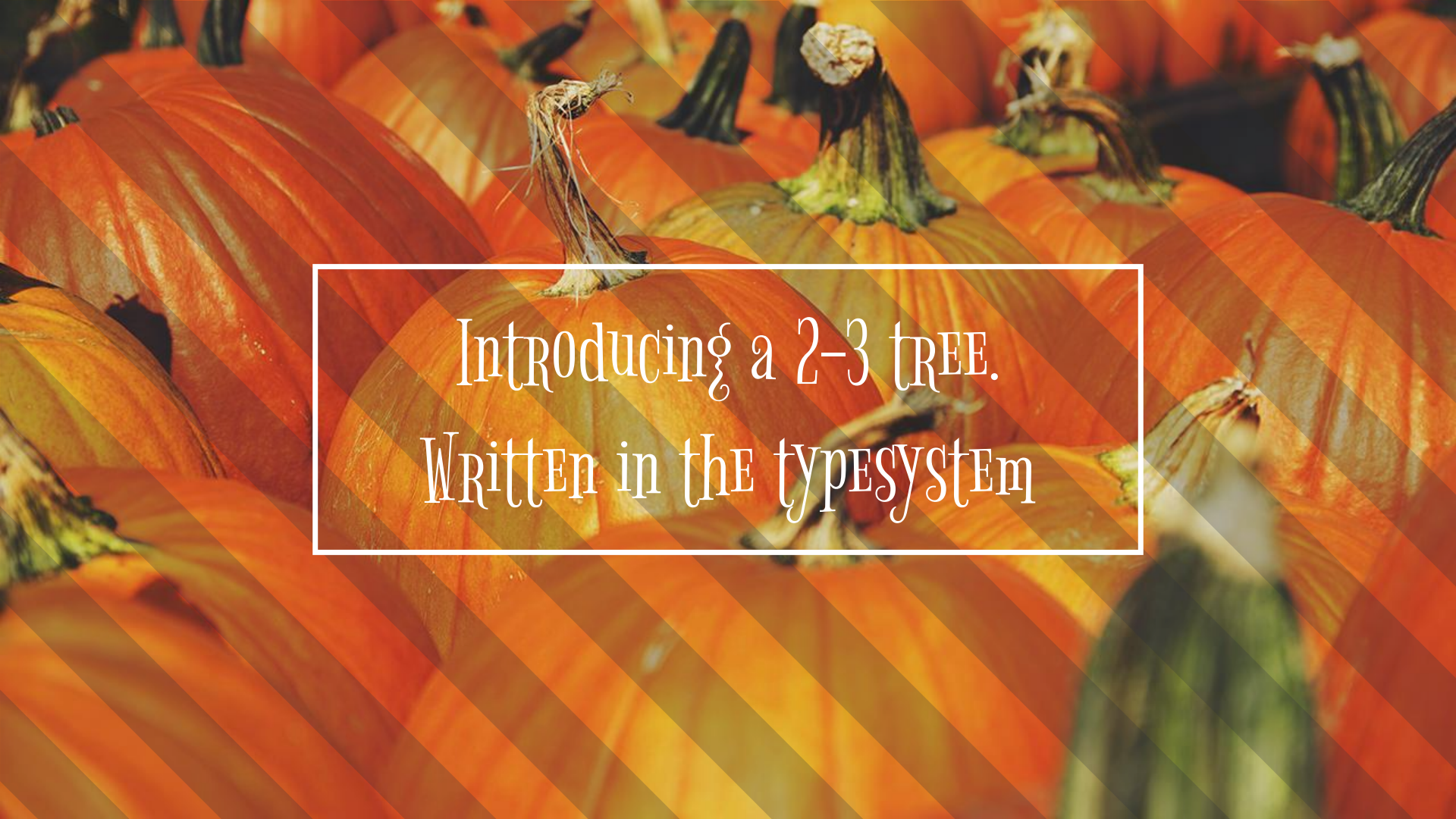
```
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>, B0>;
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>, B1>;
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>, B0>;
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
```

How does typenum?

```rust
/// `UInt<Ul, B1> + UInt<Ur, B1> = UInt<(Ul + Ur) + B1, B0>`
impl<Ul: Unsigned, Ur: Unsigned> Add<UInt<Ur, B1>> for UInt<Ul, B1>
where
    Ul: Add<Ur>,
    Sum<Ul, Ur>: Add<B1>,
{
    type Output = UInt<Add1<Sum<Ul, Ur>>, B0>;
    #[inline]
    fn add(self, rhs: UInt<Ur, B1>) -> Self::Output {
        UInt {
            msb: self.msb + rhs.msb + B1,
            lsb: B0,
        }
    }
}
```

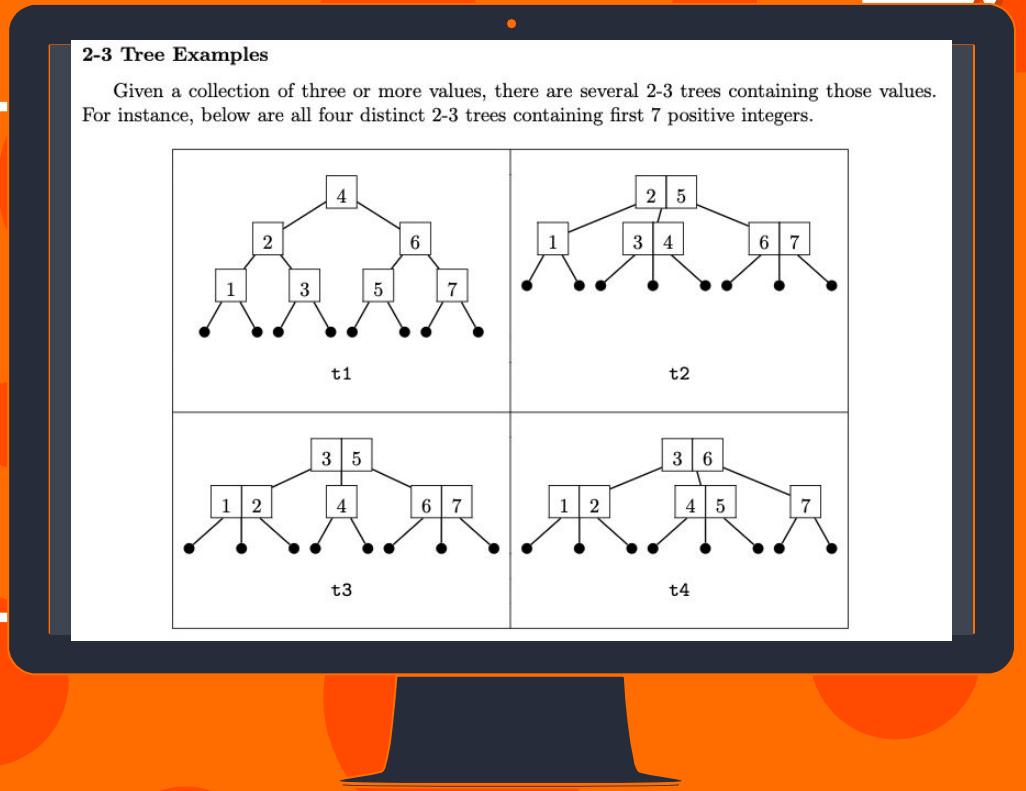
Why does typenum?

```rust
pub trait OutputSizeUser {
    type OutputSize: ArrayLength<u8> + 'static;
}
pub type Output<T> = GenericArray<u8, <T as OutputSizeUser>::OutputSize>;
pub trait Digest: OutputSizeUser {
    fn new() -> Self;
    fn update(&mut self, data: impl AsRef<[u8]>);
    fn finalize(self) -> Output<Self>;
}
```


Introducing a 2-3 tree.
Written in the typesystem

What is a 2-3 tree?

It's a self-balancing tree that has guaranteed log(n) insertion and retrieval lookup times



Step 1: A node

```
pub struct Node<T, D: TreeDepth<T>> {
    pub pivots: Pivots<T, D>,
    pub tail: Child<T, D>,
}

pub enum Pivots<T, D: TreeDepth<T>> {
    One([(Child<T, D>, T); 1]),
    Two([(Child<T, D>, T); 2]),
}
```

Each non-terminal node must have at least 1 pivot, and at least 2 children.

Step 2: A child-node

```
type Child<T, D> = <D as TreeDepth<T>>::Child;

impl<T> TreeDepth<T> for U0 {
    type Child = ();
}

impl<T, U: Sub1> TreeDepth<T> for U
where
    super::Sub1<U>: TreeDepth<T>,
{
    type Child = Node<T, super::Sub1<U>>;
}
```

A terminal node's children are leaf objects (using unit type here)

A non-terminal node's children are more nodes

Step 3: A root-node

```
impl<T, U: Sub1> TreeRoot<T> for U
where
    super::Sub1<U>: TreeDepth<T>,
{
    type RootNode = Node<T, U>;
}

impl<T> TreeRoot<T> for U0 {
    type RootNode = RootNode<T>;
}

pub enum RootNode<T> {
    Empty,
    Partial(Node<T, U0>),
}
```

A terminal root-node can have 0 pivots/children

A non-terminal root-node is a regular node

It's as simple as that...

```rust
impl<T: Ord, U: TreeDepth<T, Child = Node<T, crate::Sub1<U>>> + Sub1> Insert<T> for Node<T, U>
where
    crate::Sub1<U>: TreeDepth<T>,
    Child<T, U>: Insert<T, Depth = crate::Sub1<U>>,
{
    type Depth = U;
    fn insert(self, k: T) -> Result<InsertOverflow<Self, Split<T, Self::Depth>>, (Self, T)> {
        let Self { pivots, tail } = self;
        match pivots {
            Pivots::One([x]) if k < x.1 => match x.0.insert(k) {
                Ok(InsertOverflow::Same(x0)) => Ok(InsertOverflow::Same(Self {
                    pivots: Pivots::One([(x0, x.1)]),
                    tail,
                })),
                Ok(InsertOverflow::Overflow((l, p, r))) => Ok(InsertOverflow::Same(Self {
                    pivots: Pivots::Two([(l, p), (r, x.1)]),
                    tail,
                })),
                Err((x0, t)) => Err((
                    Self {
                        pivots: Pivots::One([(x0, x.1)]),
                        tail,
                    },
                    t,
                )),
            },
            // ...
        }
    }
}
```


Demo

Happy HalloWEEn!

Any questions?

You can find me at @conradludgate in most places