Stop the Decay

Document Number: P4288R1
Date: 2026-07-31
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes that `std::execution` algorithms pass through by-reference result datums from child operations rather than uniformly decay-copying them. To aid in the accomplishment thereof it also proposes class templates which abstract the necessary completion storage and completion signature transformation. This paper does not propose changing how `std::execution` algorithms store the arguments passed when creating the sender, or connecting the same.

Background

`std::execution` can be thought of as the realization of the asynchronous analogue of a synchronous function [1][2]. The sender is the function itself (with all arguments selected and applied), the receiver is the channel by which the function communicates with its caller (in the synchronous realm by returning or throwing), and the operation state is the “stack” (i.e. a collection of variables with automatic storage duration, transitively for all children).

For the above-described analogy to be borne out we’d expect to see analogous functionality in `std::execution` for each capability of synchronous functions. Synchronous functions can be called without allocating, and so too can asynchronous operations [3]. Synchronous functions have semantically-discriminated completion modalities (i.e. returning vs. throwing), and so too do asynchronous operations [4]. Et cetera.

Synchronous functions are capable of returning references. This is uncontroversial. Some of the first functions new C++ developers interact with return references (e.g. `std::vector::operator[]`). Therefore it stands to reason that asynchronous functions should have this capability as well.

Verily `std::execution` allows asynchronous operations to send references as values or errors (§33.10 [exec.cmplsig], emphasis added):

“A type `Fn` satisfies `completion-signature` *if and only if it is a function type with one of the following forms:*

- *`set_value_t(Vs...)`, where `Vs` is a pack of object or* ***reference*** *types.*

- *`set_error_t(Err)`, where `Err` is an object or **reference** type.*
- *`set_stopped_t()`"*

However `std::execution` is more than a bare asynchronous function protocol. It also includes asynchronous algorithms meant to generically manipulate asynchronous operations ([5] at §1.1, emphasis added):

*"[`std::execution` is] a Standard C++ model for asynchrony based around three key abstractions: schedulers, senders, and receivers, and **a set of [...] asynchronous algorithms**."*

As adopted into the C++26 working draft in St. Louis `std::execution` had a single asynchronous algorithm which completed with one or more references. It was removed before C++26 shipped [6] (note that the fact `std::execution::split` completed with references meant that it was the asynchronous analogue of a function which returns a reference to a local variable, a detail which was masked by the fact asynchronous operations do not generally eagerly destroy their child operation states [7] (unlike synchronous functions which destroy variables with automatic storage duration immediately upon return therefrom), a defect which is discussed at some length in a later section of this paper).

The `std::execution` algorithms which manipulate child asynchronous operations can be categorized thusly with respect to reference completions (note that the below lists are ordered by the order in which the algorithms are presented in the standard):

- Those which pass through reference completions:
  - `unstoppable`
  - `starts_on`
  - `then`
  - `upon_error`
  - `stopped_as_error`
  - `associate`
- Those which decay-copy all reference completions:
  - `continues_on`
  - `on` (because of transitive use of `continues_on`)
  - `when_all` (although there is an in flight proposal which would have the side effect of changing this for the unary case [8])
  - `when_all_with_variant` (because of transitive use of `when_all` and `into_variant`)
  - `into_variant`
  - `stopped_as_optional`
  - `spawn_future`
  - `as_awaitable`
  - `affine` (unless a sender customizes it) (because of transitive use of `continues_on`)
- Those which decay-copy some reference completions:

    - `let_value` (only value completions)
    - `let_error` (only error completions)
- Those which don't admit non-nullary completions:
    - `spawn` (children are only allowed to have `set_value_t()` and/or `set_stopped_t()` completions)

Note that with two exceptions (`then` and `upon_error`) all of the algorithms in the first and third category pass through those reference completions they do pass through simply because the algorithm's definition does not specify a `set_value` or `set_error` which would receive such completions. That is these algorithms don't go out of their way to support reference completions (other than the exceptions mentioned) it's just that they don't go out of their way to not support them either.

The two exceptions to the above (`then` and `upon_error`) have (along with `upon_stopped`) a special structure: They intercept a completion flowing on a certain channel (the channel suggested by their names), invoke a synchronous invocable with the values or references associated with that completion, and then complete either:

- Successfully with the value or reference returned by said invocation, or
- In error with a `std::exception_ptr` indicating the exception thrown by said invocation

So these algorithms pass through references in two ways:

- From the child operation to the synchronous invocable, and
- From the invocable's return channel to the final receiver

Note that it may not be obvious from the implementation-as-specification (§33.9.12.9 [exec.then]):

```
static constexpr auto complete =
  []<class Tag, class... Args>
    (auto, auto& fn, auto& rcvr, Tag, Args&&... args) noexcept -> void {
      if constexpr (same_as<Tag, decayed-typeof<set-cpo>>) {
        TRY-SET-VALUE(rcvr,
                      invoke(std::move(fn), std::forward<Args>(args)...));
      } else {
        Tag()(std::move(rcvr), std::forward<Args>(args)...);
      }
    };
```

That references returned by the invocable will be passed to the final receiver. This is provided for by blanket wording (§33.9.2 [exec.snd.expos]):

“Then [the completion signatures are represented by] the specialization of `completion_signatures` the set of whose template arguments correspond to the set of completion operations that are potentially evaluated as a result of evaluating `op.start()`.”

This still requires further non-local elaboration as to what it means for a template argument to “correspond” to some set of completion operations (§33.3 [exec.async.ops]):

“An asynchronous operation has a finite set of possible completion signatures corresponding to the completion operations that the asynchronous operation potentially evaluates. For a completion function `set`, receiver `rcvr`, and pack of arguments `args`, let `c` be the completion operation `set(rcvr, args...)`, and let `F` be the function type `decltype(auto(set))(decltype((args))...)`. A completion signature `Sig` is associated with `c` if and only if `MATCHING-SIG`*`(Sig, F)` is `true`.”*

In the instance where the invocable returns some `T&` we see that F becomes `decltype(auto(set))(T&)`. Consulting with the definition of *`MATCHING-SIG`* (§33.1 [exec.general]) we see that this necessitates that `then` and `upon_error` advertise a reference completion in this instance.

Discussion

The Meaning of Completion Signatures

Value categories in C++ are complicated. An expression can be a “prvalue,” an “xvalue,” or an “lvalue.” In spite of this trichotomy the references received when “perfect forwarding” are dichotomous: “rvalue” or “lvalue.”

The above is apropos `std::execution`’s asynchronous operations because the completion of an asynchronous operation is both advertised (via an instantiation of `std::execution::completion_signatures`) and sent (via an invocation of `std::execution::set_value`, `::set_error`, or `::set_stopped`). The former can avail itself of a trichotomous formulation:

- `std::execution::set_value_t(int)`,
- `std::execution::set_value_t(int&&)`, or
- `std::execution::set_value_t(int&)`

Whereas the latter is, at least for generic code, dichotomous as a receiver that advertises:

```
template<typename T>
void set_value(T&&) && noexcept;
```

Will be unable to distinguish prvalues and xvalues.

The above means that while all asynchronous operations send a reference of one form or another (in that a reference will be deduced by a generic/forwarding receiver) they need not advertise a reference. The distinction between a by-value completion and an rvalue-reference completion is not what will be deduced by a generic/forwarding `set_value` member function of the receiver, but rather what the ownership model is for that particular argument to the completion.

Because synchronous functions are first-class features of the language rather than being a library feature (as is the case for the asynchronous functions of `std::execution`) the above divide doesn't exist. A function which advertises by-value return generates a prvalue whereas a function which advertises rvalue-reference return generates an xvalue. The language provides features such as return value optimization to ease or eliminate the cost of chaining together by-value returns.

When the model of `std::execution` is considered as a projection of the synchronous model an interesting gap emerges. Consider (§33.3 [exec.async.ops], emphasis added):

"The lifetime of an asynchronous operation*, also known as the operation's* async lifetime*, begins when its start operation begins executing and ends when* ***its completion operation begins executing****. [...]* ***After an asynchronous operation executes a completion operation****, its associated operation state is invalid. Accessing any part of an invalid operation state is undefined behavior."*

Notice an asymmetry which doesn't exist in the synchronous model: The "async lifetime" of an asynchronous operation begins when the operation begins (i.e. when `std::execution::start` is invoked upon the operation state) and ends at the beginning of the invocation of the completion operation. However despite the end of the "async lifetime" the operation state briefly remains not "invalid" until the completion operation returns. This allows the recipient of a completion signal to continue accessing the contents of the operation state unless they otherwise cause it to be invalidated (e.g. by destroying it). This is not how synchronous functions work. There is no brief moment wherein a function's variables with automatic storage duration remain valid after that function has returned. This allows for operations such as `std::execution::just` to be optimized. Consider the specification thereof (§33.9.11.2 [exec.just]):

```
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)...);
    };
};
```

Note that the “result datums” (§33.3 [exec.async.ops]) are not first moved onto the stack (thereby guarding against reentrant destruction of the operation state) but are rather sent as rvalue references into the operation state. The brief moment of validity discussed above is exploited to eliminate one or more move operations.

The above is accounted for in the design of `std::execution::let_value` et al. Despite the fact said algorithms “end the lifetime of the predecessor operation state [...] [b]efore calling the provided invocable” [7] (internal quotation marks omitted) they do so only “[a]fter persisting the values sent thereby” (ibid.).

The above brings into focus a problematic flexibility/ambiguity in the specification of `std::execution`. Recall that it was previously established (in the background section) that the advertised completion signatures of an asynchronous algorithm are determined via a correspondence between a “completion signature” (which is advertised) and a “completion operation” (which is evaluated). But above we established there’s a cardinality mismatch between these: The former is trichotomous whereas the latter is dichotomous.

Consider that the “completion operation[] [...] evaluate[d]” (§33.3 [exec.async.ops]) by `std::execution::just(5)` has an associated function type `F` which is defined to be `std::execution::set_value_t(int&&)` (ibid.). This function type is used to limit the acceptable completion signatures which may be advertised by `std::execution::just(5)` to those “completion signature[s] `Sig` [for which] *`MATCHING-SIG`*`(Sig, F)` is `true`” (ibid.). We then note that “[f]or function types `F1` and `F2` denoting `R1(Args1...)` and `R2(Args2...)`, respectively, *`MATCHING-SIG`*`(F1, F2)` is `true` if and only if `same_as<R1(Args1&&...), R2(Args2&&...)>` is `true`” (§33.1 [exec.general]). This means that the standard permits `std::execution::just(5)` to advertise either `std::execution::set_value_t(int)` or `std::execution::set_value_t(int&&)`.

This should be changed. Not only is there value in the standard mandating consistency here, but there’s also value in reserving syntactic space for operations which truly send rvalue references, and causing `std::execution::just` to advertise syntactically (i.e. a by-value parameter in the completion signature) what it does semantically (i.e. send a reference to a data member of the operation state which has a restricted period of validity).

Expanding on the last point above consider that while `std::execution::just` sends rvalue references it is performing the moral equivalent of a synchronous by-value return: The operation produces a value stored in its operation state (in the same way a synchronous function might return a value by copying or moving a variable with automatic storage duration) as an rvalue reference so the consumer can efficiently inspect and forward it, but the consumer is not allowed to rely on the validity thereof past the end of the operation state’s lifetime (in the same way a dangling reference is invalidated with the destruction of variables with automatic storage duration), nor is it allowed to rely on the validity thereof once the completion operation ends and the operation state becomes “invalid.”

Put differently: The exact modalities may differ, but a synchronous function which returns by value (generating a prvalue) and an asynchronous function which completes by value (generating an xvalue which may or may not refer to an object in the operation state) function similarly, producing a transient object which must be persisted elsewhere if the recipient thereof wishes for said object to remain valid long term. Therefore, since the modalities are similar, they should be advertised similarly (i.e. consistently without rvalue reference qualification).

Now let's consider the previously-standard [6] algorithm which advertised and sent lvalue references: `std::execution::split` [5]. Consider the specification of its completion (id.):

```
visit(
  [this](const auto& tupl) noexcept -> void {
    apply(
      [this](auto tag, const auto&... args) noexcept -> void {
        tag(std::move(*rcvr), args...);
      },
      tupl);
  },
  sh_state->result);
```

Note that `args` is a pack of lvalues therefore the blanket wording of `std::execution` mandates that this algorithm advertise by-lvalue-reference completions (this blanket wording was previously explained above). Note also that `sh_state` is a pointer to a reference-counted state shared by:

- The sender, and
- All operation states formed by connecting it

Consider what happens if a consumer of `std::execution::split` infers from the by-reference completion signature that, unlike in the case of by-value completion signatures, destroying the operation state of `std::execution::split` (or otherwise allowing it to become "invalid") does not affect the validity of the references it is sent. In many cases nothing will happen, the references will remain valid because another operation state will be not "invalid," or the sender still exists somewhere. But in unfortunate circumstances this will cause undefined behavior as the lifetime of the referred-to object(s) ends.

The above might seem like a counter argument to not decay-copying (i.e. the purpose of this paper). But think about the shape of `std::execution::split` projected into the synchronous domain:

```
const int& split(const std::shared_ptr<const shared-state> sh_state) {
  // Compute the result if not present in *sh_state
  return sh_state->result;
}
```

Facially bizarre. There are situations wherein this synchronous analogue can be used safely, but the interface doesn't capture them as well as, for example:

```
const int& split(const std::shared_ptr<const shared-state>& sh_state);
```

There are of course situations wherein this interface can create dangling references, but that is clearly the fault of the caller, unlike in the case of the previous interface.

From the previous paragraph flows the conclusion: By-reference return from synchronous functions, and therefore by analogy by-reference completion of asynchronous operations, indicates that the referred-to object has a non-local locus of ownership. Therefore in the generic case, where we don't control the inputs to the asynchronous operation we shouldn't inhibit the transmission of references by arbitrarily inserting decay-copies.

Put in the form of a single concrete example/analogy: `std::apply` returns `decltype(auto)`, not `auto`.

`set_value_t(T)` vs. `set_value(T)`

The previous section assumes that the `set_value` and `set_error` (note that this discussion doesn't apply to `set_stopped` because it must be nullary) accept their parameters by forwarding reference, not by value, even in the case where the operation parameterized by said receiver advertises a by-value completion.

In certain, specific cases there's nothing wrong with `set_value` or `set_error` accepting one or more of their parameters by value, but this is not true in general. Receivers which wish to be truly generic vis-à-vis the operation they're parameterizing must accept all parameters by reference (whether forwarding or otherwise). The reason for this is because:

- Objects, in general, can have throwing move [9] and copy constructors,
- A move or copy may be required to populate a by-value parameter of `set_value` or `set_error`, and
- `std::execution::set_value` (§33.7.2 [exec.set.value]) and `::set_error` (§33.7.2 [exec.set.error]) require the operations they perform, including population of parameters, not to throw exceptions via *`MANDATE-NOTHROW`* (§33.1 [exec.general])

Therefore while a receiver which happens to know that the type of a parameter is nothrow move or copy constructible (depending on the situation under which it's received) can accept that parameter by value, a receiver which wishes to accept parameters generally cannot do so [10].

Reference Validity and Structured Concurrency

Using reference (rather than value) semantics in asynchronous contexts can trigger reflexive terror and recoil among practitioners of "unsafe" languages (such as C++). The issue being that

the initiating function of the asynchronous operation (in `std::execution` this refers to `std::execution::start`) returns synchronous control to its synchronous caller while asynchronous control simultaneously proceeds in the "background." The "foreground" thread of synchronous control might, if casually or uncarefully coded, invalidate references upon which the asynchronous operation relied.

The doctrinaire solution to this is to use value semantics wherever possible, and where reference semantics are actually required to, instead of using raw references and pointers, use owning, value-like wrappers (e.g. `std::unique_ptr` and `std::shared_ptr`). While this approach might assuage reflexive horror it has a problem: Overhead in the form of dynamic allocations.

While there are domains which can simply eat the cost of dynamic allocations and thereby avoid, in some sense, reference semantics this does not describe every domain. Moreover baking such a requirement into C++'s standardized approach to asynchrony would violate C++'s core value proposition: Zero-cost abstractions.

Instead let's consider why reference semantics are so differentially terrifying in the asynchronous domain. First we begin by asking ourselves why this synchronous code is not terrifying:

```
int a = 5;
int b = 6;
int& c = std::max(a, b);
```

A simplistic answer is that the lifetimes of `a` and `b` persist beyond the return from `std::max` and the binding of `c`. But this is a statement of what is, not why it is. The reason is that C++ is composed of iteratively-nested structures. Consider in the above that:

- The entire snippet is nested inside some function
- The lifetime of `a` is nested inside said function
- The lifetime of `b` is nested inside the lifetime of `a`
- The invocation of `std::max` is nested inside the lifetime of `b`

It flows from this nesting that when `c` is bound both `a` and `b` will be within their lifetimes, and therefore that the use of a reference is not problematic. If however we wrote:

```
int& foo() {
  int a = 5;
  int b = 6;
  int& c = std::max(a, b);
  return c;
}
```

We would (correctly) identify this code as problematic. This is because the code belies its nesting: The lifetimes of `a` and `b` are nested within the invocation of `foo`, and therefore returning `c` generates a dangling reference by causing a reference to either `a` or `b` to escape the scope in which the lifetimes of said objects are nested.

Notice the use of the word “scope” in the preceding paragraph. When discussing C++ we often consider a scope as a lexical construct, for example `foo`’s function body is a scope. Scopes do not only exist lexically however, and by examining their nesting in a platonic rather than lexical sense we can discuss “structured programming” and, as a special case thereof, “structured concurrency,” i.e. the application of structured programming to problems of concurrency.

Consider then the following code:

```
std::execution::just(5) |
std::execution::let_value([](int& i) {
  return
    std::execution::schedule(std::execution::get_system_scheduler()) |
    std::execution::then([&i]() { return i; });
});
```

Let’s walk through what would happen if the sender resulting from the above expression were connected and the resulting operation state were started:

1. `std::execution::just` completes with `std::execution::set_value(..., 5)`
2. `std::execution::let_value` persists 5 into its operation state
3. `std::execution::let_value` invokes the outer lambda providing a reference to the integer persisted into its operation state
4. The outer lambda creates and returns a sender
5. `std::execution::let_value` connects and starts the sender returned by the outer lambda
6. The system execution context completes the schedule operation with `std::execution::set_value` with no result datums
7. `std::execution::then` nullary invokes the inner lambda
8. The inner lambda copies the integer stored in `std::execution::let_value`’s operation state (since it captured `i`, which is a reference to said integer, by reference) to populate its return value (which it then returns)
9. The overall operation completes with `std::execution::set_value(..., 5)`

Step 8 is exactly the sort of terrifying formulation being discussed: When `std::execution::just` completes and causes `std::execution::let_value` to store the value sent thereby and then invoke the outer lambda the inner lambda is constructed and captures a reference to the persisted value. There is then an intervening asynchronous `std::execution::schedule` operation wherein execution of the asynchronous operation waits for the system execution context to schedule follow on work. This is exactly the sort of formulation which seems like it might be unsafe: `std::execution::set_value` calls

`std::execution::start` on the operation state obtained by connecting and starting the sender obtained by invoking the outer lambda, and then synchronous and asynchronous execution diverge with asynchronous execution running in the “background” with the expectation that the reference it depends on will remain valid.

The example, however, is correct. Consider that:

- `std::execution::schedule`’s operation state is nested within `std::execution::then`’s operation state,
- `std::execution::then`’s operation state (which refers to the integer) is nested within `std::execution::let_value`’s operation state, and
- The inner lambda is nested within `std::execution::then`’s operation state

This ensures structurally that the lifetime of the inner lambda, which refers to the integer, is nested within the lifetime of the integer to which it refers.

The above is not the entire story, however. When `std::execution::let_value` calls `std::execution::start` on `std::execution::then`’s operation state `std::execution::then` calls `std::execution::start` on `std::execution::schedule`’s operation state. This causes the system execution context to refer to `std::execution::schedule`’s operation state until some future point whereat the `std::execution::schedule` operation completes on an execution agent owned by the system execution context. Perhaps then it is this reference to `std::execution::schedule`’s operation state which is fraught. This brings into focus one of the most fundamental requirements of `std::execution` (§33.3 [exec.async.ops]):

“If the lifetime of an asynchronous operation's associated operation state ends before the lifetime of the asynchronous operation, the behavior is undefined.”

At first this seems like a restatement of the problem with which this section opened: Careful care and attention is required to ensure references passed to asynchronous operations remain appropriately valid, we’ve just rederived this in the context of a reference to `std::execution::schedule`’s operation state. But consider that:

- We’re no longer talking about `&i` (i.e. the by-reference capture), we’re instead talking about a reference to an operation state, and
- The operation state of `std::execution::schedule` is a subobject of the operation state of `std::execution::then`

The second bullet means that the above-quoted requirement flows outwards. `std::execution::then` doesn’t complete until after its predecessor completes and therefore if the owner of its operation state satisfies the above-quoted requirement it trivially, transitively does so as well. This flow continues outwards until the boundary of the structured concurrency

of `std::execution` is reached whereat it becomes incumbent upon someone to non-trivially satisfy the above-quoted requirement.

What happens at the above-mentioned boundary is the projection of non-lexical scopes. The projection of asynchronous scopes. The bootstrapping of structured programming in the asynchronous domain.

But consider the first bullet above. The asynchronous scoping required to safely use references is present whether we use references or not. Each reference doesn't add complexity or requirements to the model because the model projects the lexical scopes we're used to in synchronous programming into the asynchronous domain and therefore our synchronous intuition works in the asynchronous domain.

Given the above it seems needless to pessimize asynchronous operations with by-reference completions (via friction when attempting to combine them with standard asynchronous algorithms). We shouldn't project the trauma wrought by unstructured concurrency onto its structured alternative once said alternative has been found and standardized.

The Ronseal [11] Argument

One of the priorities of `std::execution` is to "[e]ncapsulate common asynchronous patterns in customizable and reusable algorithms, so users don't have to invent things themselves" [5]. This implies that algorithms should not arrogate to themselves responsibilities which are not logically theirs because doing so makes them less generally useful. An algorithm with one responsibility may be used wherever that responsibility must be discharged. If, however, that algorithm accrues another responsibility it can now only be used where both responsibilities must be discharged, narrowing the set of situations in which it's useful. This narrowed set of situations can lead to a user with the narrow set of requirements being dissatisfied and having to "invent [some]thing[] themselves."

The name of `std::execution::when_all`, for example, suggests a single purpose: To "complete[] once all of [its] input senders have completed" [5]. Nothing about the quoted description of purpose has anything to do with decay-copying, and therefore `std::execution::when_all` shouldn't decay-copy anymore than it should randomly interact with stop requests [12]. Decay-copying, like interacting with stop requests [13], can be encapsulated in a distinct algorithm [14] which addresses that single purpose. That algorithm can then be combined with `std::execution::when_all` when the two distinct purposes happen to both be required.

Note that this section is effectively an argument for the Single Responsibility Principle [15].

Suspension and Resumption

The discussion of reference completions for asynchronous algorithms is not just about which completions to advertise and send, it's also about which to store.

Just as synchronous functions often block as the ability to make forward progress is delegated to some other task, so too do asynchronous operations often "suspend" waiting for some other asynchronous operation to complete whereupon they "resume." Upon "suspension" the local, synchronous stack frame is exited, and therefore it's necessary to store in flight values (e.g. the completion of some child asynchronous operation) until "resumption." Ergo the importance of storing the completions of some other operation (whether it sends references or otherwise).

Currently `std::execution`'s algorithms uniformly perform this by persisting the completion using *`decayed-tuple`* (§33.4 [execution.syn]):

```
template<class... Ts>
  using decayed-tuple = tuple<decay_t<Ts>...>;  // exposition only
```

Ergo the title of this paper. Beyond describing why we should not use this approach (see above) we ought to formulate an alternative. Given how pervasive the pattern of persisting another operation's completion signatures for use later is, whatever formulation we arrive at should arguably be available for consumption by users, rather than just left as an implementation detail of the library.

Note that *`decayed-tuple`* on its own is not enough, in general, to persist the completion of an operation. `std::execution`'s asynchronous operations can complete in a variety of ways and therefore, in general, to store the completion of another operation one needs, assuming one is using the current idiom of pervasive decay-copying, a `std::variant<`*`decayed-tuple`*`<...>>`.

One of the fundamental basis operations of a utility which aims to replace `std::variant<`*`decayed-tuple`*`<...>>` is the ability to "arrive:" I.e. provide the completion tag and result datums and "suspend" (i.e. store) the completion for later "resumption" (i.e. retrieval). From this let's start designing an interface:

```
namespace std::execution {
  template<valid-completion-signatures Signatures>
    struct storage_for_completion_signatures {
      using storage-type = variant<monostate, ...>;
      storage-type storage;

      template<class CompletionTag, class... ResultDatums>
        constexpr void arrive(
          CompletionTag tag,
          ResultDatums&& result_datums)
```

```
            noexcept(...);
    };
}
```

But consider the shape of the `arrive` member function vis-à-vis what was discussed earlier with respect to the divide between advertised completion signature and received/deduced type: When one of the elements of `ResultDatums` deduces to an rvalue reference, does this correspond to a by-value completion parameter or a by-reference completion parameter?

The above suggests that we need a separate source of truth, a way to derive the completion signature correspondence from the completion signatures (i.e. `Signatures`) rather than from the parameters to `arrive` (i.e. `ResultDatums`). Fortunately the standard gives us the building blocks with which to do this: *`MATCHING-SIGS`* (§33.1 [exec.general]):

"For function types `F1` and `F2` denoting `R1(Args1...)` and `R2(Args2...)`, respectively, `MATCHING-SIG`*`(F1, F2)` is `true` if and only if `same_as<R1(Args1&&...), R2(Args2&&...)>` is `true`."*

Note the addition of rvalue reference qualification when performing the comparison, thereby reifying and addressing the ambiguity discussed above. But notice something else: This means that completion signatures that are identical save rvalue reference qualification cannot be distinguished, that is to say that arrival of a by value completion is indistinguishable from a by reference completion when the reference is an rvalue reference.

In order to mediate the above we propose a utility for matching the parameters to `arrive` to a completion signature:

```
template<class Sigs, class Tag, class... Args>
  using matching_completion_signature_t = ...;
```

Given the ambiguity discussed above, and given the fact the template may be evaluated with arbitrary arguments (i.e. not necessarily arguments which correspond to one or more completion signatures) we also propose a utility for determining if a match exists:

```
template<class Sigs, class Tag, class... Args>
  inline constexpr bool has_matching_completion_signature_v = ...;
```

We now have the tools to:

- Constrain `arrive`, and
- Determine which variant alternative, if any, should be populated thereby

There is a final issue with this interface: The `noexcept` clause. If the matching completion signature accepts all parameters by reference there's no issue, since everything can be persisted with no possibility of throwing an exception. However even if the matching completion signature accepts some or all parameters by value there still may be no problem: They may all

be able to be persisted without throwing. Unfortunately not all types fall under the preceding penumbrae. It might be tempting to say that when persistence can throw the `noexcept` clause should simply be equivalent to `noexcept(false)`, however this would be missing an opportunity to concentrate common complexity and thereby abstract it away. There are two sorts of operations vis-à-vis persistence failures, those which wish to:

- Persist an error completion whose associated result datum is `std::current_exception()` (i.e. `std::execution::continues_on`), and
- Handle the exception in some other way (i.e. `std::execution::when_all` which requests stop and marks the overall operation as failed rather than simply the particular child whose result datums could not be persisted)

The former scenario can be handled easily with the current form: `arrive` is always `noexcept(true)`, and the storage can persist the moral equivalent of `std::execution::set_error_t(std::exception_ptr)` if needed (i.e. if persisting any completion signature can throw an exception). Note however another scenario emerging: The conditional need to enrich the advertised set of completion signatures based on whether some part of the asynchronous operation (in our case persistence) can throw an exception. We can abstract this away by further enriching `storage_for_completion_signatures` so that it not only persists the given set of completion signatures, but also provides the consumer with access to a possibly-enriched set of completion signatures featuring `std::execution::set_error_t(std::exception_ptr)` if needed:

```
namespace std::execution {
  template<valid-completion-signatures Signatures>
    struct storage_for_completion_signatures {
      using storage-type = variant<monostate, ...>;
      storage-type storage;

      using completion_signatures = ...;
      static consteval completion_signatures get_completion_signatures();

      template<class CompletionTag, class... ResultDatums>
        constexpr void arrive(
          CompletionTag tag,
          ResultDatums&& result_datums)
            noexcept;
    };
}
```

This however doesn't address the second use case described above. That use case requires us to be able to configure `storage_for_completion_signatures` to not exhibit the behavior

above (i.e. catching and persisting exceptions, and “enriching” the set of completion signatures). We can redesign our class therefore to support selection of this policy via a template argument:

```
namespace std::execution {
  enum class storage_for_completion_signatures_error_policy {
    internalize,
    propagate
  };

  template<valid-completion-signatures Sigs,
    storage_for_completion_signatures_error_policy ErrorPolicy =
      storage_for_completion_signatures_error_policy::internalize>
    struct storage_for_completion_signatures;
}
```

Now that the details of how a completion should be stored (i.e. “suspended”) have been established we focus on the details of how completions should be accessed (i.e. “resumed”). The most straightforward way is to send them through to a receiver:

```
namespace std::execution {
  template<valid-completion-signatures Sigs,
    storage_for_completion_signatures_error_policy ErrorPolicy>
    struct storage_for_completion_signatures {
      // ...
      template<receiver Receiver>
        constexpr bool complete(Receiver&& rcvr) && noexcept;
    };
}
```

`bool` is returned to indicate whether there was a stored completion (`true`) or not (`false`). Note that the `rcvr` parameter is unused unless `true` is returned in the same way as the trailing, forwarded arguments to, for example, `std::unordered_map::try_emplace` are conditionally consumed. The fact that information as to whether or not a completion is stored suggests that consumers should be able to inspect this information directly:

```
namespace std::execution {
  template<valid-completion-signatures Sigs,
    storage_for_completion_signatures_error_policy ErrorPolicy>
    struct storage_for_completion_signatures {
      // ...
      constexpr bool has_completion() const noexcept;
    };
}
```

While `complete` is convenient as an API it admits only one modality by which the stored completion (if any) can be consumed: Completing an asynchronous operation by sending a completion signal to a receiver. It provides no support for simply inspecting the stored completion, nor does it provide support for combining completions. The latter is significant for a redesigned `std::execution::when_all` which makes use of `storage_for_completion_signatures`. Such a redesigned algorithm would require a `storage_for_completion_signatures` instance:

- For each child operation, and
- For the error completion (if any)

Once each child completes the algorithm:

1. Checks the error completion storage and sends that completion if any, otherwise
2. Concatenates the completions stored in the per-child-operation completion storage and sends that combined completion

This suggests an interface similar to `std::visit` over `std::variant`. This leaves two questions:

- How should each completion be represented?
- What happens if any visited `storage_for_completion_signatures` does not have a stored completion?

`std::tuple<Tag, Args...>` initially seems appealing to answer the first bullet, implementing and working with this however presents usability challenges as boilerplate is required to separate the tag from the arguments. Representing a stored completion directly in the type system becomes an obvious, superior approach:

```
namespace std::execution {
  template<class Signature>
    struct storage_for_completion_signature;
  template<class Tag, class... Args>
    struct storage_for_completion_signature<Tag(Args...)> {
      using tag_type = Tag;
      using signature_type = Tag(Args...);

      static constexpr Tag tag() noexcept;

      template<class Self>
        constexpr decltype(auto) arguments(this Self& self) noexcept;

      template<class Self>
        constexpr decltype(auto) forward_arguments(this Self&& self)
          noexcept;
```

```
      template<receiver Receiver>
        constexpr void complete(Receiver&& rcvr) && noexcept;
    };
}
```

Where arguments returns a reference to a `std::tuple`, and `forward_arguments` returns a `std::tuple` of references. Having the latter becomes essential when implementing `std::execution::when_all` as it allows the completions of all children to be combined using `std::tuple_cat(std::move(completions).forward_arguments()...)`.

This brings us to the question of visitation of several `storage_for_completion_signatures` instances where one or more of them does not store a completion. At first mirroring the interface of `std::visit` seems appropriate with no completion represented by `std::monostate`. However this falls apart as we once again consider application of this primitive to `std::execution::when_all`. Consider `std::execution::when_all` applied to 10 child senders. Because "`when_all` only accepts senders with a single value completion signature" (§33.9.12.12 [exec.when.all]) such a sender will have exactly one value completion. But the `storage_for_completion_signatures` instances used to persist the completion of each child will have two states, empty or not empty. This means that a `std::visit`-style API will need to generate 2^{10} (i.e. 1 024) branches which differentially invoke the visitor.

Which brings us to the final design for `visit_stored_completion`: Invokes the provided visitor with the completion stored in each provided `storage_for_completion_signatures` if each has a completion, otherwise nullary invokes the visitor.

Note that as has been the case elsewhere the requirement to "suspend" a completion and later "resume" it is not unique to the asynchronous domain. Careful analysis finds that most of the problems addressed by `std::execution` are projections of problems which exist in the synchronous domain. This analysis is non-trivial because those problems in the synchronous domain are handled at the language level (for example construction and destruction [16]), the assembly level (function call and return), or which are greatly simplified by the fact that synchronous functions make forward progress at the expense of their caller (meaning synchronous functions only benefit from serial compositional modalities, i.e. the equivalent of `std::execution::when_all` is unavailable to them).

As discussed above "suspension" of a completion allows some intervening asynchronous operation to make forward progress. At the completion thereof "resumption" allows the "suspended" operation to complete as if it were not "suspended" (note that decay-copying does not fulfill this requirement since it materially changes the completion and therefore it is not "as if it were not 'suspended'"). This pattern is seen in the synchronous domain when destructors are run on exit from a function. The fact that the return must be "suspended" so that one or more intervening synchronous operations (i.e. one or more destructors) may run:

- Provides an analogue (albeit in the language rather than the library as has been observed about `std::execution` as a whole [1]) to the functionality designed and discussed in this section, and
- Bolsters the argument made by this paper as a whole, since running destructors while returning from a function does not alter or interfere with the return modality (i.e. it does not add a decay-copy)

Proposal

[execution.syn]

[...]

```
// [exec.getcomplsigs], get completion signatures
template<class Sndr, class... Env>
  consteval auto get_completion_signatures() ->
    valid-completion-signatures auto;

template<class Sndr, class... Env>
    requires sender_in<Sndr, Env...>
  using completion_signatures_of_t = decltype(
    get_completion_signatures<Sndr, Env...>());

// [exec.matchingcomplsigs], completion signature matching
template<class Sigs, class Tag, class... Args>
  using matching_completion_signature_t = see below;
template<class Sigs, class Tag, class... Args>
  struct matching_completion_signature : std::type_identity<
    matching_completion_signature_t<Sigs, Tag, Args...>> {};

template<class Sigs, class Tag, class... Args>
  inline constexpr bool has_matching_completion_signature_v = see below;
template<class Sigs, class Tag, class... Args>
  struct has_matching_completion_signature : std::bool_constant<
    has_matching_completion_signature_v<Sigs, Tag, Args...>> {};

// [exec.storageforcomplsigs], completion signature storage
template<class Signature>
  struct storage_for_completion_signature;
template<class Tag, class... Args>
  struct storage_for_completion_signature<Tag(Args...)>;
```

```
enum class storage_for_completion_signatures_error_policy {
  internalize,
  propagate
};

template<valid-completion-signatures Sigs,
  storage_for_completion_signatures_error_policy ErrorPolicy =
    storage_for_completion_signatures_error_policy::internalize>
  struct storage_for_completion_signatures;

// [exec.storageforcomplsigs.visit], completion signature storage visitation
template<class Visitor, class... Storages>
  constexpr see below visit_stored_completion(
    Visitor&& visitor, Storages&&... storages) noexcept(see below);
template<class R, class Visitor, class... Storages>
  constexpr R visit_stored_completion(
    Visitor&& visitor, Storages&&... storages) noexcept(see below);

// [exec.connect], the connect sender algorithm
struct connect_t;
inline constexpr connect_t connect{};

template<class Sndr, class Rcvr>
  using connect_result_t =
    decltype(connect(declval<Sndr>(), declval<Rcvr>()));
```

[...]

[exec.matchingcomplsigs]

Note: This is a new section.

For a type `Sigs`, a completion tag type `Tag`, and a pack of argument types `Args`, if `Sigs` is not a specialization of `completion_signatures`, then `matching_completion_signature_t<Sigs, Tag, Args...>` does not denote a type.

Otherwise, let M be the set of template arguments `Sig` of `Sigs` for which *`MATCHING-SIG`*`(Sig, Tag(Args...))` is true.

If M contains exactly one type, `matching_completion_signature_t<Sigs, Tag, Args...>` denotes that type, otherwise `matching_completion_signature_t<Sigs, Tag, Args...>` does not denote a type.

`has_matching_completion_signature_v<Sigs, Tag, Args...>` is `true` if `matching_completion_signature_t<Sigs, Tag, Args...>` denotes a type, `false` otherwise.

[exec.storageforcomplsigs]

Note: This is a new section.

```
namespace std::execution {
  template<class Tag, class... Args>
    struct storage_for_completion_signature<Tag(Args...)> {
      using tag_type = Tag;
      using signature_type = Tag(Args...);
      using normalized-signature-type = Tag(Args&&...);   // exposition only

      std::tuple<Args...> storage;                        // exposition only

      template<class... Ts>
        requires sizeof...(Ts) == sizeof...(Args) &&
                 constructible_from<tuple<Args...>, Ts...>
        constexpr explicit storage_for_completion_signature(
          Tag, Ts&&... args)
            noexcept(is_nothrow_constructible_v<tuple<Args...>, Ts...>)
          : storage(std::forward<Ts>(args)...) {}

      static constexpr Tag tag() noexcept {
        return {};
      }

      template<class Self>
        constexpr decltype(auto) arguments(this Self& self) noexcept {
          return std::forward<Self>(self).storage;
        }

      template<class Self>
        constexpr decltype(auto) forward_arguments(this Self&& self) noexcept
        {
          return apply(
            [](auto&&... args) noexcept {
              return forward_as_tuple(std::forward<decltype(args)>(args)...);
            },
            std::forward<Self>(self).arguments());
        }
```

```
    template<receiver Receiver>
      constexpr void complete(Receiver&& rcvr) && noexcept {
        apply(
          tag(),
          forward_as_tuple(std::forward<Receiver>(rcvr)),
          std::move(*this).forward_arguments());
      }
  };

  template<valid-completion-signatures Sigs,
    storage_for_completion_signatures_error_policy ErrorPolicy>
  struct storage_for_completion_signatures {
    template<class T>
      static constexpr bool nothrow-storable =
        is_reference_v<T> || is_nothrow_move_constructible_v<T>;
                                                    // exposition only

    static constexpr bool internalizing =
      ErrorPolicy ==
        storage_for_completion_signatures_error_policy::internalize;
                                                    // exposition only

    template<class T>
      static constexpr bool nothrow-arrive = internalizing ||
        nothrow-storable<T>;                        // exposition only

    static constexpr bool nothrow_arrive = see below;

    using input-completion-signatures = see below;  // exposition only
    using completion_signatures = see below;

    static consteval completion_signatures get_completion_signatures();

    using storage-type = see below;                 // exposition only
    storage-type storage;                           // exposition only

    template<class Tag, class... Args>
      constexpr void arrive(Tag tag, Args&&... args) noexcept(see below);

    constexpr bool has_completion() const noexcept {
      return holds_alternative<monostate>(storage);
    }
```

```
    template<class Self, class Visitor>
      constexpr decltype(auto) visit(this Self&& self, Visitor&& visitor)
        noexcept(
          noexcept(
            visit_stored_completion(
              std::forward<Visitor>(visitor),
              std::forward<Self>(self).storage)))
      {
        return visit_stored_completion(
          std::forward<Visitor>(visitor),
          std::forward<Self>(self).storage);
      }
    template<class R, class Self, class Visitor>
      constexpr decltype(auto) visit(this Self&& self, Visitor&& visitor)
        noexcept(
          noexcept(
            visit_stored_completion<R>(
              std::forward<Visitor>(visitor),
              std::forward<Self>(self).storage)))
      {
        return visit_stored_completion<R>(
          std::forward<Visitor>(visitor),
          std::forward<Self>(self).storage));
      }

    template<std::execution::receiver Receiver>
      constexpr bool complete(Receiver&& rcvr) && noexcept {
        return visit([&rcvr](auto&& completion) noexcept {
          if constexpr (is_same_v<
            remove_cvref_t<decltype(completion)>,
            monostate>)
          {
            return false;
          } else {
            std::forward<decltype(completion)>(completion).complete(
              std::forward<Receiver>(rcvr));
            return true;
          }
        });
      }
```

```
  };
}
```

Let `Ts` be the pack formed by concatenating the function argument types of each template argument of `Sigs`. The value of the `nothrow_arrive` static data member of `storage_for_completion_signatures` template specializations is (*`nothrow-arrive`*`<Ts> && ...`).

Let `Sigs2` be the pack of template arguments of `Sigs` except with duplicate types removed, and let `Ts` be the pack formed by concatenating the function argument types of the types in `Sigs2`. The nested exposition-only *`input-completion-signatures type`* of of `storage_for_completion_signatures` template specializations denotes `execution::completion_signatures<Sigs2...>`. The nested `completion_signatures` type of `storage_for_completion_signatures` template specializations denotes *`input-completion-signatures`* if `set_error_t(exception_ptr)` is one of the types in `Sigs2` or (*`nothrow-storable`*`<Ts> && ...`) is `true`, otherwise it denotes `execution::completion_signatures<Sigs2..., set_error_t(exception_ptr)>`.

Let `Sigs2` be the pack of template arguments of `completion_signatures`. The exposition-only nested *`storage-type`* type of `storage_for_completion_signatures` template specializations denotes `variant<monostate, storage_for_completion_signature<Sigs2>...>`.

```
static consteval completion_signatures get_completion_signatures();
```

Effects: Let `Sigs2` be the pack of template arguments of *`input-completion-signatures`*. Let `Sigs3` denote the pack `storage_for_completion_signature<Sigs2>::`*`normalized-signature-type`*`...` except with duplicate types removed. Let `Ts` be the pack formed by concatenating the function argument types of function types in `Sigs2`. Equivalent to:

```
if (
  sizeof...(Sigs2) == sizeof...(Sigs3) &&
  ((is_reference_v<Ts> || is_move_constructible_v<Ts>) && ...)) {
  return completion_signatures{};
} else {
  throw unspecified-exception();
}
```

```
template<class Tag, class... Args>
  constexpr void arrive(Tag tag, Args&&... args) noexcept(see below);
```

Constraints: `has_matching_completion_signature_v<`*`input-completion-signatures`*`, Tag, Args...>` is `true`.

Hardened preconditions: `has_completion()` is `false`.

Effects: Equivalent to:

```
try {
  storage.template emplace<
    storage_for_completion_signature<
      matching_completion_signature_t<
        input-completion-signatures, Tag, Args...>>>(
          std::forward<Args>(args)...);
} catch (...) {
  if constexpr (internalizing) {
    storage.template emplace<
      storage_for_completion_signature<set_error_t(exception_ptr)>>(
        current_exception());
  } else {
    storage.template emplace<monostate>();
    throw;
  }
}
```

Remarks: Let `Ts` be the pack formed by concatenating the function argument types of each template argument of the type denoted by `matching_completion_signature_t<input-completion-signatures, Tag, Args...>`. The expression in the noexcept clause is equivalent to `(nothrow-arrive<Ts> && ...)`.

[exec.storageforcomplsigs.visit]

Note: This is a new section.

```
template<class Visitor, class... Storages>
  constexpr see below visit_stored_completion(
    Visitor&& visitor, Storages&&... storages) noexcept(see below);
template<class R, class Visitor, class... Storages>
  constexpr R visit_stored_completion(
    Visitor&& visitor, Storages&&... storages) noexcept(see below);
```

Let *as-storage* denote the following exposition-only function templates:

```
template<class Signatures, auto ErrorPolicy>
  constexpr auto&& as-storage(
    storage_for_completion_signatures<Signatures, ErrorPolicy>& s)
  {
    return s;
  }
template<class Signatures, auto ErrorPolicy>
```

```
  constexpr auto&& as-storage(
    const storage_for_completion_signatures<Signatures, ErrorPolicy>& s)
  {
    return s;
  }
template<class Signatures, auto ErrorPolicy>
  constexpr auto&& as-storage(
    storage_for_completion_signatures<Signatures, ErrorPolicy>&& s)
  {
    return std::move(s);
  }
template<class Signatures, auto ErrorPolicy>
  constexpr auto&& as-storage(
    const storage_for_completion_signatures<Signatures, ErrorPolicy>&& s)
  {
    return std::move(s);
  }
```

Let *n* be `sizeof...(Storages)`. For each $0 \le i < n$, let $\texttt{StorageBase}_i$ denote the type `decltype(as-storage(std::forward<Storages`$_i$`>(storages`$_i$`)))`.

Constraints: $\texttt{StorageBase}_i$ is a valid type for all $0 \le i < n$.

Let `StorageBases` denote the pack of types $\texttt{StorageBase}_i$.

Let *m* be a pack of *n* values of type `size_t`. Such a pack is valid if $0 \le m_i <$ `variant_size_v<remove_reference_t<StorageBase`$_i$`>::storage-type> - 1` for all $0 \le i <$ *n*. For each valid pack *m*, let *e*(*m*) denote the expression:

```
INVOKE(std::forward<Visitor>(visitor),
       std::get<m + 1>(std::forward<StorageBases>(storages).storage)...)
```

for the first form and

```
INVOKE<R>(std::forward<Visitor>(visitor),
          std::get<m + 1>(std::forward<StorageBases>(storages).storage)...)
```

for the second form.

Let *f* denote the expression:

```
INVOKE(std::forward<Visitor>(visitor))
```

for the first form and

```
INVOKE<R>(std::forward<Visitor>(visitor))
```

for the second form.

Mandates: For each valid pack *m*, *e*(*m*) is a valid expression. *f* is a valid expression. All such expressions *e*(*m*) and expression *f* are of the same type and value category.

Returns: *f* if *as-storage*(storages$_i$).holds_completion() is false for any $0 \le i < n$. Otherwise *e*(*m*), where *m* is the pack for which m_i is *as-storage*(storages$_i$).*storage*.index() - 1 for all $0 \le i < n$. The return type is decltype(*f*) for the first form.

Remarks: The expression in the noexcept clause is equivalent to true if:

- For each valid pack *m*, noexcept(*e*(*m*)) is true, and
- noexcept(*f*) is true

false otherwise.

Complexity: For $n \le 1$, the invocation of the callable object is implemented in constant time, i.e., for $n = 1$, it does not depend on the number of alternative types of V_0. For $n > 1$, the invocation of the callable object has no complexity requirements.

[exec.snd.expos]

[...]

```
template<class Tag, class Data, class... Child>
  template<decays-to<basic-sender> Sndr, class... Env>
    constexpr auto basic-sender<Tag, Data, Child...>::
      get_completion_signatures();
```

Let Rcvr be the type of a receiver whose environment has type E, where E is the first type in the list Env..., env<>. Let *CHECK-TYPES*() be the expression *impls-for*<Tag>::template *check-types*<Sndr, E>(), and let CS be a type determined as follows:

- If *CHECK-TYPES*() is a core constant expression, let op be an lvalue subexpression whose type is connect_result_t<Sndr, Rcvr>. Then CS is the specialization of completion_signatures the set of whose template arguments ~~correspond to the set of completion operations that are~~ are determined as follows unless otherwise specified. For each completion operation that is potentially evaluated ([basic.def.odr]) as a result of evaluating op.start()~~.~~:
  - If the completion operation is evaluated as a result of an invocation of complete on a specialization of storage_for_completion_signature, the corresponding template argument is the signature_type member of that specialization,
  - Otherwise, let Sig be a completion signature associated with the completion operation ([exec.async.ops]), if more than one such completion signature exists,

an arbitrary such completion signature is selected, if `Sig` is of the form `Tag(Args...)`, the corresponding template argument is `Tag(`*`DECAY-RVALUE-REFERENCES`*`(Args)...)`, for type `T`, *`DECAY-RVALUE-REFERENCES`*`(T)` denotes `decay_t<T>` if `T` is an rvalue reference type, `T` otherwise

- Otherwise, `CS` is `completion_signatures<>`.

Constraints: *`CHECK-TYPES`*`()` is a well-formed expression.

Effects: Equivalent to:

```
CHECK-TYPES();
return CS();
```

[...]

[exec.continues.on]

[...]

The exposition-only class template *`impls-for`* ([exec.snd.expos]) is specialized for `continues_on_t` as follows:

```
namespace std::execution {
  template<>
  struct impls-for<continues_on_t> : default-impls {
    static constexpr auto get-state = see below;
    static constexpr auto complete = see below;

    template<class Sndr, class... Env>
      static consteval void check-types();

  };

}
```

The member *`impls-for`*`<continues_on_t>::`*`get-state`* is initialized with a callable object equivalent to the following lambda:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(see below)
    requires sender_in<child-type<Sndr>, FWD-ENV-T(env_of_t<Rcvr>)> {
  auto& [_, sch, child] = sndr;

  using sched_t = decltype(auto(sch));
  using ~~variant_t~~storage_t = see below;
```

```
  using receiver_t = see below;
  using operation_t = connect_result_t<
    schedule_result_t<sched_t>, receiver_t>;
  constexpr bool nothrow = noexcept(
    connect(schedule(sch), receiver_t{nullptr}));

  struct state-type {
    Rcvr& rcvr;                                  // exposition only
    variant_t async_resultstorage_t storage;     // exposition only
    operation_t op-state;                        // exposition only

    explicit state-type(sched_t sch, Rcvr& rcvr) noexcept(nothrow)
      : rcvr(rcvr), op-state(connect(schedule(sch), receiver_t{this})) {}
  };

  return state-type{sch, rcvr};
}

template<class Sndr, class... Env>
  static consteval void check-types();
```

Let *ONLY-VALUE-COMPLETIONS*(Sigs) denote a specialization of completion_signatures comprising the template arguments of Sigs whose completion tag is set_value_t.

Effects: Equivalent to:

```
get_completion_signatures<
  schedule_result_t<data-type<Sndr>>, FWD-ENV-T(Env)...>();
auto cs = get_completion_signatures<child-type<Sndr>, FWD-ENV-T(Env)...>();
decay-copyable-result-datums(cs);   // see [exec.snd.expos]
storage_for_completion_signatures<ONLY-VALUE-COMPLETIONS(decltype(cs))>::
  get_completion_signatures();
```

Objects of the local class *state-type* can be used to initialize a structured binding.

Let AllSigs be a pack of the arguments to the completion_signatures specialization named by completion_signatures_of_t<*child-type*<Sndr>, *FWD-ENV-T*(env_of_t<Rcvr>)>. Let Sigs be a pack of those types in AllSigs with a return type of set_value_t. Then storage_t denotes the type storage_for_completion_signatures<completion_signatures<Sigs>>.Let *as-tuple* be an alias template such that *as-tuple*<Tag(Args...)> denotes the type *decayed-tuple*<Tag, Args...>, and let *is-nothrow-decay-copy-sig* be a variable template such that auto(*is-nothrow-decay-copy-sig*<Tag(Args...)>) is a constant expression of type bool and equal to (is_nothrow_constructible_v<decay_t<Args>, Args> && ...). Let *error-completion* be a pack consisting of the type set_error_t(exception_ptr) if

~~(is-nothrow-decay-copy-sig<Sigs> &&...) is false, and an empty pack otherwise. Then variant_t denotes the type variant<monostate, *as-tuple*<Sigs>..., *error-completion*...>, except with duplicate types removed.~~

`receiver_t` is an alias for the following exposition-only class:

```
namespace std::execution {
  struct receiver-type {
    using receiver_concept = receiver_tag;
    state-type* state;          // exposition only

    void set_value() && noexcept {
      visit(
        [this]<class Tuple>(Tuple& result) noexcept -> void {
          if constexpr (!same_as<monostate, Tuple>) {
            auto& [tag, ...args] = result;
            tag(std::move(state->rcvr), std::move(args)...);
          }
        },
        state->async-result);
      state->storage.complete(std::move(state->rcvr));
    }

    template<class Error>
    void set_error(Error&& err) && noexcept {
      execution::set_error(std::move(state->rcvr), std::forward<Error>(err));
    }

    void set_stopped() && noexcept {
      execution::set_stopped(std::move(state->rcvr));
    }

    decltype(auto) get_env() const noexcept {
      return FWD-ENV(execution::get_env(state->rcvr));
    }
  };
}
```

The expression in the `noexcept` clause of the lambda is `true` if the construction of the returned *`state-type`* object is not potentially throwing; otherwise, `false`.

The member *`impls-for`*`<continues_on_t>::`*`complete`* is initialized with a callable object equivalent to the following lambda:

```
[]<class Tag, class... Args>(
  auto, auto& state, auto& rcvr, Tag, Args&&... args) noexcept -> void {
- using result_t = decayed-tuple<Tag, Args...>;
- constexpr bool nothrow =
-   (is_nothrow_constructible_v<decay_t<Args>, Args> && ...);

- try {
-   state.async-result.template emplace<result_t>(
-     Tag(),
-     std::forward<Args>(args)...);
- } catch (...) {
-   if constexpr (!nothrow)
-     state.async-result.template emplace<
-       tuple<set_error_t, exception_ptr>>(set_error, current_exception());
- }
  state.storage.arrive(Tag(), std::forward<Args>(args)...);
  start(state.op-state);
};
```

Let `out_sndr` be a subexpression denoting a sender returned from `continues_on_t(sndr, sch)` or one equal to such, and let `OutSndr` be the type `decltype((out_sndr))`. Let `out_rcvr` be a subexpression denoting a receiver that has an environment of type `Env` such that `sender_in<OutSndr, Env>` is `true`. Let `op` be an lvalue referring to the operation state that results from connecting `out_sndr` with `out_rcvr`. Calling `start(op)` shall start `sndr` on the current execution agent and execute completion operations on `out_rcvr` on an execution agent of the execution resource associated with `sch`. If scheduling onto `sch` fails, an error completion on `out_rcvr` shall be executed on an unspecified execution agent.

[exec.when.all]

[...]

```
template<class Sndr, class... Env>
  static consteval void check-types();
```

Let `Is` be the pack of integral template arguments of the `integer_sequence` specialization denoted by *`indices-for`*`<Sndr>`.

Effects: Equivalent to:

```
auto fn = []<class Child>() {
  auto cs = get_completion_signatures<Child, when-all-env<Env>...>();
  if constexpr (cs.count-of(set_value) >= 2)
```

```
    throw unspecified-exception();
  decay-copyable-result-datums(cs); // see [exec.snd.expos]
  storage_for_completion_signatures<decltype(cs)>::
    get_completion_signatures();
};
(fn.template operator()<child-type<Sndr, Is>>(), ...);
```

[...]

The member *impls-for*<when_all_t>::*get-state* is initialized with a callable object equivalent to the following lambda expression:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(noexcept(e))
  -> decltype(e) {
  return e;
}
```

where *e* is the expression

```
std::forward<Sndr>(sndr).apply(make-state<Rcvr>())
```

and where *make-state* is the following exposition-only class template:

```
enum class disposition { started, error, stopped };            // exposition only

template<class Rcvr>
struct make-state {
  template<class... Sndrs>
  auto operator()(auto, auto, Sndrs&&... sndrs) const {
    using values_tuplestorage = see below;
    using errors_variantstorage = see below;
    using stop_callback =
      stop_callback_for_t<stop_token_of_t<env_of_t<Rcvr>>, on-stop-request>;

    struct state-type {
      void arrive(Rcvr& rcvr) noexcept {                       // exposition only
        if (0 == --count) {
          complete(rcvr);
        }
      }

      void complete(Rcvr& rcvr) noexcept;                      // exposition only

      atomic<size_t> count{sizeof...(sndrs)};                  // exposition only
      inplace_stop_source stop_src{};                          // exposition only
```

```
      atomic<disposition> disp{disposition::started};          // exposition only
      errors_variantstorage errors{};                          // exposition only
      values_tuplestorage values{};                            // exposition only
      optional<stop_callback> on_stop{nullopt};                // exposition only
    };

    return state-type{};
  }
};
```

~~Let *copy-fail* be exception_ptr if decay-copying any of the child senders' result datums can potentially throw; otherwise, *none-such*, where *none-such* is an unspecified empty class type.~~

~~The alias values_tuple denotes the type~~

~~tuple<value_types_of_t<Sndrs, *FWD-ENV-T*(env_of_t<Rcvr>), *decayed-tuple*, optional>...>~~

~~if that type is well-formed; otherwise, tuple<>.~~

~~The alias errors_variant denotes the type variant<none-such, *copy-fail*, Es...> with duplicate types removed, where Es is the pack of the decayed types of all the child senders' possible error result datums.~~

Let *value-sigs* be an alias template that transforms a specialization completion_signatures<Fns...> into the specialization completion_signatures<Vs...>, where Vs is the pack of function types in Fns whose return types are set_value_t. The alias values_storage denotes the type

```
tuple<
  storage_for_completion_signatures<
    value-sigs<
      completion_signatures_of_t<Sndrs, when-all-env<env_of_t<Rcvr>>>>,
    storage_for_completion_signatures_error_policy::propagate>...>
```

Let Es be the pack of types denoted by error_types_of_t<Sndrs, *when-all-env*<env_of_t<Rcvr>>>..., with duplicate types removed. Let ThrowingEs be the pack of types denoted by Es... if values_storage::nothrow_arrive is true, otherwise the pack of types denoted by Es..., set_error_t(exception_ptr), with duplicate types removed. The alias errors_storage denotes the type storage_for_completion_signatures<completion_signatures<ThrowingEs...>>.

The member void *state-type*::*complete*(Rcvr& rcvr) noexcept behaves as follows:

- If disp is equal to *disposition*::*started*, evaluates:
  ~~auto tie = []<class... T>(tuple<T...>& t) noexcept {~~

~~ return tuple<T&...>(t);~~
~~};~~
~~auto set = [&](auto&... t) noexcept {~~
~~ set_value(std::move(rcvr), std::move(t)...);~~
~~};~~

on_stop.reset();
apply(
  [&](auto&... opts) noexcept {
~~   apply(set, tuple_cat(tie(*opts)...));~~
    visit_stored_completion(
      [&](auto&... completions) noexcept {
        if constexpr (sizeof...(completions) != 0) {
          set-completion-signature(completions...);
          apply(
            set_value,
            tuple_cat(
              forward_as_tuple(std::move(rcvr)),
              std::move(completions).forward_arguments()...));
        }
      },
      storages...);
  },
  values);

Where *set-completion-signature*(completions...) is expression-equivalent to the empty expression. Let Completions be the pack of types remove_reference_t<decltype(completions)>::signature_type... where completions denotes the pack of expressions provided to *set-completion-signature*. Let ResultDatums be the pack formed by concatenating, in order, each pack Args for every completion signature Tag(Args...) in Completions. The template argument of CS ([exec.snd.expos]) that corresponds to the completion operation potentially evaluated ([basic.def.odr]) by the invocation of set_value above is set_value_t(ResultDatums...) ([exec.async.ops]).

- Otherwise, if disp is equal to *disposition*::*error*, evaluates:

on_stop.reset();
~~visit(~~
~~ [&]<class Error>(Error& error) noexcept {~~
~~   if constexpr (!same_as<Error, *none_such*>) {~~
~~   set_error(std::move(rcvr), std::move(error));~~
~~   }~~
~~ },~~
~~ errors);~~
errors.complete(std::move(rcvr));

- Otherwise, evaluates:
  ```
  if constexpr (sends-stopped) {
    on_stop.reset();
    set_stopped(std::move(rcvr));
  }
  ```
  where *sends-stopped* equals `true` if and only if there exists an element `S` of `Sndrs` such that `completion_signatures_of_t<S, when-all-env<Env>>` contains `set_stopped_t()`.

The member *impls-for*`<when_all_t>::`*start* is initialized with a callable object equivalent to the following lambda expression:

```
[]<class State, class Rcvr, class... Ops>(
    State& state, Rcvr& rcvr, Ops&... ops) noexcept -> void {
  state.on_stop.emplace(
    get_stop_token(get_env(rcvr)),
    on-stop-request{state.stop_src});
  (start(ops), ...);
}
```

The member *impls-for*`<when_all_t>::`*complete* is initialized with a callable object equivalent to the following lambda expression:

```
[]<class Index, class State, class Rcvr, class Set, class... Args>(
  this auto& complete, Index, State& state, Rcvr& rcvr, Set, Args&&... args)
  noexcept -> void {
  if constexpr (same_as<Set, set_error_t>) {
    if (disposition::error != state.disp.exchange(disposition::error)) {
      state.stop_src.request_stop();
      ~~TRY-EMPLACE-ERROR(state.errors, std::forward<Args>(args)...);~~
      state.errors.arrive(Set{}, std::forward<Args>(args)...);
    }
  } else if constexpr (same_as<Set, set_stopped_t>) {
    auto expected = disposition::started;
    if (state.disp.compare_exchange_strong(expected, disposition::stopped)) {
      state.stop_src.request_stop();
    }
  } else if constexpr (!same_as<decltype(State::values), tuple<>>) {
    if (state.disp == disposition::started) {
      auto& ~~opt~~storage = get<Index::value>(state.values);
      ~~TRY-EMPLACE-VALUE(complete, opt, std::forward<Args>(args)...);~~
      try {
        storage.arrive(Set{}, std::forward<Args>(args)...);
      } catch (...) {
```

```
        if constexpr (!storage.nothrow_arrive) {
          complete(Index(), state, rcvr, set_error, current_exception());
        }
      }
    }
  }
  state.arrive(rcvr);
}
```

~~where *TRY-EMPLACE-ERROR*(v, e), for subexpressions v and e, is equivalent to:~~

```
try {
  v.template emplace<decltype(auto(e))>(e);
} catch (...) {
  v.template emplace<exception_ptr>(current_exception());
}
```

~~if the expression decltype(auto(e))(e) is potentially throwing; otherwise, v.template emplace<decltype(auto(e))>(e); and where *TRY-EMPLACE-VALUE*(c, o, as...), for subexpressions c, o, and pack of subexpressions as, is equivalent to:~~

```
try {
  o.emplace(as...);
} catch (...) {
  c(Index(), state, rcvr, set_error, current_exception());
  return;
}
```

~~if the expression *decayed-tuple*<decltype(as)...>{as...} is potentially throwing; otherwise, o.emplace(as...).~~

The expression `when_all_with_variant(sndrs...)` is expression-equivalent to *`make-sender`*`(when_all_with_variant, {}, sndrs...)`.

[...]

[exec.spawn.future]

[...]

Let *`spawn-future-state-base`* be the exposition-only class template:

```
namespace std::execution {
  template<class Completions>
  struct spawn-future-state-base;            // exposition only
```

```
  template<class... Sigs>
  struct spawn-future-state-base<completion_signatures<Sigs...>>
    : try-cancelable {                                  // exposition only
    using variant-t = see below;                        // exposition only
    variant-t result;                                   // exposition only
    storage_for_completion_signatures<Completions>
      storage;                                          // exposition only
    virtual void complete() noexcept = 0;  // exposition only
  };
}
```

Let Sigs be the pack of arguments to the completion_signatures specialization provided as a parameter to the *spawn-future-state-base* class template. Let *as-tuple* be an alias template that transforms a completion signature Tag(Args...) into the tuple specialization *decayed-tuple*<Tag, Args...>.

If is_nothrow_constructible_v<decay_t<Arg>, Arg> is true for every type Arg in every parameter pack Args in every completion signature Tag(Args...) in Sigs then *variant-t* denotes the type variant<monostate, tuple<set_stopped_t>, *as-tuple*<Sigs>...>, except with duplicate types removed.

Otherwise *variant-t* denotes the type variant<monostate, tuple<set_stopped_t>, tuple<set_error_t, exception_ptr>, *as-tuple*<Sigs>...>, except with duplicate types removed.

Let *spawn-future-receiver* be the exposition-only class template:

```
namespace std::execution {
  template<class Completions>
  struct spawn-future-receiver {                    // exposition only
    using receiver_concept = receiver_tag;

    spawn-future-state-base<Completions>* state;  // exposition only

    template<class... T>
      void set_value(T&&... t) && noexcept {
        set-complete<set_value_t>(std::forward<T>(t)...);
      }

    template<class E>
      void set_error(E&& e) && noexcept {
        set-complete<set_error_t>(std::forward<E>(e));
      }
```

```
    void set_stopped() && noexcept {
      set-complete<set_stopped_t>();
    }

  private:
    template<class CPO, class... T>
      void set-complete(T&&... t) noexcept {       // exposition only
        constexpr bool nothrow =
          (is_nothrow_constructible_v<decay_t<T>, T> && ...);
        try {
          state->result.template emplace<decayed-tuple<CPO, T...>>(
            CPO{}, std::forward<T>(t)...);
        }
        catch (...) {
          if constexpr (!nothrow) {
            using tuple_t = decayed-tuple<set_error_t, exception_ptr>;
            state->result.template emplace<tuple_t>(
              set_error_t{}, current_exception());
          }
        }
        state->storage.arrive(CPO{}, std::forward<T>(t)...);
        state->complete();
      }
  };
}
```

[...]

```
void consume(receiver auto& rcvr) noexcept;
```

Effects:

- If this invocation of *consume* happens before an invocation of *complete* on `*this` and no invocation of *try-set-stopped* on `*this` happened before this invocation of *consume* then `rcvr` is registered to be completed when *complete* is subsequently invoked on `*this`;
- otherwise, if this invocation of *consume* happens after an invocation of *try-set-stopped* on `*this` and no invocation of *complete* on `*this` happened before this invocation of *consume* then `rcvr` is completed as if by `set_stopped(std::move(rcvr))`;
- otherwise, `rcvr` is completed as if by:

```
std::move(this->result).visit(
  [&rcvr](auto&& tuple) noexcept {
```

```
      if constexpr (!same_as<
          remove_reference_t<decltype(tuple)>, monostate>) {
        apply([&rcvr](auto cpo, auto&&... vals) {
          cpo(std::move(rcvr), std::move(vals)...);
        }, std::move(tuple));
      }
    });
    this->result.complete(std::move(rcvr));
    destroy();
```

[...]

[exec.as.awaitable]

[...]

The type *sender-awaitable*<Sndr, Promise> is equivalent to:

```
namespace std::execution {
  template<class Sndr, class Promise>
  class sender-awaitable {
    struct unit {};                                                  // exposition only
    using signatures = see below;                                    // exposition only
    using value-type = see below;                                    // exposition only
    single-sender-value-type<Sndr, env_of_t<Promise>>;
    using resultstorage-type =                                       // exposition only
      conditional_t<is_void_v<value-type>, unit, value-type>;
      storage_for_completion_signatures<signatures>;
    struct awaitable-receiver;                                       // exposition only

    variant<monostate, result-type, exception_ptr> result{};         // exposition only
    storage-type storage;                                            // exposition only

    connect_result_t<Sndr, awaitable-receiver> state;                // exposition only

  public:
    sender-awaitable(Sndr&& sndr, Promise& p);
    static constexpr bool await_ready() noexcept { return false; }
    void await_suspend(coroutine_handle<Promise>) noexcept { start(state); }
    value-type await_resume();
  };
}
```

signatures is the specialization of `completion_signatures` whose template arguments are the completion signatures in `completion_signatures_of_t<Sndr, env_of_t<Promise>>` other than `set_stopped_t()`.

value-type is defined as follows:

- If *signatures* contains no `set_value_t` completion signature, *value-type* denotes `void`
- Otherwise, let `set_value_t(Ts...)` be the sole `set_value_t` completion signature in *signatures*
  - If `sizeof...(Ts)` is zero, *value-type* denotes `void`
  - Otherwise, if `Ts...` denotes a single type `T`, *value-type* denotes `T`
  - Otherwise, *value-type* denotes `tuple<Ts...>`.

awaitable-receiver is equivalent to:

```
struct awaitable-receiver {
  using receiver_concept = receiver_tag;
  variant<monostate, result-type, exception_ptr>* result-ptr;  // exposition only
  storage-type* storage-ptr;                                   // exposition only
  coroutine_handle<Promise> continuation;                      // exposition only
  // see below
};
```

Let `rcvr` be an rvalue expression of type *awaitable-receiver*, let `crcvr` be a const lvalue that refers to `rcvr`, let `vs` be a pack of subexpressions, and let `err` be an expression of type `Err`. Let *MAKE-NOEXCEPT*`(expr)` for some subexpression expr be expression-equivalent to `[&] noexcept -> decltype(auto) { return (expr); }()`. Then:

- The expression `set_value(rcvr, vs...)` is equivalent to:
  ```
  try {
    rcvr.result-ptr->template emplace<1>(vs...);
  } catch(...) {
    rcvr.result-ptr->template emplace<2>(current_exception());
  }
  rcvr.storage-ptr->arrive(set_value, vs...);
  MAKE-NOEXCEPT(rcvr.continuation.resume());
  ```
  Mandates: `constructible_from<result-type, decltype((vs))...>` is satisfied.
- The expression `set_error(rcvr, err)` is equivalent to:
  ```
  try {
    rcvr.result-ptr->template emplace<2>(AS-EXCEPT-PTR(err));
      // see [exec.general]
  } catch(...) {
    rcvr.result-ptr->template emplace<2>(current_exception());
  ```

```
}
rcvr.storage-ptr->arrive(set_error, err);
MAKE-NOEXCEPT(rcvr.continuation.resume());
```

- The expression `set_stopped(rcvr)` is equivalent to:

```
MAKE-NOEXCEPT(
  static_cast<coroutine_handle<>>(
    rcvr.continuation.promise().unhandled_stopped()).resume());
```

- For any expression `tag` whose type satisfies *forwarding-query* and for any pack of subexpressions `as`, `get_env(crcvr).query(tag, as...)` is expression-equivalent to:

```
tag(get_env(as_const(MAKE-NOEXCEPT(crcvr.continuation.promise()))),
    as...)
```

```
sender-awaitable(Sndr&& sndr, Promise& p);
```

Effects: Initializes *state* with

```
connect(std::forward<Sndr>(sndr),
        awaitable-receiver{addressof(resultstorage),
                           coroutine_handle<Promise>::from_promise(p)})
```

```
value-type await_resume();
```

Effects: Equivalent to:

```
if (result.index() == 2)
  rethrow_exception(get<2>(result));
if constexpr (!is_void_v<value-type>)
  return std::forward<value-type>(get<1>(result));
return visit_stored_completion(
  std::move(result),
  [&](auto&&... completions) -> value-type {
    if constexpr (sizeof...(completions)) {
      return [&](auto&& completion) -> value-type {
        if constexpr (is_same_v<set_error_t, decltype(completion.tag())>) {
          auto&& err = std::get<0>(
            std::forward<decltype(completion)>(completion).arguments());
          if constexpr (
            is_same_v<exception_ptr, remove_cvref_t<decltype(err)>>)
          {
            rethrow_exception(std::forward<decltype(err)>(err));
          } else {
            rethrow_exception(
              AS-EXCEPT-PTR(std::forward<decltype(err)>(err)));
          }
        } else {
```

```
          using tuple_type =
            remove_cvref_t<decltype(completion.arguments())>;
          constexpr size_t arguments = tuple_size_v<tuple_type>;
          if constexpr (arguments > 1) {
            return
              std::forward<decltype(completion)>(completion).arguments();
          } else if constexpr (arguments) {
            return std::get<0>(
              std::forward<decltype(completion)>(completion).arguments());
          }
        }
      }(std::forward<decltype(completions)>(completions)...);
    }
  });
```

`as_awaitable` is a customization point object. For subexpressions `expr` and `p` where `p` is an lvalue, `Expr` names the type `decltype((expr))` and `Promise` names the type `decay_t<decltype((p))>`, `as_awaitable(expr, p)` is expression-equivalent to, except that the evaluations of `expr` and `p` are indeterminately sequenced:

[...]

Open Design Questions

- Should `storage_for_completion_signatures::completion_signatures` be exposition only?

Implementation Experience

This paper has been implemented against nVidia's reference implementation of `std::execution` [?].

Revision History

R1

- Added implementation experience

Acknowledgements

The author would like to thank Ian Petersen and Dietmar Kühl for reviewing this paper.

References

[1] K. Shoop. async-object - aka async-RAII P2849R0
[2] I. Petersen. Towards Senders in Interfaces P4223R0
[3] L. Baker et al. Eliminating heap-allocations in sender/receiver with connect()/start() as basis operations P2006R1
[4] E. Niebler et al. One-Way execute is a Poor Basis Operation P1525R1
[5] M. Dominiak et al. std::execution P2300R10
[6] R. Leahy. Remove std::execution::split P3682R0
[7] R. Leahy. Of Operation States and Their Lifetimes P3373R4
[8] R. Leahy. when_all Oughtn't Hallucinate set_stopped P4269R0
[9] V. Voutilainen. We cannot (realistically) get rid of throwing moves P0129R0
[10] R. Leahy. Can set_value Actually Throw? CppCon 2025
[11] https://www.ronseal.com/
[12] R. Leahy. Make when_all a Ronseal Algorithm P3887R1
[13] R. Leahy. unless_stop_requested P3892R0
[14] R. Leahy. std::execution::decay_copy P4293R0
[15] R. Martin. The Single Responsibility Principle. The Clean Code Blog
[16] R. Leahy. It’s Scopes All the Way Down P3955R0
[17] https://github.com/NVIDIA/stdexec/pull/2153