State of Rust 2016

Alex Crichton

Rust is one year old!

- 11,894 commits by 702 contributors
- 88 RFCs merged
- 24+ compiler targets introduced
- 9 releases shipped
- 1 year of stability delivered

Rust in production

Dropbox

ETHCORE

TILDE

AppSignal

mozilla

OneSignal

https://www.rust-lang.org/friends.html

Focus after 1.0

- Branching out: taking Rust to new places
- Doubling down: infrastructure investments
- Zeroing in: closing gaps in our key features

Rust Everywhere!

Embedding Rust

No runtime, zero cost FFI, portable

Introducing Helix

Rust + Ruby, Without The Glue.

`panic!`

- Bugs happen :(
- Stack unwinding by default
- Undefined behavior going into C
- Isolation boundaries

std::panic

```
pub fn catch_unwind<F, R>(f: F) -> Result<R>
    where F: FnOnce() -> R + UnwindSafe
```

- Not one, but two RFCs!
- Allows propagation of errors at boundaries
- Is **not** a shift in Rust’s error handling

-C panic=abort

- Can’t always recover from error
- Can’t always implement unwinding
- “Landing pads” are extra code to generate
    - 10% faster compiles
    - 10% smaller binaries

Compiler targets, oh my!

- 6 targets with binaries at 1.0.0
- 30 targets with binaries today
- MSVC is now a Tier 1 platform
- 4.5 GB of artifacts every night
- MIPS, ARM, AArch64, PowerPC, NetBSD, FreeBSD, Rumprun, Android, iOS

Ok, what now?

Did someone say static binaries?

```
$ curl https://sh.rustup.rs | sh
$ rustup target add x86_64-unknown-linux-musl
$ cargo new foo && cd foo
$ cargo build --target x86_64-unknown-linux-musl
$ ldd target/debug/foo
        not a dynamic executable
```

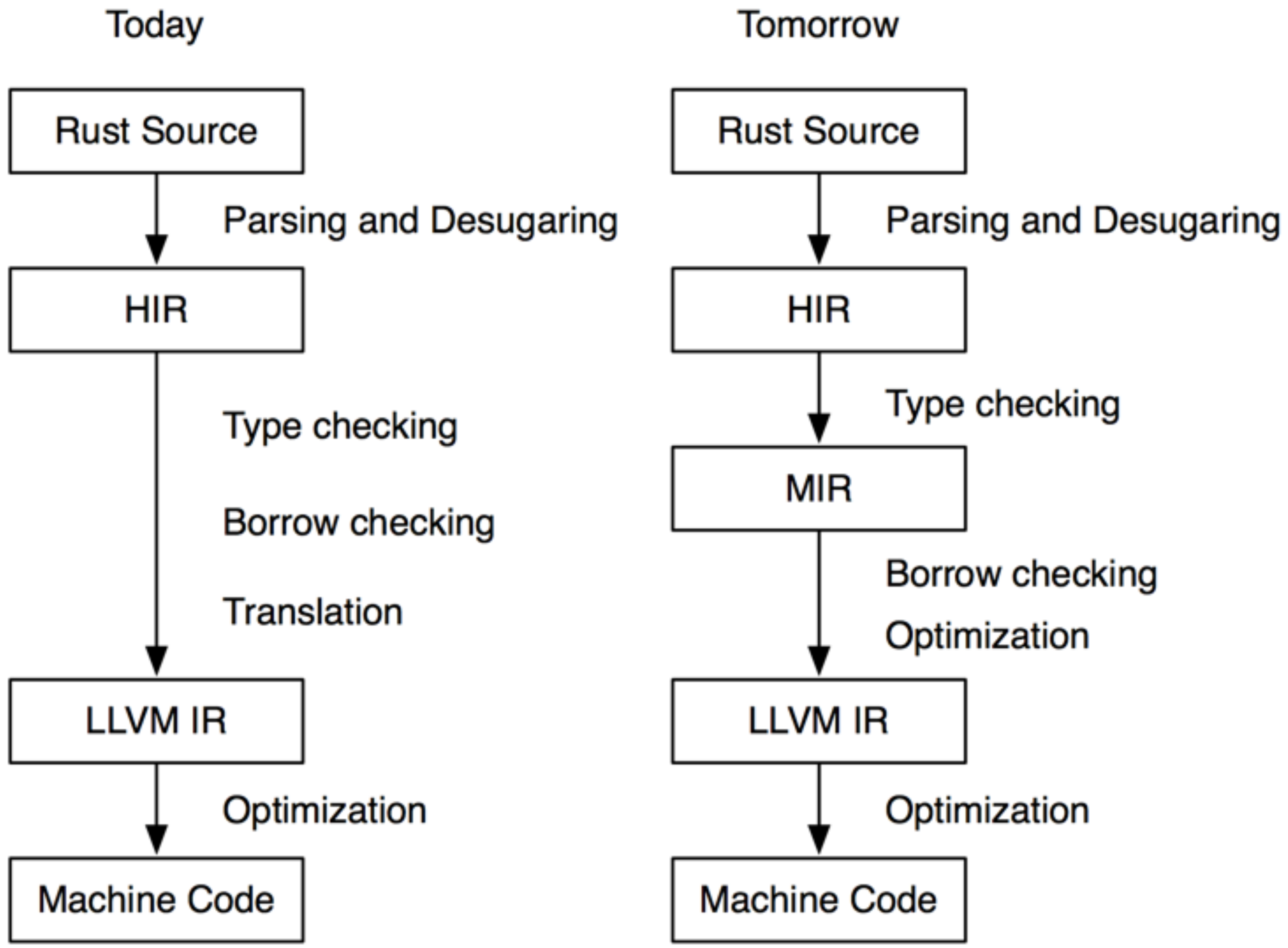
Down to business

```
$ rustup target add arm-linux-androideabi
$ cargo build --target arm-linux-androideabi
error: linking with `cc` failed
```

- NDK management
- Cargo configuration

MIR

MIR?


Today
Rust Source
Parsing and Desugaring
HIR
Type checking
Borrow checking
Translation
LLVM IR
Optimization
Machine Code
Tomorrow
Rust Source
Parsing and Desugaring
HIR
Type checking
MIR
Borrow checking
Optimization
LLVM IR
Optimization
Machine Code

Why MIR?

- Faster compile times
- Faster execution times
- More precise type checking
- Engineering benefits

Simplifying Rust

```
for elem in vec {
    process(elem);
}
```

Simplifying Rust

```rust
for elem in vec {
    process(elem);
}
```

$$\Downarrow$$

```rust
let mut iter = vec.into_iter();
while let Some(elem) = iter.next() {
    process(elem);
}
```

Simplifying Rust

```
let mut iter = vec.into_iter();
while let Some(elem) = iter.next() {
    process(elem);
}
```

```
let mut iter = vec.into_iter();
loop {
    match iter.next() {
        Some(elem) => process(elem),
        None => break,
    }
}
```

Simplifying Rust

```
let mut iter = vec.into_iter();
loop {
    match iter.next() {
        Some(elem) => process(elem),
        None => break,
    }
}
```

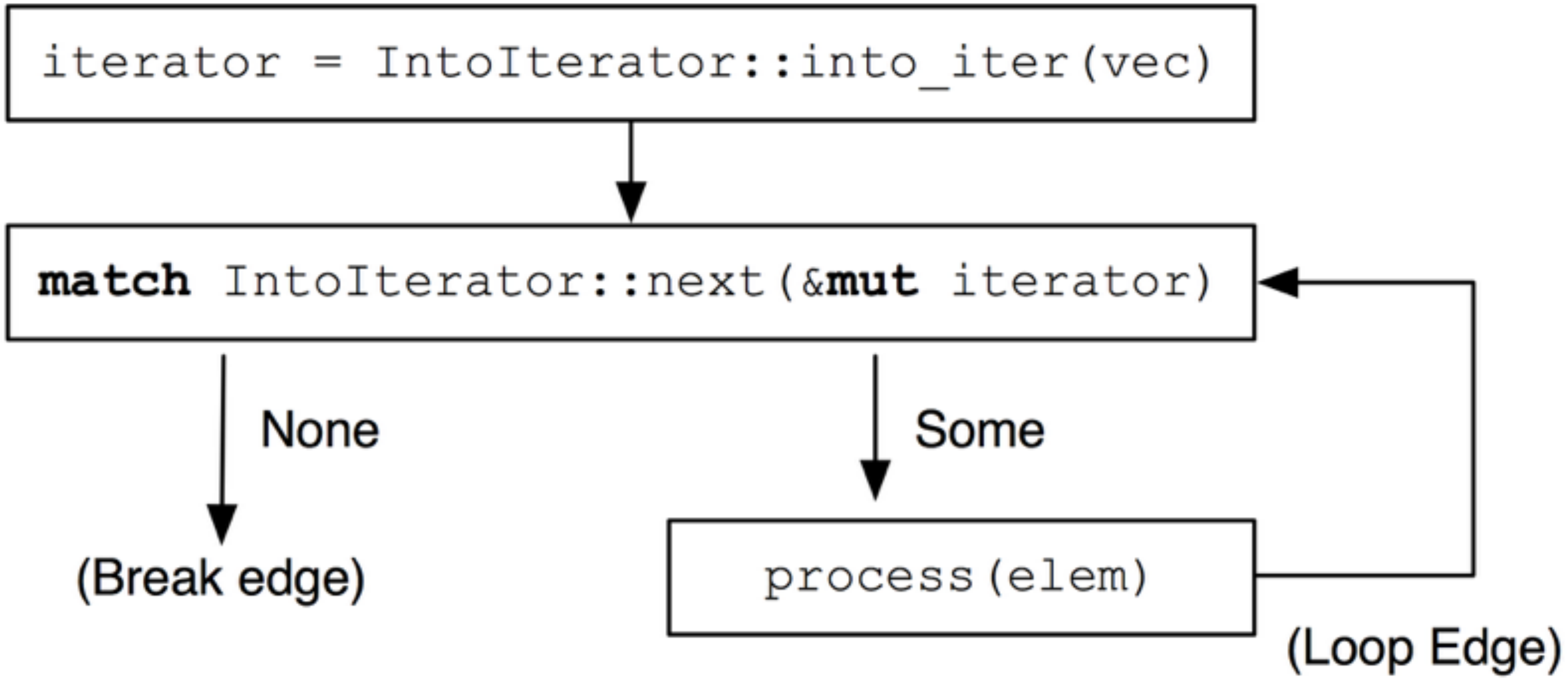
```
let mut iter = IntoIterator::into_iter(vec);
loop {
   match Iterator::next(&mut iterator) {
        Some(elem) => process(elem),
        None => break,
    }
}
```

Simplifying Rust

```
let mut iter = IntoIterator::into_iter(vec);
loop {
   match Iterator::next(&mut iterator) {
        Some(elem) => process(elem),
        None => break,
    }
}
```

```
    let mut iter = IntoIterator::into_iter(vec);
loop:
    match Iterator::next(&mut iter) {
        Some(e) => { process(e); goto loop; }
        None => { goto break; }
    }
break:
    ...
```

Control-flow Graphs


iterator = IntoIterator::into_iter(vec)
match IntoIterator::next(&mut iterator)
None
Some
(Break edge)
process(elem)
(Loop Edge)

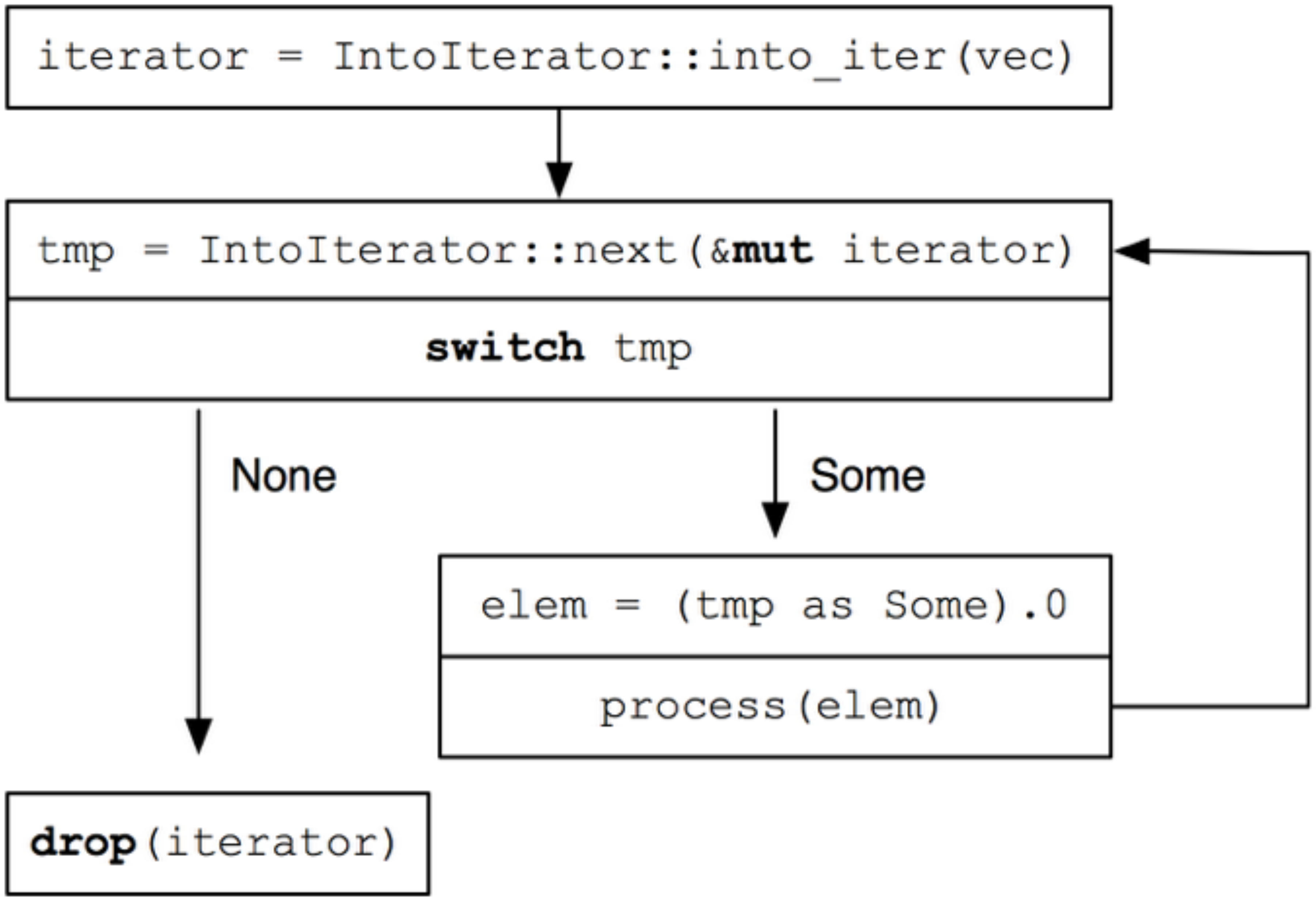
Simplifying `match`

```
match Iterator::next(&mut iter) {
    Some(e) => process(e),
    None => break,
}
```

Simplifying `match`

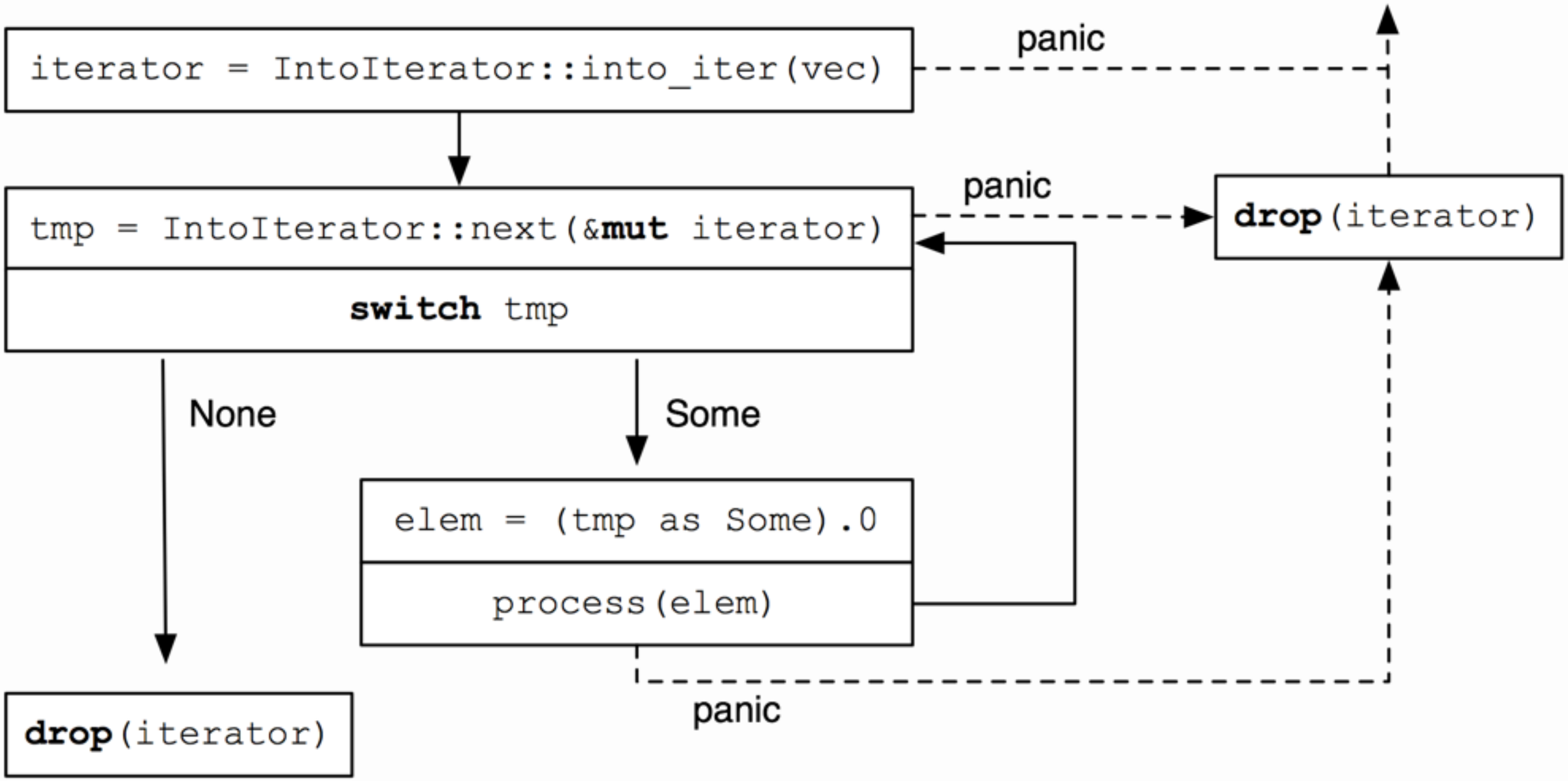
```
switch tmp {
    Some => {
        let e = (tmp as Some).0;
        process(e);
        goto loop;
    }
    None => goto break,
}
```

Drop



iterator = IntoIterator::into_iter(vec)
tmp = IntoIterator::next(&mut iterator)
switch tmp
None
Some
elem = (tmp as Some).0
process(elem)
drop(iterator)

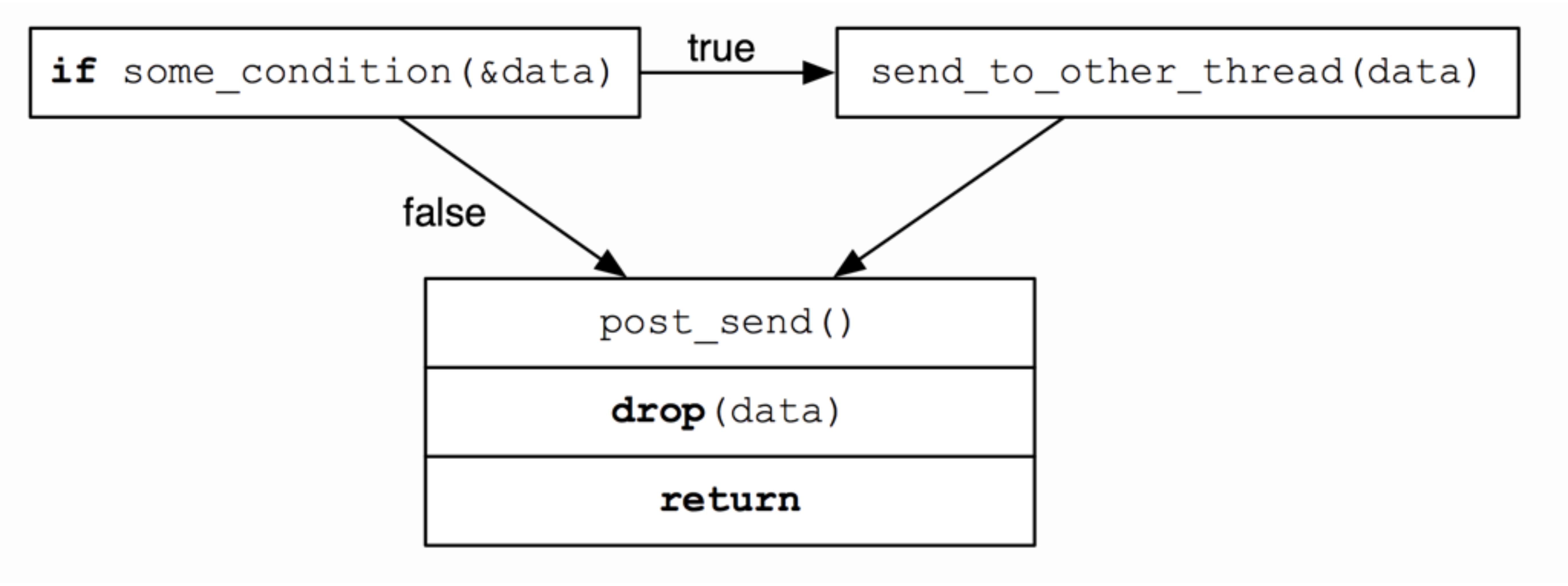
Drop


iterator = IntoIterator::into_iter(vec)
panic
tmp = IntoIterator::next(&mut iterator)
switch tmp
panic
drop(iterator)
None
Some
elem = (tmp as Some).0
process(elem)
panic
drop(iterator)

Drop flags

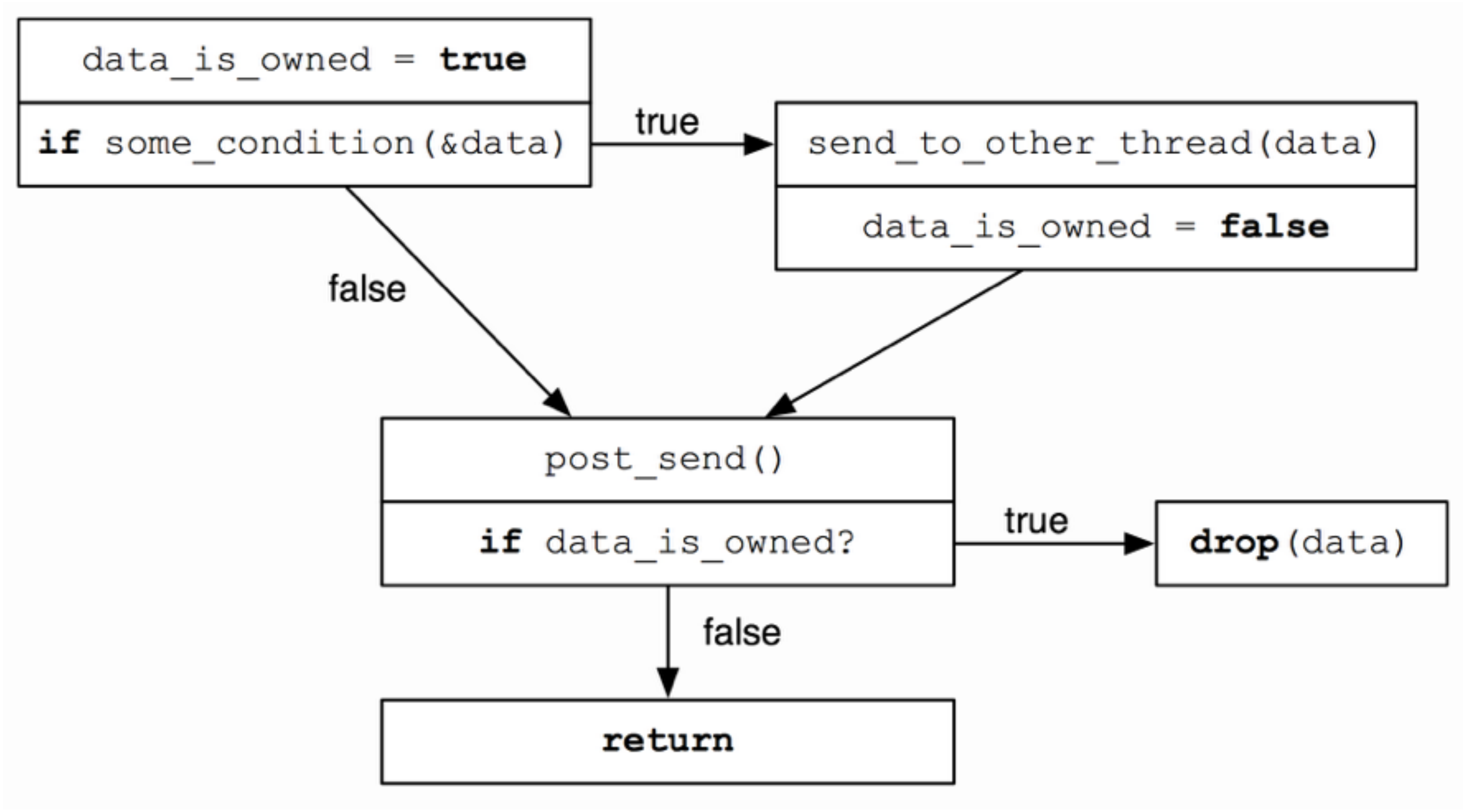
```
fn send_if(data: Vec<Data>) {
    if some_condition(&data) {
        send_to_other_thread(data);
    }
    post_send();
}
```

Drop flags



if some_condition(&data)
true
send_to_other_thread(data)
false
post_send()
drop(data)
return

Drop flags



data_is_owned = true
if some_condition(&data)
true
send_to_other_thread(data)
data_is_owned = false
false
post_send()
if data_is_owned?
true
drop(data)
false
return

Async I/O and Futures

I/O in std

- Blocking APIs in std::fs, std::net, …
- Read/Write reports errors on ‘would block’
- Composing I/O is difficult
  - Accepting a connection with a timeout
  - Waiting on one of two I/O events to happen

carllerche/mio

- Metal I/O - thin epoll/kqueue wrapper
- “Dear kernel, what happened since I last asked?”
- Windows support through IOCP and shims
  - Not as metal here
- Foundation for Async I/O and event loops

mio Echo Server

```
extern crate mio;

#[macro_use] extern crate log;
extern crate env_logger;

use std::io;
use std::io::{Error, ErrorKind};
use std::net::SocketAddr;
use std::str::FromStr;

use mio::*;
use mio::buf::ByteBuf;
use mio::tcp::*;
use mio::util::Slab;

fn main() {

    // Before doing anything, let us register a logger. The mio library has really good logging
    // at the _trace_ and _debug_ levels. Having a logger setup is invaluable when trying to
    // figure out why something is not working correctly.
    env_logger::init().ok().expect("Failed to init logger");

    let addr: SocketAddr = FromStr::from_str("127.0.0.1:8000")
        .ok().expect("Failed to parse host:port string");
    let sock = TcpListener::bind(&addr).ok().expect("Failed to bind address");

    let mut event_loop = EventLoop::new().ok().expect("Failed to create event loop");

    // Create our Server object and register that with the event loop. I am hiding away
    // the details of how registering works inside of the `Server#register` function. One reason I
    // really like this is to get around having to have `const SERVER = Token(0)` at the top of my
    // file. It also keeps our polling options inside `Server`.
    let mut server = Server::new(sock);
    server.register(&mut event_loop).ok().expect("Failed to register server with event loop");

    info!("Even loop starting...");
    event_loop.run(&mut server).ok().expect("Failed to start event loop");
}


struct Server {
    // main socket for our server
    sock: TcpListener,

    // token of our server. we keep track of it here instead of doing `const SERVER = Token(0)`.
    token: Token,

    // a list of connections _accepted_ by our server
    conns: Slab<Connection>,

}

impl Handler for Server {
    type Timeout = ();
    type Message = ();

    fn ready(&mut self, event_loop: &mut EventLoop<Server>, token: Token, events: EventSet) {
        debug!("events = {:?}", events);
        assert!(token != Token(0), "[BUG]: Received event for Token(0)");

        if events.is_error() {
            warn!("Error event for {:?}", token);
            self.reset_connection(event_loop, token);
            return;
        }

        if events.is_hup() {
            trace!("Hup event for {:?}", token);
            self.reset_connection(event_loop, token);
            return;
        }

        // We never expect a write event for our `Server` token . A write event for any other token
        // should be handed off to that connection.
        if events.is_writable() {
            trace!("Write event for {:?}", token);
            assert!(self.token != token, "Received writable event for Server");

            self.find_connection_by_token(token).writable()
                .and_then(|_| self.find_connection_by_token(token).reregister(event_loop))
                .unwrap_or_else(|e| {
                    warn!("Write event failed for {:?}, {:?}", token, e);
                    self.reset_connection(event_loop, token);
                });
        }

        // A read event for our `Server` token means we are establishing a new connection. A read
        // event for any other token should be handed off to that connection.
        if events.is_readable() {
            trace!("Read event for {:?}", token);
            if self.token == token {
                self.accept(event_loop);
            } else {

                self.readable(event_loop, token)
                    .and_then(|_| self.find_connection_by_token(token).reregister(event_loop))
                    .unwrap_or_else(|e| {
                        warn!("Read event failed for {:?}: {:?}", token, e);
                        self.reset_connection(event_loop, token);
                    });
            }
        }
    }
}
```

```
impl Server {
    fn new(sock: TcpListener) -> Server {
        Server {
            sock: sock,

            // I don't use Token(0) because kqueue will send stuff to Token(0)
            // by default causing really strange behavior. This way, if I see
            // something as Token(0), I know there are kqueue shenanigans
            // going on.
            token: Token(1),

            // SERVER is Token(1), so start after that
            // we can deal with a max of 126 connections
            conns: Slab::new_starting_at(Token(2), 128)
        }
    }

    /// Register Server with the event loop.
    ///
    /// This keeps the registration details neatly tucked away inside of our implementation.
    fn register(&mut self, event_loop: &mut EventLoop<Server>) -> io::Result<()> {
        event_loop.register_opt(
            &self.sock,
            self.token,
            EventSet::readable(),
            PollOpt::edge() | PollOpt::oneshot()
        ).or_else(|e| {
            error!("Failed to register server {:?}, {:?}", self.token, e);
            Err(e)
        })
    }

    /// Register Server with the event loop.
    ///
    /// This keeps the registration details neatly tucked away inside of our implementation.
    fn reregister(&mut self, event_loop: &mut EventLoop<Server>) {
        event_loop.reregister(
            &self.sock,
            self.token,
            EventSet::readable(),
            PollOpt::edge() | PollOpt::oneshot()
        ).unwrap_or_else(|e| {
            error!("Failed to reregister server {:?}, {:?}", self.token, e);
            let server_token = self.token;
            self.reset_connection(event_loop, server_token);
        })
    }

    /// Accept a _new_ client connection.
    ///
    /// The server will keep track of the new connection and forward any events from the event loop
    /// to this connection.
    fn accept(&mut self, event_loop: &mut EventLoop<Server>) {
        debug!("server accepting new socket");

        // Log an error if there is no socket, but otherwise move on so we do not tear down the
        // entire server.
        let sock = match self.sock.accept() {
            Ok(s) => {
                match s {
                    Some(sock) => sock,
                    None => {
                        error!("Failed to accept new socket");
                        self.reregister(event_loop);
                        return;
                    }
                }
            },
            Err(e) => {
                error!("Failed to accept new socket, {:?}", e);
                self.reregister(event_loop);
                return;
            }
        };

        // `Slab#insert_with` is a wrapper around `Slab#insert`. I like `#insert_with` because I
        // make the `Token` required for creating a new connection.
        //
        // `Slab#insert` returns the index where the connection was inserted. Remember that in mio,
        // the Slab is actually defined as `pub type Slab<T> = ::slab::Slab<T, ::Token>;`. Token is
        // just a tuple struct around `usize` and Token implemented `::slab::Index` trait. So,
        // every insert into the connection slab will return a new token needed to register with
        // the event loop. Fancy...
        match self.conns.insert_with(|token| {
            debug!("registering {:?} with event loop", token);
            Connection::new(sock, token)
        }) {
            Some(token) => {
                // If we successfully insert, then register our connection.
                match self.find_connection_by_token(token).register(event_loop) {
                    Ok(_) => {},
                    Err(e) => {
                        error!("Failed to register {:?} connection with event loop, {:?}", token, e);
                        self.conns.remove(token);
                    }
                }
            },
            None => {
                // If we fail to insert, `conn` will go out of scope and be dropped.
                error!("Failed to insert connection into slab");
            }
```

```
    /// Forward a readable event to an established connection.
    ///
    /// Connections are identified by the token provided to us from the event loop. Once a read has
    /// finished, push the receive buffer into the all the existing connections so we can
    /// broadcast.
    fn readable(&mut self, event_loop: &mut EventLoop<Server>, token: Token) -> io::Result<()> {
        debug!("server conn readable; token={:?}", token);
        let message = try!(self.find_connection_by_token(token).readable());

        if message.remaining() == message.capacity() { // is_empty
            return Ok(());
        }

        // TODO pipeine this whole thing
        let mut bad_tokens = Vec::new();

        // Queue up a write for all connected clients.
        for conn in self.conns.iter_mut() {
            // TODO: use references so we don't have to clone
            let conn_send_buf = ByteBuf::from_slice(message.bytes());
            conn.send_message(conn_send_buf)
                .and_then(|_| conn.reregister(event_loop))
                .unwrap_or_else(|e| {
                    error!("Failed to queue message for {:?}: {:?}", conn.token, e);
                    // We have a mutable borrow for the connection, so we cannot remove until the
                    // loop is finished
                    bad_tokens.push(conn.token)
                });
        }

        for t in bad_tokens {
            self.reset_connection(event_loop, t);
        }

        Ok(())
    }

    fn reset_connection(&mut self, event_loop: &mut EventLoop<Server>, token: Token) {
        if self.token == token {
            event_loop.shutdown();
        } else {
            debug!("reset connection; token={:?}", token);
            self.conns.remove(token);
        }
    }

    /// Find a connection in the slab using the given token.
    fn find_connection_by_token<'a>(&'a mut self, token: Token) -> &'a mut Connection {
        &mut self.conns[token]
    }
}

/// A stateful wrapper around a non-blocking stream. This connection is not
/// the SERVER connection. This connection represents the client connections
/// _accepted_ by the SERVER connection.
struct Connection {
    // handle to the accepted socket
    sock: TcpStream,

    // token used to register with the event loop
    token: Token,

    // set of events we are interested in
    interest: EventSet,

    // messages waiting to be sent out
    send_queue: Vec<ByteBuf>,
}

impl Connection {
    fn new(sock: TcpStream, token: Token) -> Connection {
        Connection {
            sock: sock,
            token: token,

            // new connections are only listening for a hang up event when
            // they are first created. we always want to make sure we are
            // listening for the hang up event. we will additionally listen
            // for readable and writable events later on.
            interest: EventSet::hup(),

            send_queue: Vec::new(),
        }
    }

    /// Handle read event from event loop.
    ///
    /// Currently only reads a max of 2048 bytes. Excess bytes are dropped on the floor.
    ///
    /// The recieve buffer is sent back to `Server` so the message can be broadcast to all
    /// listening connections.
    fn readable(&mut self) -> io::Result<ByteBuf> {

        // ByteBuf is a heap allocated slice that mio supports internally. We use this as it does
        // the work of tracking how much of our slice has been used. I chose a capacity of 2048
        // after reading
        // https://github.com/carllerche/mio/blob/eed4855c627892b88f7ca68d3283cbc708a1c2b3/src/io.rs#L23-27
        // as that seems like a good size of streaming. If you are wondering what the difference
        // between messaged based and continuous streaming read
        // http://stackoverflow.com/questions/3017633/difference-between-message-oriented-protocols-and-stream-oriented-
```

Async I/O Ecosystem

- eventual - threadsafe futures
- mioco - coroutines on mio
- gj - single-threaded futures and I/O
- Lots of experience outside of Rust
  - Finagle in Scala at Twitter
  - Wangle in C++ at Facebook

What’s a Future

- In computer science, future, promise, delay, and deferred refer to constructs used for synchronizing in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete.

What’s a Future

```
trait Future {
    type Item;
    type Error;

    fn schedule<F>(&mut self, f: F)
        where F: FnOnce(Result<Item, Error>)
                 + Send + ‘static;
}
```

Isn’t that callback hell?

```
fn num_downloads() -> impl Future<i32> {
    let url = “https://crates.io/summary”;
    http::get(url)
        .and_then(json::parse)
        .map(|j| j.get(“num_downloads”))
        .and_then(i32::from_str)
}
```

Futures in Rust

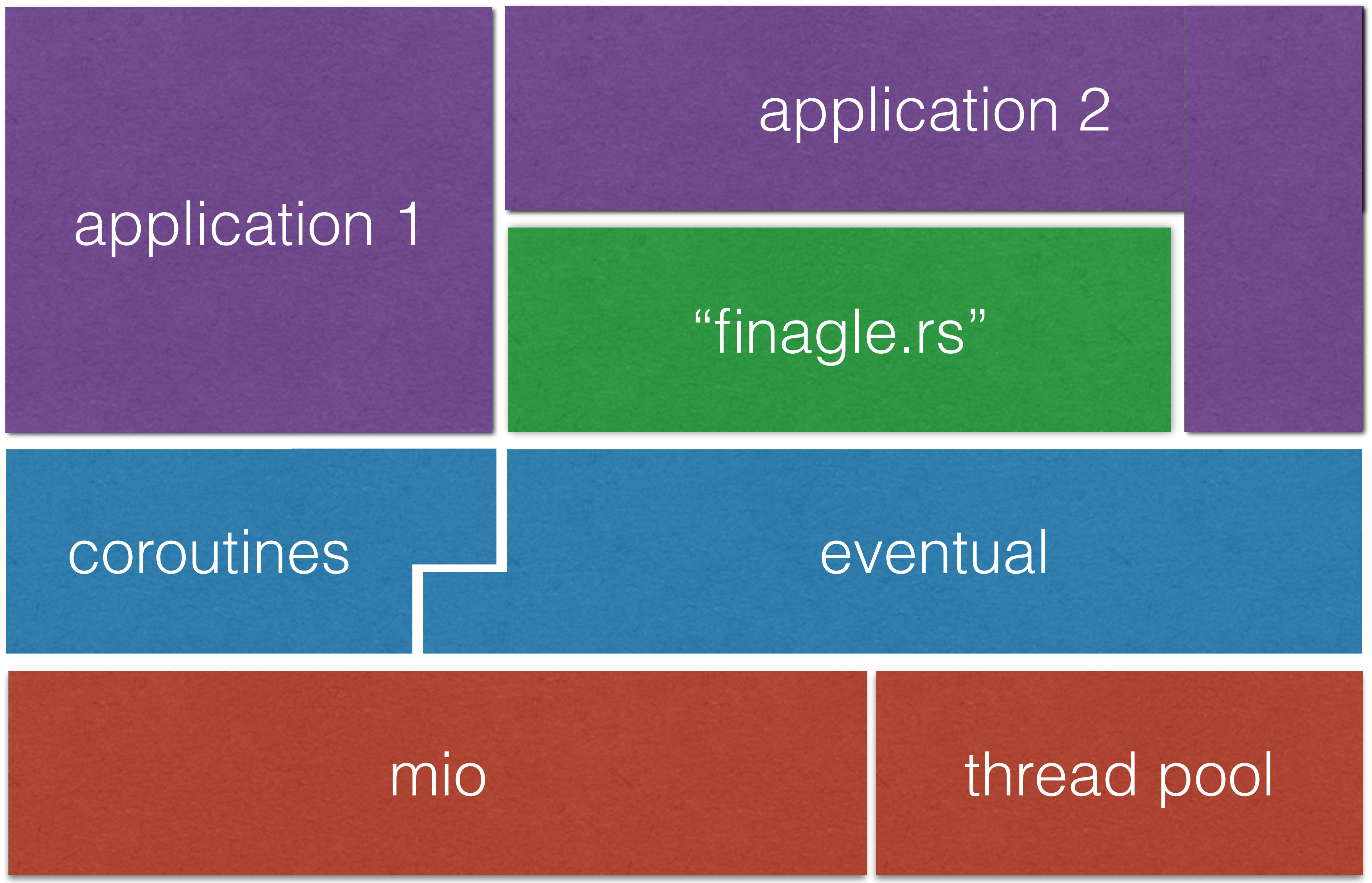
- Ownership is key, a future is resolved once
- Trait allows zero-cost implementations
    - Borrows from `Iterator` for composition and ergonomics
- Cancellation of futures is a core primitive

Cancellation

```
let socket = listener.accept();
let req = socket.map(process);
let timeout = timeout_ms(1_000);
let both = socket.select(timeout);
event_loop.await(both);
```

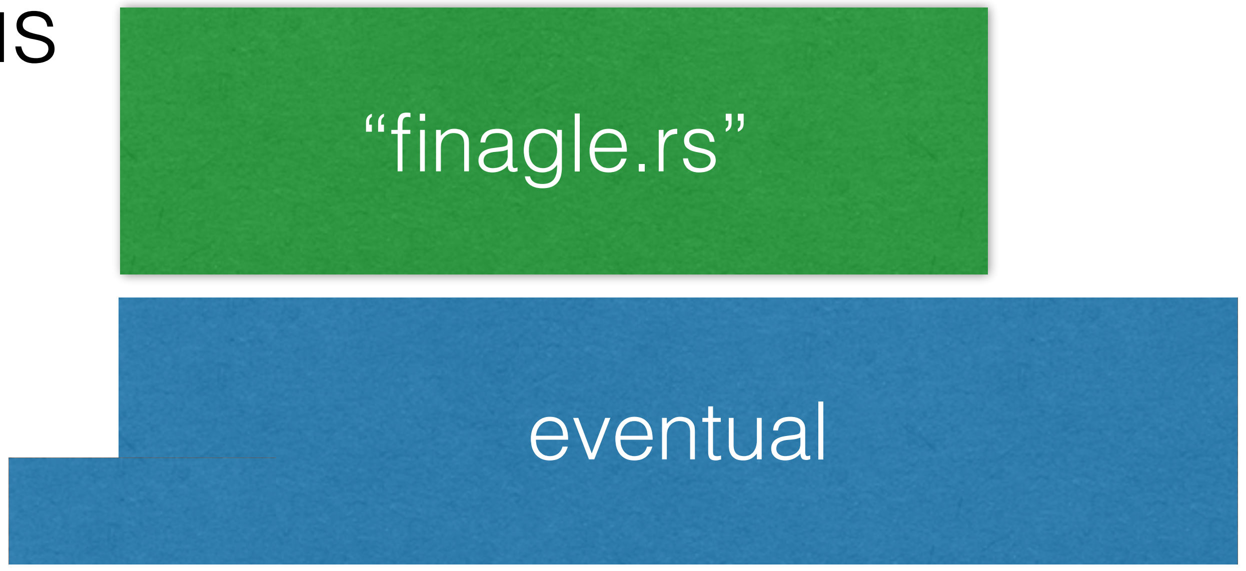
- If timeout happens `process` is *never called*

Future of Futures



Future of Futures

- Requires consensus
- Lots to implement
- Lots to talk about



Upcoming in 2016

2016
Rust
CONF

RustFest
2016

Rust Belt Rust Conference

October 27th & 28th, 2016

Conferences

- RustConf 2016, 9/9-9/10 in Portland
- RustFest 2016, 9/17-9/18 in Berlin
- Rust Belt Rust, 10/27-10/28 in Pittsburgh

Rust Releases

- 1.9 released May 26
  - Includes stabilization of `std::panic`
- 1.10 to be released July 7
  - Includes `panic=abort`, panic hooks, and Unix sockets

New features

- Incremental compilation
- Non-zeroing drop
- Error messages v2
- Flexible borrowing
- IDE initiative
- rustup NDKs
- impl Trait
- Specialization
- pub(restricted)
- WebAssembly
- rustfmt
- plugins
- GC integration
- macro_rules! v2

Focus now

- Branching out: taking Rust to new places
- Doubling down: infrastructure investments
- Zeroing in: closing gaps in our key features