FROM BLOCKING TO ASYNC

AGENDA

- What async means
- Building blocks
- Old way: continuators
- New way: async/await
- Crates

COMPUTATION MODELS

- Sequential calculations
- Parallel calculations
- Async calculations

ASYNCHRONIUS PROGRAMMING IS DEFINING CALCULATIONS AS A GRAPH AND DELEGATE ACTUAL COMPUTATION TO RUNTIME.

BUILDING BLOCKS

FUTURES

CALCULATIONS WHICH MAY EVENTUALLY GIVE SOME RESULT IN FUTURE

FUTURES

```
trait Future {
    fn poll(self: Pin<&mut Self>, cx: &mut Context)
        ->  Poll<Self::Output>;
}

enum Poll {
    Ready(T),
    Pending,
}
```

FUTURES - CONTINUATORS

```
trait FutureExt {
    fn map<U, F>(self, f: F)
        -> impl Future<Item = U>;
    fn then<Fut, F>(self, f: F)
        -> impl Future<Item = Fut::Output>;
    fn inspect<F>(self, f: F)
        -> impl Future<Item = Self::Output>;
}
```

STREAMS

SOURCES OF DATA WHICH MAY BECOME AVAILABLE IN FUTURE

STREAMS

```
trait Stream {
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context)
        -> Poll<Option<Self::Item>>;
}

enum Poll {
    Ready(T),
    Pending,
}
```

STREAMS - COMBINATORS

```
trait StreamExt {
    fn map<T, F>(self, f: F)
        -> impl Stream<Item = T>;
    fn filter<Fut, F>(self, f: F)
        -> impl Stream<Item = Self::Item>;
    fn filter_map<Fut, T, F>(self, f: F)
        -> impl Stream<Item = T>;
    fn then<Fut, F>(self, f: F)
        -> impl Stream<Item = Fut::Output>;

    fn collect<C>(self)
        -> impl Future<Output = C>;
    // ...
}
```

PIN

Pin is a way to ensure, that if the type cares about not being ever moved, it will never move

UNPIN

Unping is a way to say, that type doesn't care about being moved

PIN

```rust
fn main() {
    let s = create_some_stream();
    // Compile error - s is not pinned
    let polled = s.poll_next(cx);
}
```

PIN

```rust
fn main() {
    let s = create_some_stream();
    // Pinning s to stack - s may not be ever moved
    pin_mut!(s);
    let polled = s.poll_next(cx);
}
```

```rust
fn main() {
    let s = create_some_stream();
    // Pinning s to heap - s may not be ever moved
    // (but whole box may)
    let s = Box::pin(s);
    let polled = s.poll_next(cx);
}
```

The `Pin` API is highly unsafe - it is not a good idea to deal with it directly!

ASYNC

Simplifyinig a little, `async` is just syntax sugar for `impl Future<...>`, but allowing usage of `await`.

```
async fn try_give_3() -> Result<u8, String> {
    Ok(3)
}
```

```
fn try_give_3() -> impl Future<Item=Result<u8, String>> {
    future::ok(3)
}
```

ASYNC

But it can be also used on blocks, to make them futures.

```
let three = async {
    3
};
```

```
let three = future::ready(3);
```

AWAIT

Again simiplifying, `await` is replacement for `and_then`/`map`, but cleaner.

```
let twitts_fut = async {
    let body = fetch_url("www.rust-wroclaw.pl").await.body;
    let twitter = find_twitter(body);
    let twitts = fetch_twitts(twitter).await;
};
```

```
let twitts_fut = fetch_url("www.rust-wroclaw.pl").await.body
    .map(|body| find_twitter(body))
    .and_then(|twitter| fetch_twitts(twitter))
```

AWAIT

But it also makes branching easy.

```
let requests = requests_stream();
let _ = async {
    pin_mut!(requests);
    while let Some(req) = requests.next().await {
        let resp = process(req).await;
        if let Some(error) = last_system_error() {
            send_log(error).await;
        }
        send_response(resp).await;
    }

    finalize().await
};
```

Excercise: do it with continuators (this is actually possible).

AWAIT

And it also helps with borrowing.

```
let msg = message_to_be_send();
let _ = async move {
    log_msg(&msg).await;
    send_msg(&msg).await;
    wait_resp(&msg).await
}
```

Excercise: do it with continuators.

AWAIT

```
enum FutStage<'a> {
    BeforeLog(&'a Message),
    BeforeSend(LogMsg<'a>),
    BeforeWait(SendMsg<'a>),
    WaitingResp(WaitResp<'a>),
}

struct Fut {
    msg: Message,
    stage: FutStage<'???>,
}
```

This may be possible in future, with Polonius, but it is not for now, but `async/await` can figure out lifetime for `FutStage` safely - just because it may ensure `msg` will never move.

ASYNC/AWAIT

`Async/await` is commonly traeted just like syntax sugar, and making code cleaner is probably the most important benefit of it. However it is good to have in mind, that it also prevents for unnecessary overhead, like obsolete clonning (which is commonly avoided by `Arc`, but `Arc` is an overhead on its own).

TASK

Future which is constantly polled via executor. Actual equivalent of thread in parallel world.

TASK

```rust
struct JoinHandle<T> { /* ... */ }

impl Future for JoinHandle {
    type Item = Result<T, ...>;
    // ...
}
```

TASK - SPAWNING ASYNC

```rust
fn spawn<T>(task: T) -> JoinHandle<T::Output>
where
    T: Future + Send + 'static,
    Future::Output: Send + 'static
{ /* ... */ }
```

TASK

```rust
async fn handle_client(
    stream: impl Stream<Item=Req>,
    sink: impl Sink<Resp>,
) {
    let resps = stream.map(|req| process(req));
    stream.forward(sink)
}

async fn handle_server(stream: impl Stream<Item=Client>) {
    let Some(client) = stream.next().await {
        let hdl = handle_client(client.stream, client.sink);
        tokio::spawn(hdl).await.unwrap();
    }
}
```

TASK - SPAWNING BLOCKING

```
fn spawn_blocking<F, R>(f: F) -> JoinHandle<R>
where
    F: FnOnce() -> R + Send + 'static,
    R: Send + 'static
{ /* ... */ }
```

CASE STUDY

1. Send message
2. Wait response
    1. If future resolves, forward the result
    2. If timeout occured before response is received:
        1. If there were less than 3 attempts, goto 1)
        2. Otherwise return error
3. Return response

DESIGN

- `register` method setups some synchronization primitive for waiting for response
- `send` method sends message
- `cancel_response` removes any primitives needed for response waiting
- function should not block - if it will, it will be executed on dedicated thread

PARALLEL SOLUTION

```rust
fn send_request(&self, message: &Msg) -> Result<Msg, E> {
    let (cvar, mutex) = self.register(message.id);

    for _ in 0..3 {
        self.send(message.clone())?;
        let (lock, resp) =
            cvar.wait_timeout(
                    mutex.lock().unwrap(),
                    Duration::from_secs(3)
                )
                .unwrap();

        if !resp.timed_out() {
            self.cancel_response(message);
            return Ok(self.get_response(lock));
        }
    }

    self.cancel_response(message);
    Err(E::Timeout)
}
```

PROS

- Fairly simple both to read and write

CONS

- Involves new thread for every request
- Synchronizations is a bit nasty

ASYNC SOLUTION WITH CONTINUATORS

```
fn send_request(&self, message: &Msg) -> impl Future<Output=Result<Msg, E>> {
    let response = self.register(message.id);
    let shared = self.clone();

    let retransmit = stream::unfold((), move |_| {
        shared.send(message.clone())
            .and_then(tokio::time::delay_for(TIMEOUT))
            .map(|_| Some(((), ())))
    })
    .take(3)
    .try_for_each(|_| future::ready(()))
    .then(|_| feature::ready(Err(E::Timeout)));

    let shared = self.clone();
    select(response, retransmit)
        .map(|resp| resp.factor_first())
        .inspect(|_| shared.cancel_response())
}
```

PROS

- Threads are controlled by runtime

CONS

- WTF/min count
- Additional shared pointer is introduced - it's obsolete

ASYNC SOLUTION WITH ASYNC/AWAIT

```rust
async fn send_request(&self, message: &Msg) -> Result<Msg, E> {
    let response = self.register(message.id);

    let retransmit = async {
        let i = tokio::interval(TIMEOUT);
        for _ in 0..3 {
            self.send(message.clone()).await?;
            i.tick().await;
        }
        Err(E::Timeout)
    };

    let res = select(response, retransmit)
        .await
        .factor_first();

    self.cancel_response(message.id);

    res.await
}
```

PROS

- Looks very straightforward
- Threads are controlled by runtime
- No unnecessary overhead

CONS

- New syntax to get used to

PROBLEMS

There is no syntax for async closures… yet.

Proposed syntax (`async_closure` in nighlty):

```
async |_| { /* ... */ }
```

Workaround:

```
move |_| async move { /* ... */ }
```

USEFULL CRATES

- Futures-rs
- Tokio
- Async-std
- Async-stream
- Async-std

FUTURES-RS

- Futures continuators
- Streams combinators
- Tools for easy creation of own Futures/Streams
- Basic synchronization primitives

TOKIO

- Runtime
- IO Streams (FS, Net, Signals)
- Time handling
- Less basic synchronization primitives

TOKIO

```rust
#[tokio::main]
async fn main() {
    // ...
}
```

```rust
#[tokio::test]
async fn test() {
    assert_eq!(3, foo().await.unwrap())
}
```

ASYNC-STD

Kind of mariage of Futures-RS and Tokio, but pretends to mimic std.

ASYNC-STD

```rust
#[async_std::main]
async fn main() {
    // ...
}
```

```rust
#[async_std::test]
async fn test() {
    assert_eq!(3, foo().await.unwrap())
}
```

ASYNC-STREAM

Allows to create custom streams very easly.

```
let s = stream! {
    for i in 0..3 {
        yield i;
    }
};
```

PIN PROJECT

Allows reasonable cooperation with `Pin`.

```
#[pin_project]
struct Struct<T, U> {
    #[pin]
    pinned: T,
    unpinned: U,
}

impl<T, U> Struct<T, U> {
    fn foo(self: Pin<&mut Self>) {
        let this = self.project();
        let _: Pin<&mut T> = this.pinned;
        let _: &mut U = this.unpinned;
    }
}
```

QUESTIONS?

THANK YOU

Find me on github:

https://github.com/hashedone/

Find me on Rust Wrocław:

http://www.rust-wroclaw.pl/