# Of Operation States and Their Lifetimes

## P3373R0

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca


LSEG

# `std::execution` Generalizes Function Calls

**Decomposing the function call**

- Sender represents the function to call, and the arguments thereto
- Operation state represents the stack frame (variables with “automatic storage duration” are members thereof)
- Receiver represents the return

# Regular Function Call

This outputs 8

```
int foo(int i) {
    return i + 1;
}

int bar(int i) {
    return i + 2;
}

int corge(int i) {
    return bar(foo(i));
}

int main() {
    std::cout << corge(5) << std::endl;
}
```

# Stack Frames

Note that the stack frames of `foo` and `bar` overlap

```
int foo(int i) {
    return i + 1;
}

int bar(int i) {
    return i + 2;
}

int corge(int i) {
    return bar(foo(i));
}

int main() {
    std::cout << corge(5) << std::endl;
}
```


LSEG

# Asynchronous Equivalent-ish

This also outputs 8

```
auto foo(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 1;
        });
}

auto bar(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 2;
          });
}

auto corge(std::execution::sender auto&& i) {
    return bar(
        foo(
            std::forward<decltype(i)>(i)));
}

int main() {
    std::cout << std::get<0>(
        std::execution::sync_wait(
            std::execution::just(5)).value()) << std::endl;
}
```

LSEG

# Operation States

Using `then` repeatedly is the simplest asynchronous transformation but isn't directly isomorphic, so it's not surprising that the operation states are nested like this since `foo` returns to `bar` which returns to `corge` since senders are passed around

```
auto foo(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 1;
        });
}

auto bar(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 2;
          });
}

auto corge(std::execution::sender auto&& i) {
    return bar(
        foo(
            std::forward<decltype(i)>(i)));
}

int main() {
    std::cout << std::get<0>(
        std::execution::sync_wait(
            std::execution::just(5)).value()) << std::endl;
}
```


LSEG

# Asynchronous Equivalent

Another example that outputs 8, but this one isomorphic to the synchronous example

```
auto foo(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 1;
        });
}

auto bar(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 2;
        });
}

auto corge(std::execution::sender auto&& i) {
    return
        foo(std::forward<decltype(i)>(i)) |
        std::execution::let_value([](int i) {
            return bar(std::execution::just(i));
        });
}

int main() {
    std::cout << std::get<0>(
        std::execution::sync_wait(
            std::execution::just(5)).value()) << std::endl;
}
```

LSEG

# Operation States

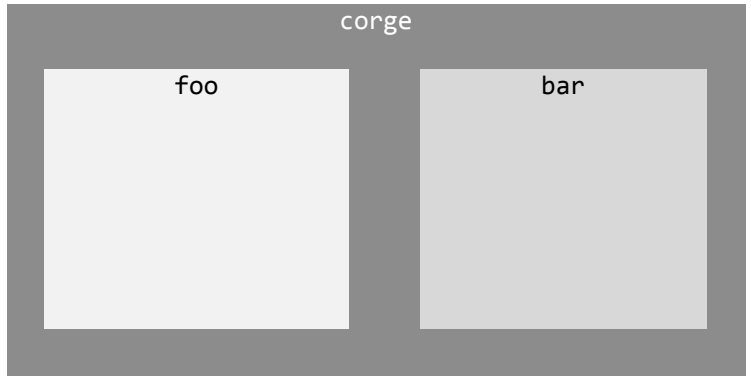
Unlike in the synchronous example the “stack frame” (in the asynchronous world the “operation state”) for **foo** is not freed and therefore is not reused for **bar**

```
auto foo(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 1;
        });
}

auto bar(std::execution::sender auto&& i) {
    return
        std::forward<decltype(i)>(i) |
        std::execution::then([](int i) {
            return i + 2;
        });
}

auto corge(std::execution::sender auto&& i) {
    return
        foo(std::forward<decltype(i)>(i)) |
        std::execution::let_value([](int i) {
            return bar(std::execution::just(i));
        });
}

int main() {
    std::cout << std::get<0>(
        std::execution::sync_wait(
            std::execution::just(5)).value()) << std::endl;
}
```



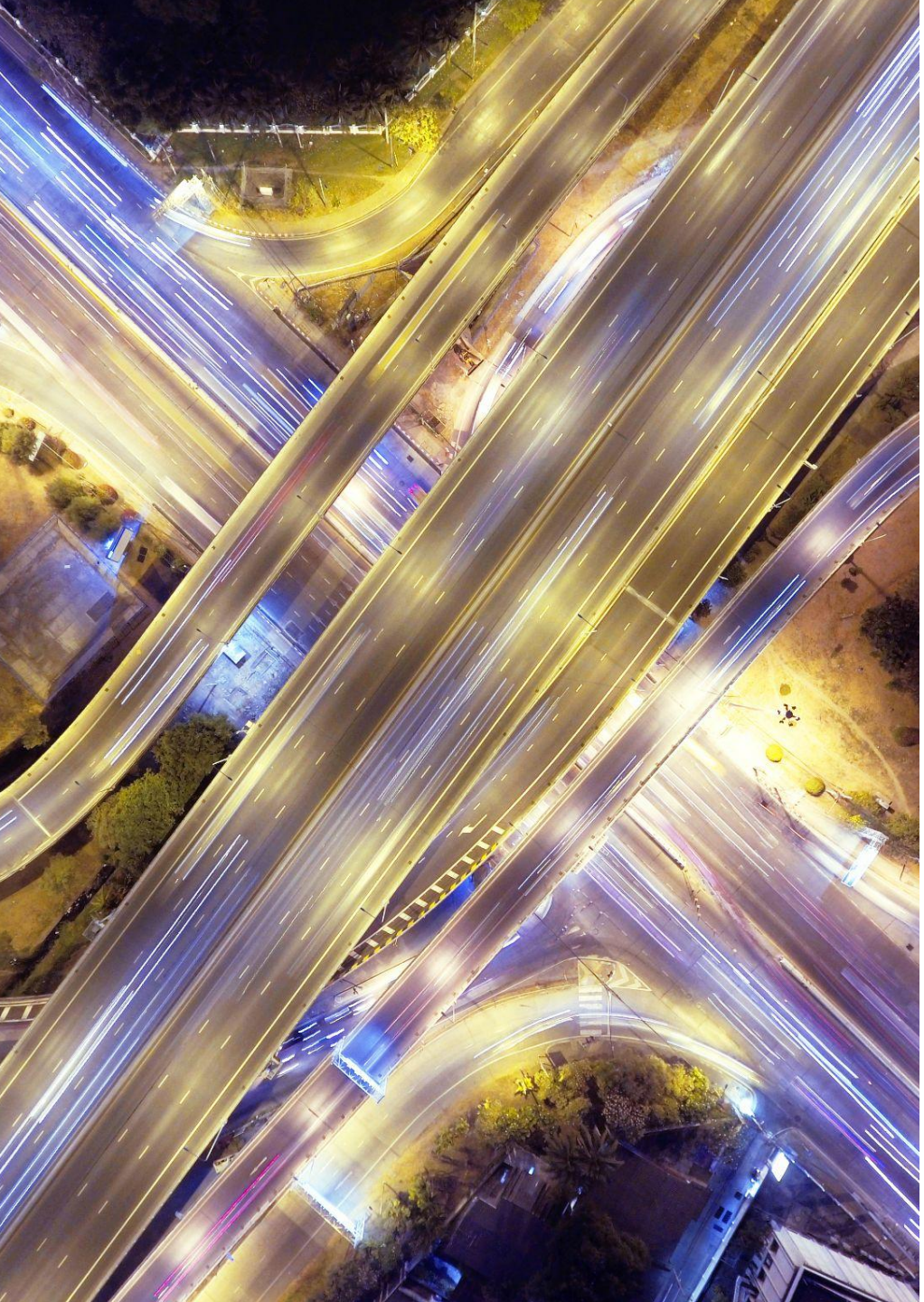

LSEG

# Mandated Behavior

**Standardizing equivalence to code**

P2300 specifies almost everything as being equivalent to code (or something approximating C++ code) which has the effect of standardizing nearly every minute detail of its behavior

In the case of operation state lifetimes it has seemingly standardized the maximum possible lifetime, which means operation state lifetimes must persist for the full, containing asynchronous operation, perhaps even long after that part of the operation has ended (contrast with regular, synchronous function calls where the stack frame is cleaned up immediately upon return)



# Alternatives

**Do we change the status quo?**

- **Maximal lifetimes (status quo):** Operation state lifetimes persist until the overall operation state's lifetime ends (note that this can be interpreted as extending the receiver contract with the addition of a signal indicating when the overall operation has ended)
- **Minimal:** Operation state lifetimes end immediately upon the completion of the operation (implementer overhead)
- **Ad hoc:** We decide on an operation-by-operation bases
- **Implementation-defined:** Has a Hyrum's Law problem

# Discuss

Robert Leahy
Lead Software Engineer
rleahy@rleahy.ca


LSEG