Of Operation States and Their Lifetimes

Document Number: P3373R0
Date: 2024-08-14
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG

Abstract

This paper is intended as a vehicle for considering operation states [1] and the lifetimes thereof in the context of operations which compose them.

Background

Operation States and Stack Frames

In broad terms a regular (i.e. synchronous) function call has access to two forms of storage throughout its lifetime (note that the “lifetime” of a regular function call is the time between the call thereto and the return therefrom):

- Stack storage (i.e. “variables with automatic storage duration”)
- Heap storage (i.e. “variables with dynamic storage duration”)

Asynchronous operations within the framework of `std::execution` analogously have access to two forms of storage throughout their lifetime (note that the “lifetime” of an asynchronous operation is the time between a call to `std::execution::start` on the operation state and the fulfillment of the “receiver contract”):

- Contents of the operation state
- Heap storage (note this is identical to the synchronous case)

Note that asynchronous operations may, depending on their form, have access to stack storage at points throughout their execution but it is not, in general, available throughout their lifetime.

The analogues between components of a regular function call and an asynchronous operation do not end there. A regular function call consists of:

1. Call (i.e. the synchronous surrender of control of the thread of execution thereto by the caller)
2. Execution (i.e. the use of the thread of execution to perform the desired task)
3. Either
    - Error (transmitted via exception)

    - Success (transmitted via the return of exactly one homogeneously-typed value (note that sum (`std::optional`, `std::expected`, et cetera) and product (`std::pair`, `std::tuple`, et cetera) types permit the *de facto* transmission of multiple heterogeneously-typed values)

Whereas an asynchronous operation consists of:

1. Initiation (i.e. `std::execution::start`)
2. Execution (note that in the model of `std::execution` this may be synchronous or asynchronous whereas other models of asynchronous computation do not permit the former ([2] at §13.2.7.6))
3. Satisfaction of the receiver contract by exactly one of:
    - Error (transmitted via `std::execution::set_error`)
    - Success (transmitted via the transmission of potentially multiple, potentially heterogeneously-typed values via `std::execution::set_value`)
    - Stopped (transmitted via `std::execution::set_stopped`) (note that this has no direct synchronous analogue [3])

We could continue with these analogues until each element of an asynchronous operation within the framework of `std::execution` has a synchronous analogue.

Despite the elegance of the above-described analogues there is one area in which they lack predictive power: The lifetime of the operation state. By the above-described analogy one would expect that the lifetime of the operation state is ended by the invocation of `std::execution::set_value`, `::set_error`, or `::set_stopped` (since when a regular function returns or throws the lifetime of variables with automatic storage duration ends) however this is not guaranteed to be the case in general and in fact `std::execution` appears to guarantee exactly the opposite.

Directed Graphs

Both synchronous and asynchronous code can be expressed in terms of a directed graph (note that the possibility of recursion precludes expression in terms of a directed acyclic graph). The direction of the edges in this graph depends on the relationship one chooses to model:

- Calls: The edge is directed from the caller to the callee
- Returns to: The edge is directed from the callee to the caller

Note that in the context of std::execution the latter relationship causes vertices (i.e. asynchronous operations) to have edges directed towards the asynchronous operation which owns their operation state and is therefore the convention which will be useful for our analysis.

Relationships Between Operations

std::execution contains two distinct syntaxes for creating and composing senders (and, by extension, the asynchronous operations initiated thereby).

The first is regular function call syntax, for example:

```
std::execution::then(
  std::execution::just(5),
  [](const int i) noexcept { return float(i); })
```

With this syntax our returns-to relationship (see above) is directed from the inside to the output, or from right to left. Note that it has been claimed that control and data flow in this model is more difficult to understand [4].

The second is pipe syntax, for example (equivalent to the above modulo currying overhead):

```
std::execution::just(5) |
  std::execution::then([](const int i) noexcept { return float (i); });
```

With this syntax our returns-to relationship is directed along the pipe, or from left to right. This alignment between how the code is typed and the direction of our relationship indicates to us one of the motivations for this syntax: It intuitively expresses data flow through the resulting overall operation.

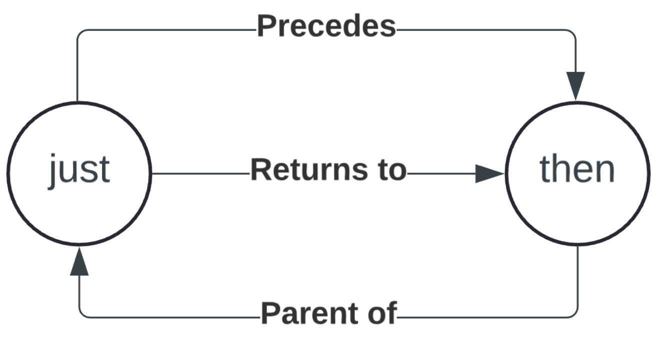
For simple, linear pipelines everything can be expressed using only either of the above syntaxes. In this case the choice of which to use is purely one of preference or convention (again modulo currying overhead). We can use this to introduce two relationships in addition to our returns-to relationship (note that in this simple case these relationships will be isomorphic to returns-to but this will stop being the case as we continue our exploration):

- Predecessor/successor: An operation is the "predecessor" of another operation when its sole interaction therewith is to send values thereto
- Parent/child: An operation is the "parent" of another operation when it owns the operation state thereof

In our above example:

- `std::execution::just` returns to `std::execution::then`
- `std::execution::just` precedes `std::execution::then`
- `std::execution::then` parent of `std::execution::just`

Visually:



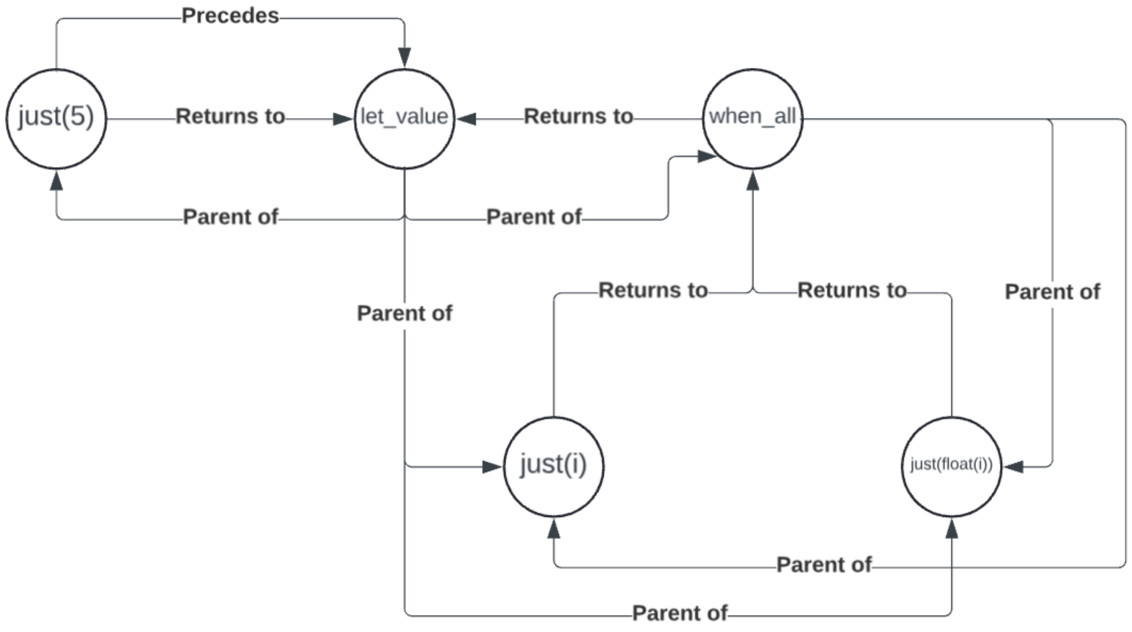
As pointed out above this example is simple. `std::execution::just` simply produces a value. `std::execution::then` simply transforms the value produced by its predecessor. `std::execution` contains operations with more complicated relationships which start to illustrate the difference between these relationships, for example:

```
std::execution::just(5) |
  std::execution::let_value([](const int i) noexcept {
    return std::execution::when_all(
        std::execution::just(i),
        std::execution::just(float(i)));
  })
```

In the above example `std::execution::just` has the following relationships with `std::execution::let_value`:

- Returns to
- Precedes
- Child of

But beyond that the relationships become much more complicated. `std::execution::let_value` is the parent of `std::execution::when_all` (which in turns returns to it) but is not preceded or succeeded thereby. `std::execution::let_value` has transitive parent-of relationships with operations that do not return to it (since parent-of is transitive whereas returns-to is not). This example is sufficiently complicated that describing the relationships is far less expressive than visualizing them (and this example is not even that complicated).



Note that with the introduction of these more complicated relationships the pipe syntax was insufficient. The fact that it was not used exclusively in the above example was not a stylistic choice rather some of the relationships therein cannot be expressed thereby.

Examples

std::execution::then

```
struct print_from_destructor {
  ~print_from_destructor() noexcept {
    std::cout << "Destructor" << std::endl;
  }
};
auto ptr = std::make_unique<print_from_destructor>();
struct receiver {
  using receiver_concept = std::execution::receiver_t;
  void set_value(const int& i) noexcept {
    std::cout << "*" << &i << " = " << i << std::endl;
  }
  void set_error(std::exception_ptr) noexcept {}
  void set_stopped() noexcept {}
};
```

```
const auto scheduler = ctx.get_scheduler();
auto op = std::execution::connect(
  std::execution::just() | std::execution::then([
    p = std::move(ptr),
    vec = std::vector<int>{1, 2, 3}]() noexcept
  {
    std::cout << &vec.front() << std::endl;
    return std::cref(vec.front());
  }) | std::execution::continue_on(scheduler),
  receiver{});
std::execution::start(op);
ctx.run();
```

Running against a reference implementation of `std::execution` [5] this outputs the following:

```
0x6020000000f0
*0x6020000000f0 = 1
Destructor
```

Using the previously-described analogues with regular function calls we can write a synchronous analogue of this code:

```
const auto a = []() {
  const print_from_destructor print;
  const std::vector<int> vec{1, 2, 3};
  std::cout << &vec.front() << std::endl;
  return std::cref(vec.front());
};
const auto b = [](const int& i) {
  std::cout << "*" << &i << " = " << i << std::endl;
};
b(a());
```

This code outputs:

```
0x602000000110
Destructor
```

Followed by a lengthy AddressSanitizer diagnostic due to the fact it has undefined behavior.

Alternate Synchronous Analogues

One could make an argument that the following synchronous code (which does not have undefined behavior) is also an analogue of the asynchronous example being discussed above:

```
const auto a = [
  g = print_from_destructor{},
  vec = std::vector<int>{1, 2, 3}]()
{
  std::cout << &vec.front() << std::endl;
  return std::cref(vec.front());
};
const auto b = [](const int& i) {
  std::cout << "*" << &i << " = " << i << std::endl;
};
b(a());
```

Another alternate interpretation is:

```
[](const int& i) {
  std::cout << "*" << &i << " = " << i << std::endl;
}(
  [
    g = print_from_destructor{},
    vec = std::vector<int>{1, 2, 3}]()
  {
    std::cout << &vec.front() << std::endl;
    return std::cref(vec.front());
  }());
```

Note that this still does not have undefined behavior due to the fact the lifetimes of temporaries are extended to the end of the enclosing *full-expression*. However the above hints to another alternate interpretation:

```
auto&& i = [
  g = print_from_destructor{},
  vec = std::vector<int>{1, 2, 3}]()
{
  std::cout << &vec.front() << std::endl;
  return std::cref(vec.front());
}();
[](const int& i) {
  std::cout << "*" << &i << " = " << i << std::endl;
}(i);
```

Which does have undefined behavior.

`std::execution::let_value`

```
template<typename Writable>
auto write(Writable& writable, std::span<const std::byte> span) noexcept {
  return std::execution::just(std::move(span)) |
    std::execution::let_value([&writable](std::span<const std::byte>& span)
      noexcept
    {
      return ::exec::repeat_effect_until(
        std::execution::just() |
          std::execution::let_value([&writable, &span]() noexcept(
            noexcept(writable.write(span)))
          {
            return writable.write(span);
          }) |
          std::execution::then([&span](const std::size_t bytes_transferred)
            noexcept
          {
            span = span.subspan(bytes_transferred);
            return span.empty();
          }));
    });
}
```

The intention of the above is to repeatedly perform possibly-partial writes until:

- The desired number of bytes has been written,
- An error occurs, or
- Cancelation is requested

Note that `repeat_effect_until` is not part of `std::execution` as of this writing but is provided as an extension by the reference implementation [5] which was used in the preparation of this paper.

Discussion

`std::execution::then`

Upon first consideration the behavior of the example involving `std::execution::then` (see above) seems acceptable. The lack of UB stems from the fact that the lifetime of the invocable is bound to the lifetime of the operation state. The lifetime of the operation state persists until the end of the example because it's nested within the overall operation state and therefore elements within `vec` remain within their lifetime when they're used by successive asynchronous operations. In order for this example to have undefined behavior one of the following would need to be true:

- The implementation of `std::execution::then` explicitly ends the lifetime of the invocable upon which it is parameterized sometime after the invocation thereof
- The implementation of `std::execution::continue_on` goes out of its way to end the lifetime of the operation state of its predecessor upon satisfaction of the receiver contract thereof

Both of which would involve additional implementation complexity for no immediately-obvious gain.

Consideration of the analogues with synchronous code, however, can cast this in a different light: Since `std::execution::then` returns to `std::execution::continue_on` we would expect, by analogy with synchronous code, for `std::execution::continue_on` to end the lifetime of the operation state for `std::execution::then` in the same way that returning from a synchronous function destroys the stack frame thereof.

Extrapolating from the above we can contrast the intermediate storage requirements of regular synchronous C++ code and asynchronous code under the framework of `std::execution`. Given a directed graph modeling the returns-to relationship (see above) the synchronous code requires intermediate storage equal to the maximum sum across a set of sums obtained by summing the intermediate storage requirements of each vertex across all walks through that graph (note that it follows from this that unbounded recursion, modeled as a cycle in the graph, implies an infinite intermediate storage requirement). If such a graph models the returns-to relationship for asynchronous code under the framework of `std::execution`, however, the intermediate storage is equal to the sum across all vertices (cycles are not considered).

Put differently: Whereas the stack pointer for regular synchronous code moves both up and down ("allocating" and "freeing" stack storage) the analogue thereof for asynchronous code under the framework of `std::execution` moves in only one direction (i.e. such storage is only ever "allocated").

Note that linear asynchronous operations (i.e. those for which the directed graph of the returns-to relationships thereof is connected) the above doesn't represent much of a difference. The lifetime of the leaf operation states exists longer but the storage requirements aren't increased as there's one walk which includes all vertices (i.e. the graph is connected). As described above, however, there are operations which aren't linear.

`std::execution::let_value`

`std::execution::let_value`, unlike `std::execution::then`, has two direct child operations:

- The predecessor operation, and
- The operation spawned from the sender returned by the wrapped invocable (which is invoked with the value(s) yielded by the above)

These operations do not overlap in time and so their operation states could in principle overlap in storage however this technique is disallowed by the specification. `std::execution::let_value` is specified in terms of *`basic-operation`* which is required to be equivalent to ([1] at §34.9.1, emphasis added):

```
template<class Sndr, class Rcvr>
  requires valid-specialization<state-type, Sndr, Rcvr> &&
           valid-specialization<connect-all-result, Sndr, Rcvr>
struct basic-operation : basic-state<Sndr, Rcvr> {  // exposition only
  using operation_state_concept = operation_state_t;
  using tag-t = tag_of_t<Sndr>; // exposition only

  connect-all-result<Sndr, Rcvr> inner-ops; // exposition only

  basic-operation(Sndr&& sndr, Rcvr&& rcvr) noexcept(see below)  // exposition only
    : basic-state<Sndr, Rcvr>(std::forward<Sndr>(sndr), std::move(rcvr))
    , inner-ops(connect-all(this, std::forward<Sndr>(sndr), indices-for<Sndr>()))
  {}

  void start() & noexcept {
    auto& [...ops] = inner-ops;
    impls-for<tag-t>::start(this->state, this->rcvr, ops...);
  }
};
```

Which in turn leverages *`connect-all`* which is required to be equivalent to (ibid.):

```
[]<class Sndr, class Rcvr, size_t... Is>(
  basic-state<Sndr, Rcvr>* op, Sndr&& sndr, index_sequence<Is...>) noexcept(see below)
    -> decltype(auto) {
    auto& [_, data, ...child] = sndr;
    return product-type{connect(
      std::forward_like<Sndr>(child),
      basic-receiver<Sndr, Rcvr, integral_constant<size_t, Is>>{op})...};
  }
```

Where *`product-type`* is `std::tuple`-like (ibid.). As such the operation state for `std::execution::let_value` contains not only those values which it requires but also the operation states for all predecessors structured in such a way that the storage therefor cannot be reused until the completion of the overall operation.

In the context of our example involving `std::execution::let_value` this might initially seem fortuitous. After all if `std::execution::let_value` opportunistically destroyed the operation state for `std::execution::just` (for example to reuse the storage therefor for the child operation spawned from the sender returned by the invocable) surely:

- The lifetime of the `std::span<const std::byte>` contained thereby would end, and
- The overall operation (which depends on the ability to reference the aforementioned `std::span<const std::byte>`) would have undefined behavior?

Except this initially-intuitive analysis is incorrect. `std::execution::let_value` is completely free to destroy the operation state of `std::execution::just` without introducing undefined behavior into the example. The implementation in libunifex [6] does exactly this as of the time of this writing.

The reason for the above is that the stable reference the example relies upon is not to a value within the operation state of `std::execution::just` (i.e. the predecessor) but rather to a value within the operation state of `std::execution::let_value` (i.e. the parent). As such the lifetime of that value is not in question: By analogy with the synchronous world it's obvious that variables in the intermediate storage of the parent (in our analogy on the caller's stack) remain within their lifetime throughout execution of the child (in our analogy for the duration of the callee's execution).

For confirmation of this we look to the specification of `std::execution`. The completion of `std::execution::just` is specified to have the effect of ([1] at §34.9.10.2, emphasis added):

```
template<>
struct impls-for<decayed-typeof<just-cpo>> : default-impls {
  static constexpr auto start =
    [](auto& state, auto& rcvr) noexcept -> void {
      auto& [...ts] = state;
      set-cpo(std::move(rcvr), std::move(ts)...);
    };
};
```

Whereas reception of this value within `std::execution::let_value` is specified as interacting with the state thereof (id. at §34.9.11.8, emphasis added):

```
struct state-type {
  fn_t fn;     // exposition only
  env_t env;     // exposition only
  args_variant_t args;     // exposition only
  ops2_variant_t ops2;     // exposition only
};
```

By (ibid., emphasis added):

```
using args_t = decayed-tuple<Args...>;
auto mkop2 = [&] {
  return connect(
    apply(std::move(state.fn),
          state.args.template emplace<args_t>(std::forward<Args>(args)...)),
    receiver2{rcvr, std::move(state.env)});
};
start(state.ops2.template emplace<decltype(mkop2())>(emplace-from{mkop2}));
```

As such the effect of the current formulation (whereunder the operation state of `std::execution::just` remains within its lifetime) only has the effect of ensuring that an unused, moved-from `std::span<const std::byte>` remains within its lifetime within the operation state for `std::execution::just`.

Temporaries and Lifetime Extension

In the previous synchronous examples only certain permutations exhibited undefined behavior. Those examples which did exhibit well-defined behavior did so due to lifetime extension of temporaries. In general the lifetime of temporaries is extended to the end of the enclosing *full-expression*. This raises the question of whether or not the concept of a “full expression” exists in the asynchronous world of `std::execution`.

Due to the fact `std::execution` is a library rather than a language feature we can’t perform direct syntactic comparisons between the world of “regular” C++ code (with its *full-expressions* and temporary lifetime extension) and asynchronous, `std::execution` code (with its operation states). It is tempting however to see the pipe syntax used to chain asynchronous algorithms in the framework of `std::execution` and posit that such chains of asynchronous algorithms are an analogue of a *full-expression* (and that therefore by analogy temporary lifetime extension should apply thereto, thereby justifying the status quo).

There is however the possibility that this supposition only seems like a neat analogue because of the particular structure of the code used in the example: An entire overall asynchronous operation composed in a single *full-expression* through piping. Would the analogy seem as convincing or tempting if we rearranged the code so that several parts thereof were composed in functions or in separate *full-expressions* (with intermediate senders simply stored in a variable and later moved from to continue the composition)?

Networking TS

While the name of the Networking TS identifies networking as its primary concern it also contains an alternative asynchronous model to the one put forth by `std::execution` [7]. While efforts to adopt the model proposed thereby into the standard [8] have heretofore failed it still represents a large, wide-reaching piece of design work which did, at some point in time and in some form, gain consensus.

One criticism of the asynchronous model put forth by the Networking TS centers around allocation, the ability to control such allocation, and whether such allocation can be avoided (ibid. at §1.4). Whether this is endemic to the model is debatable [9] but whether it is or not the design direction of the Networking TS can still inform our discussion here in the context of `std::execution`.

The relevant consequences of the above are that asynchronous operations under the model of the Networking TS must ([2] at §13.2.7.11):

- Explicitly allocate all intermediate storage
- Deallocate all such intermediate storage before performing the “upcall” (i.e. the invocation of the “completion handler” which signals that an asynchronous operation is done)

This would be equivalent to the lifetime of the operation state for an operation ending immediately before fulfillment of the receiver contract (note that attempting to emulate the Networking TS’s behavior this strictly would be unworkable since an asynchronous operation within the framework of `std::execution` has no control over its operation state’s lifetime).

Note that one of the consequences of this design decision is that allocators may be provided to asynchronous operations with the assurance that all memory allocated therethrough will be available for reuse by a subsequent operation (this allows for simple (e.g. bump) allocators to be used).

Extending the Receiver Contract

`std::execution` goes to lengths to provide strong guarantees around the receiver contract. It is invalid under the framework of `std::execution` to allow the lifetime of an operation state to end once `std::execution::start` has been invoked thereupon unless the receiver contract has been satisfied.

Taking the approach that `std::execution` takes with the asynchronous algorithms it provides is tantamount to extending the receiver contract by providing a fourth signal which indicates when the most-enclosing asynchronous operation completes (whether via `set_value`, `set_stopped`, or `set_error`).

This fourth signal does not seem to be part of the intentional design surface of `std::execution`. We should explicitly decide whether this is a signal we want to provide (at least from the standard algorithms) and if not adjust what is being standardized to prevent the destructor of the operation state from being used in this manner.

Possible Designs

Status Quo

Most of the specification in `std::execution` is in the form of code that implementations must be equivalent to. This has the effect of prescribing the lifetimes of nested operation states. `std::execution` appears to consistently define these lifetimes as persisting until the end of the lifetime of the containing operation state. This defines the lifetimes thereof maximally which:

- Consumes a maximal amount of storage
- Causes perhaps initially-surprising accesses to have well-defined behavior
- Can cause surprises when an object nested within an operation state manages some resource (particularly a lock) via RAII (i.e. the resource extends long after the asynchronous operation which stored it completes)

Minimal

`std::execution` could be respecified to mandate that the operation states of nested operations are destroyed by the receiver thereof. As compared to the maximal approach this would:

- Consume a minimal amount of storage
- Cause a maximal number of accesses to have undefined behavior
- Scope resources managed by RAII types exactly to the containing operation

Note that the second point above may seem like a disadvantage however there are two classes of access which this change would render undefined:

- Intentional accesses (i.e. wherein the author thereof has a deep understanding of std::execution and knows that the access is well-defined regardless of how “surprising” that may be)
- Unintentional accesses (i.e. wherein the author thereof doesn’t possess an adequate mental model and simply happens to write well-defined code because of the fact `std::execution` happens to maximize the lifetime of nested operation states)

In the former case such programmers can simply write different code after this change is applied. They have a deep understanding of the framework, understand the patterns necessary to operate therein, and will doubtless find a method to appropriately extend lifetimes where and when they need to.

In the latter case such a change can actually be framed as an advantage. Just because the standard maximizes operation state lifetimes doesn't mean complementary operations supplied by users of the standard will follow this example. By allowing people to accidentally write well-defined code because the standard chooses to maximize nested operation state lifetimes we create the very real possibility that people will write fragile code which works with standard algorithms, but doesn't compose with non-standard algorithms. Note this could in turn become a safety issue due to the fact people will likely test their operations with standard algorithms (perhaps with sanitizers to alert them to undefined behavior) and may then later integrate those operations with non-standard algorithms with less rigor (believing the operations are sound due to previous experience testing with standard algorithms) leading to latent issues.

This approach has two disadvantages in that forcing the operation state's lifetime to end immediately after the associated operation completes:

- Requires explicit management of the lifetime of that operation state (for example by wrapping it in `std::optional`)
- May cause the lifetime of the receiver associated with the outer operation to end while within its `set_value`, `set_error`, or `set_stopped` member function

Note that both of these are simply implementation concerns, they don't have any user-facing impact.

Ad Hoc

The algorithms could be considered one-by-one with a separate nested operation state lifetime being determined for each of them. Initially this does not seem compelling as it variously suffers from all the issues of the previous two proposals and has the added disadvantage of the cognitive load it forces onto users. However one of our examples actually contains motivation for this approach in the form of `std::execution::continue_on`.

In the previous example involving `std::execution::then` were it the case that `std::execution::continue_on` exhibits the "minimal" behavior (see above) then it would have to *decay-copy* the value generated by `std::execution::then` so it could safely destroy the operation state thereof. Thereafter it would have to move that value to pass it to the value channel of the final receiver thereby introducing a copy or move where none needed to exist.

Rather than clinging firmly to the minimal approach we could arrive at a principle whereunder operations destroy their predecessor's operation state as soon as they are finished with the values generated thereby. For cases like `std::execution::continue_on` (wherein the values generated by the predecessor are simply passed through and therefore the operation is not “finished with” them for its duration) this would defer responsibility for the destruction to the successor thereof (note that this would reify itself as the successor simply destroying its predecessor’s operation state which would, transitively, destroy the predecessor’s predecessor’s operation state).

If we want to continue the analogies with synchronous code as has been done throughout this paper we could think of the above as a workaround for the fact that RVO does not exist in the asynchronous world.

Implementation-Defined

The standard could remain silent on the matter of the lifetimes of nested operation states and leave them completely up to the implementers of the standard library.

This initially seems like a tempting position: Implementers are free to choose whatever makes the most sense for their users, and whatever gives them the largest gains in terms of storage used, performance sacrificed or gained, et cetera.

From a user’s point-of-view however this is a confluence of the minimal and maximal options. In order to be technically correct (and portable) users must write their code as if the minimal option has been chosen, except when reasoning about when the destructors of objects nested in their operation state will fire, in which case they must assume either minimal or maximal (whichever happens to produce the most conservative results on a case-by-case basis).

Everyone reading this paper likely understands that not all users (or perhaps any) are as perfect as would be required by the preceding paragraph. They will incorrectly reason about lifetimes, lifetime duration of scoped resources, et cetera. Depending on implementation choices these may work. Thereafter we will be in the universe of Hyrum’s Law [10]. Instead of considering lifetimes carefully and deciding to standardize a carefully-chosen option we will live in a universe of *fait accompli* wherein the *de facto* standard is whatever some implementer happened to arrive at when they first implemented `std::execution`.

Acknowledgements

The author would like to thank Ian Petersen and Lewis Baker for their input on this design space and the alternatives therein as well as for feedback on this paper. The author would like to acknowledge Inbal Levi for impelling him to write this paper.

References

[1] Michał Dominiak et al. std::execution P2300R10

[2] J. Wakely. Working Draft, C++ Extensions for Networking N4771
[3] K. Shoop et al. Cancellation is serendipitous-success P1677R2
[4] B. Revzin. Exploring the Design Space for a Pipeline Operator P2672
[5] https://github.com/NVIDIA/stdexec/tree/main
[6] https://github.com/facebookexperimental/libunifex
[7] C. Kohlhoff. A Universal Model for Asynchronous Operations N3747
[8] J. Hoberock et al. A Unified Executors Proposal for C++ P0443R14
[9] C. Kohlhoff. Networking TS changes to improve completion token flexibility and performance P1943R0
[10] https://www.hyrumslaw.com/