P2300 Overview

C++ LEWG
2024-01-23

Dietmar Kühl

Bloomberg Engineering

https://github.com/dietmarkuehl/p2300-overview

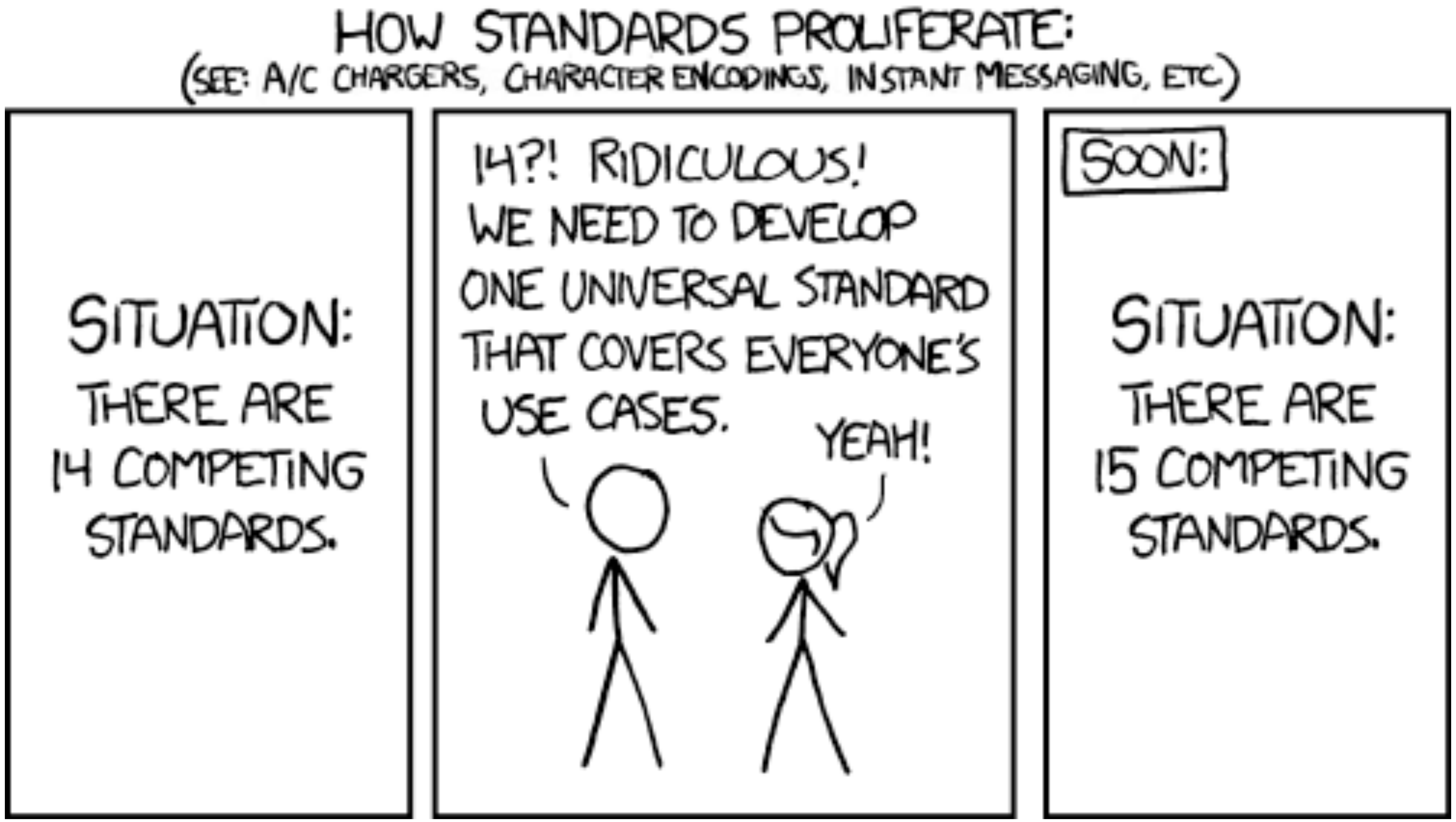
TechAtBloomberg.com

Many Forms of Async Work

- Concurrent work on threads or custom hardware
- Any form of I/O:
  - Network interaction, file access
  - Data base queries, service requests, subscriptions, etc.
  - User input, system events

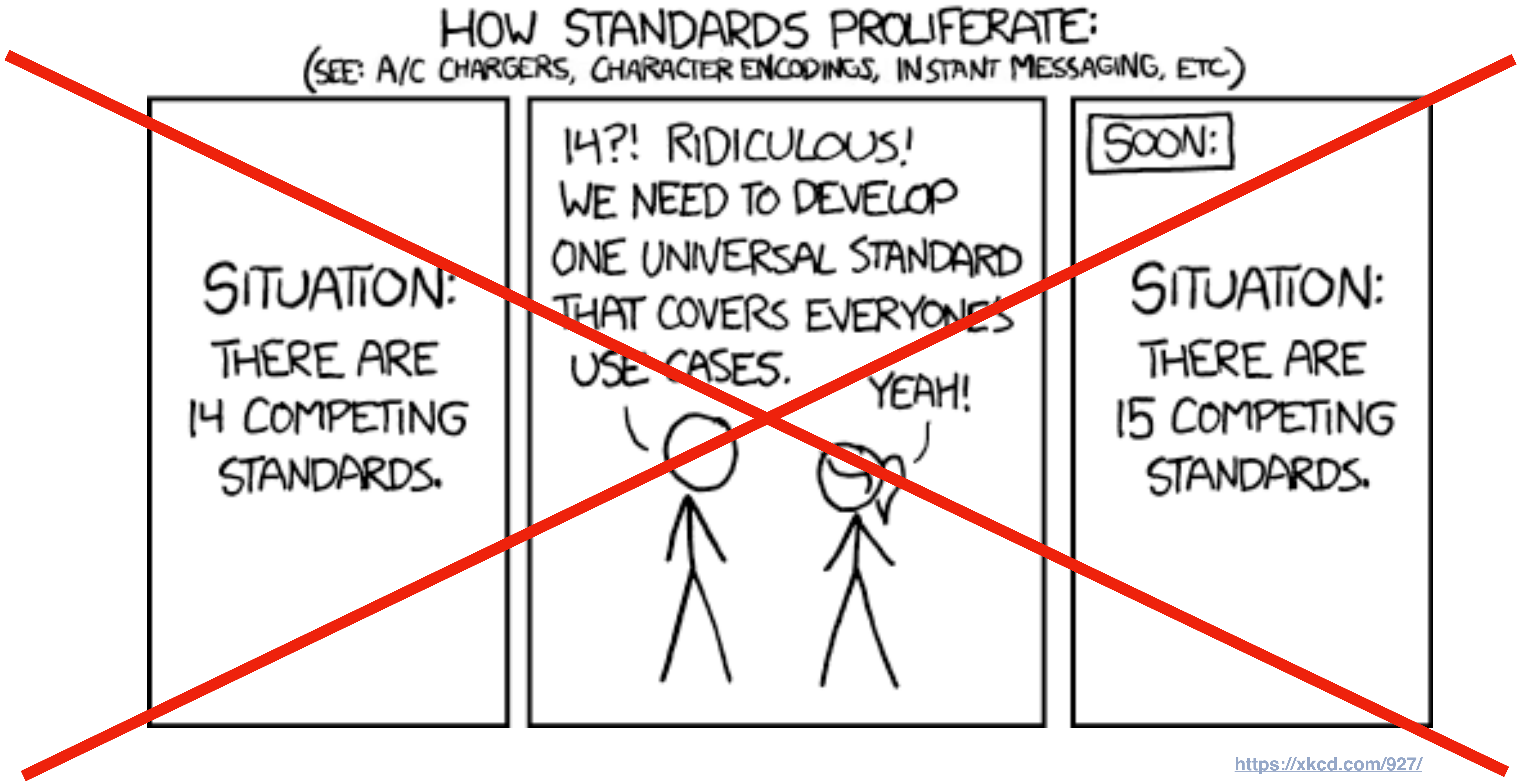
Various Async Interfaces

- std::async(…) and std::future/std::promise
- Callbacks to notify completion (quite common for C APIs)
- Completion handlers (e.g., for ASIO)
- Functions blocking on a notification (e.g., poll(…), cv.wait())
- Coroutines (co_await)



https://xkcd.com/927/

No Async Standard C++ Interface



https://xkcd.com/927/

P2300: One To Bind Them All

1. decompose work into senders each representing work
2. combine the work representation with a continuation: receiver
3. start the resulting operation state to execute the work

Example

```
thread_pool      p(…);
scheduler auto sched = p.scheduler();

sender auto s = when_all(
        schedule(sched) | then([]{ return frob(); }) | then([](auto x){ return borf(x); }),
        schedule(sched) | then([]{ return compute(); }));

auto[x, y] = sync_wait(move(s)).value();
```

P2300 User Interface

- auto result = sync_wait(move(sender)); //blocking wait
- auto result = co_await move(sender);   // suspending coroutine
- async_scope scope(…);
  …
  scope.spawn(move(sender));   // start work asynchronously
- auto sender2 = then(move(sender), fun); // compose work

TechAtBloomberg.com

Bloomberg
Engineering

What About All Those Moves?

- Often work is unique in some form
  ⇒ “single shot”

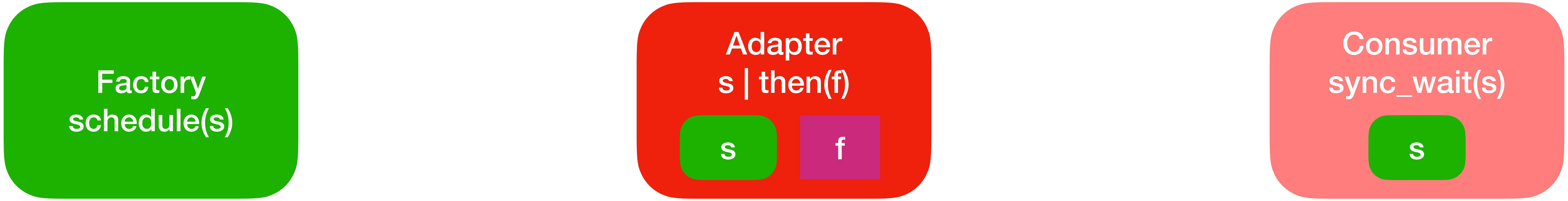
  ⇒ moves are required

- Repeatable work can be copyable
  ⇒ “multi shot”

  ⇒ no need to move work

- In general assume work is single-shot

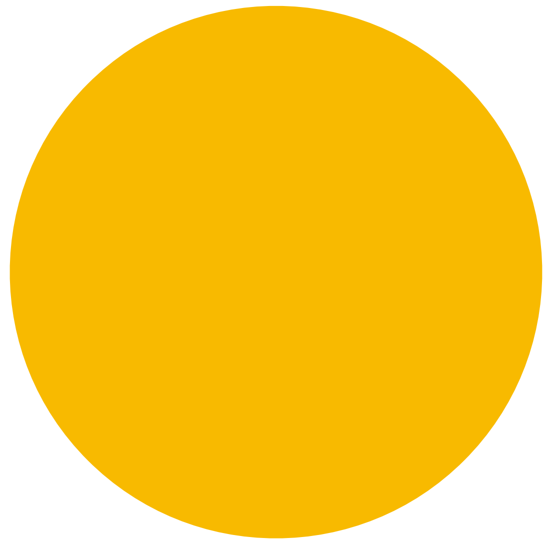
Sender: Description of Work



connect(sender, receiver) -> operation_state

get_completion_signatures(sender, env)

Receiver: Destination for Results



get_env(receiver) -> env

get_stop_token(env)

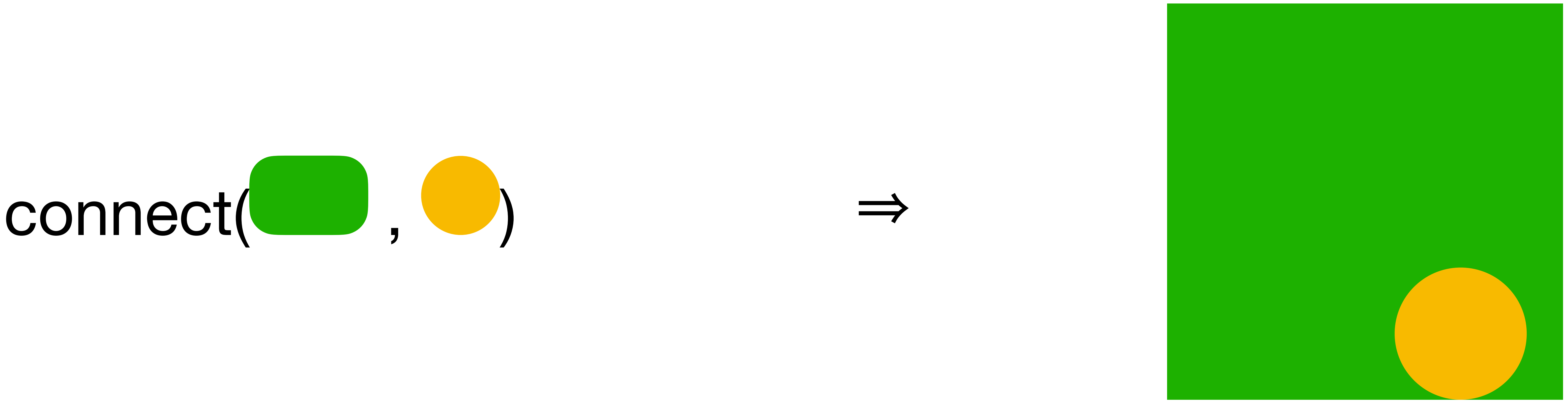
get_allocator(env)

set_value(receiver, results…)

set_error(receiver, error)

set_stopped(receiver)

Operation State: Ready to Execute Task



start(operation_state) ⇒ eventually one of the completions is called

Operation State: in place

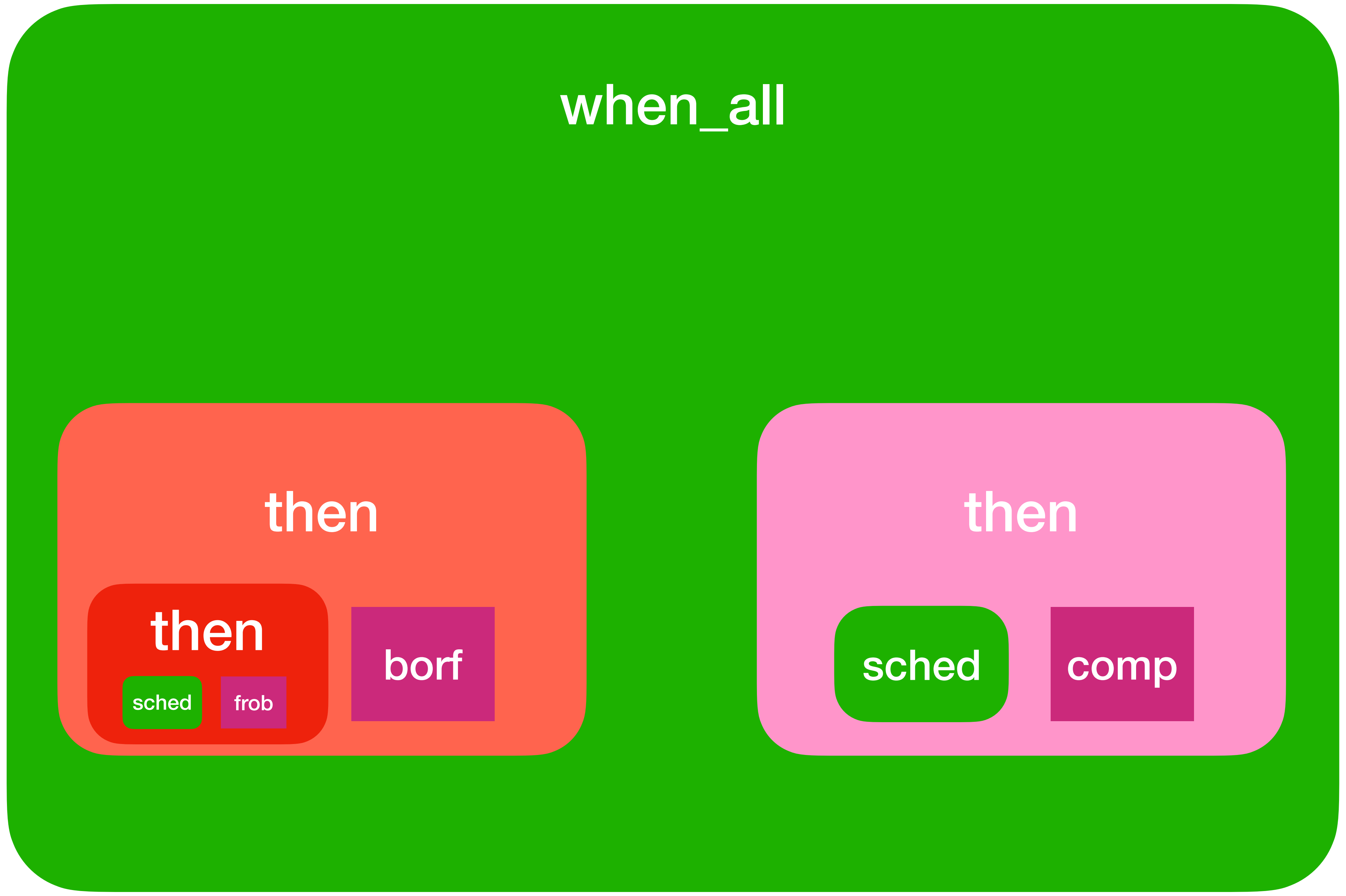
- Operation states are not required to be copyable or movable!
- Creation requires guaranteed copy elision:

```
State os(connect(sender, receiver));

State* os = new State(connect(sender, receiver));

struct h { State os; h(…): os(connect(sender, receiver)) {} … };
```

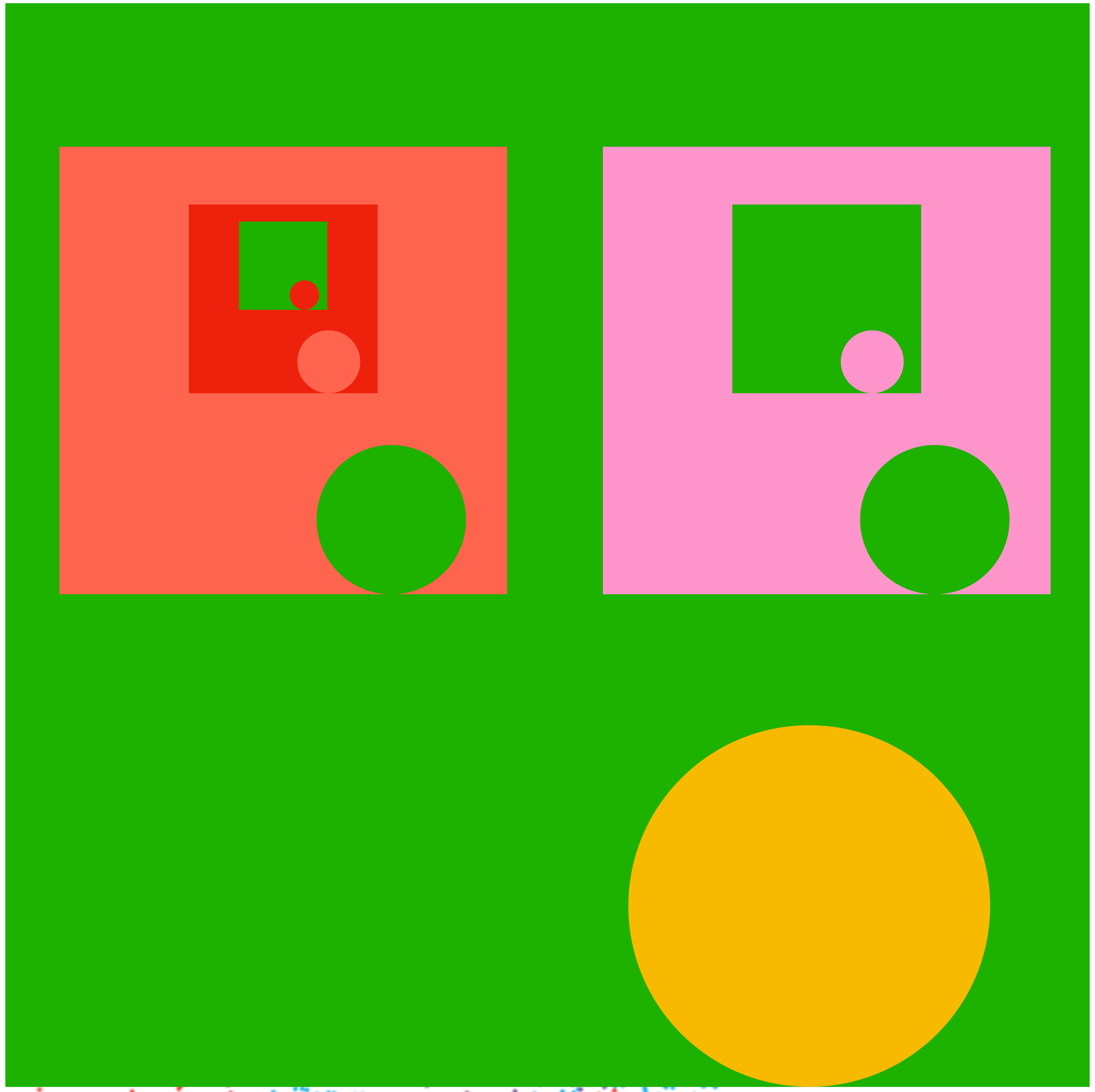
Sender/Receiver Contract

- Connecting a sender and a receiver yields an operation state.
- Once an operation state is started it has to remain alive until a completion signal is called:
  - set_value(r, v…), set_error(r, e), or set_stopped(r)
- A started operation state is expected to eventually call a completion signal.

Sender



Operation State



Cancellation

- All senders are expected to use the receiver's stop_token:
  - Active work should sometimes test stop_requested().
  - Inactive/"blocking" work should register a stop callback.
- Cancellation is configured from the consumer via the receiver:
  - never_stop_token never requests cancellation.
  - in_line_stop_token/stop_token can request cancellation.

Various Sender Algorithms

- Continuation: then(s, fun), let_value(s, fun)
- Fork/join: split(s), when_all(s…)
- Parallel execution: bulk(s, size, fun)
- Control scheduler: transfer(s, scheduler), on(scheduler, s)
- Rewrite Result: upon_error(s, fun), upon_stopped(s, fun)

Algorithms Are CPOs

- Algorithms can be customised to deal with different schedulers:
  - Schedulers for thread pools, GPU, FPGA, etc.
- P2999 adds domains and transform_sender.
  - Analogous to coroutines's await_transform

Senders Can Be Awaitables

- set_value(r, arg…) becomes co_await result
- set_error(r, e) becomes an exception thrown from co_await
- set_stopped(r) terminates the entire coroutine
- Alternatively a sender can also be an awaitable
    - Implement operator co_await()

Coroutines Can Be Senders

- co_yield/co_return become set_value(r, a…)
- An exception becomes set_error(r, e) or set_stopped(r)
- Coroutines can be cancelled by destroying them

General Features

- The framework doesn’t impose the need to allocate
- Cancellation is integrated with the framework
- Customisations are injected from the usage end via the receiver
- The abstraction allows algorithms to implemented
- Sender are easy to use: complexity is absorbed by the library

P2300 Enhancement

- P2999/Sender Algorithm Customisation
  - Add domains describing on how work may be scheduled
  - Add optional sender decomposition and transform_sender(…)
- Approved by LEWG electronic poll

In-Flight Work

- P2855/Member customisation points for S/R
  - Replace tag_invoke by member functions
- P2500/C++ parallel algorithms and P2300
  - Support schedulers as arguments to parallel algorithms
- P2079/System execution context

P2855 Change

connect(sndr, rcvr)	sndr.connect(connect, rcvr)
set_value(rcvr, args…)	rcvr.set_value(set_value, args…)
set_error(rcvr, error)	rcvr.set_error(set_error, error)
set_stopped(rcvr)	rcvr.set_stopped(set_stopped)
start(state)	state.start(start)
get_stop_token(env)	env.tag_query(get_stop_token)

Outstanding Work

- async_scope:
  - Context (scope) starting senders and cleaning up states.
  - Signals completion when all work is completed.
- Sender progress: set_next()
  - A signal yielding a sender to report progress of sender.