`variant_sender`

Document Number: P4329R0
Date: 2026-07-29
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: SG1, LEWG

Abstract

This paper proposes an asynchronous branching primitive: `std::execution::variant_sender`.

Background

The current C++29 working draft, particularly the specification of `std::execution` therein, has no asynchronous branching primitive.

Branching constructs are clearly useful [1].

Discussion

Under the status quo of the C++29 working draft one can write asynchronous code which branches synchronously:

```
sndr | std::execution::then([](const int i) {
  if (i) {
    std::cout << “Non-zero integer”;
  } else {
    std::cout << “Zero”;
  }
  std::cout << std::endl;
})
```

And asynchronous code which differentially parameterizes an asynchronous operation based on asynchronously-obtained values:

```
sndr | std::execution::let_value([](const int i) {
  if (i) {
    return std::execution::just(true);
  }
```

```
  return std::execution::just(false);
})
```

What cannot be done, at least not in a straightforward way, is selecting fundamentally different operations in the middle of an asynchronous operation (i.e. a general purpose asynchronous branch):

```
sndr | std::execution::let_value([](const int i) {
  if (i) {
    return std::execution::just(true);
  }
  return
    std::execution::just() |
    std::execution::then([]() {
      std::cout << “Zero” << std::endl;
      return false;
    });
})
```

The above code doesn’t compile because return type deduction for the outer lambda observes return statements whose operands have different types.

The solution is to collapse these two different types into a single, sum type à la `std::variant`:

```
sndr | std::execution::let_value([](const int i) {
  auto a = std::execution::just(true);
  auto b =
    std::execution::just() |
    std::execution::then([]() {
      std::cout << “Zero” << std::endl;
      return false;
    });
  using type = std::execution::variant_sender<decltype(a), decltype(b)>;
  if (i) {
    return type(std::move(a));
  }
  return type(std::move(b));
})
```

Because of the similarity to `std::variant` the name `std::execution::variant_sender` is proposed.

Wording

[execution.syn]

[...]

```
namespace std::execution {
  // [exec.consumers], consumers
  struct spawn_t { unspecified };
  inline constexpr spawn_t spawn{};

  // [exec.snd.variant], variant_sender
  template<sender... Sndrs>
    struct variant_sender;

  // [exec.cmplsig], completion signatures
  template<class Fn>
    concept completion-signature = see below;  // exposition only
  [...]
}
```

[...]

[execution.snd.variant]

Note: This is a new section.

A `variant_sender` is a sender which behaves in the same way as whichever child sender is its currently-active alternative.

Let *`variant-operation-state`* denote the following exposition-only class template:

```
template<class Rcvr, class... Sndrs>
  struct variant-operation-state {
    using operation_state_concept = operation_state_tag;

    variant-or-empty<connect_result_t<Sndrs, Rcvr>...> op;  // exposition only

    constexpr void start() & noexcept {
      visit(execution::start, op);
    }

    template<class Sndr>
```

```
    constexpr variant-operation-state(Sndr&& sndr, Rcvr rcvr) noexcept(
      noexcept(connect(declval<Sndr>(), declval<Rcvr>())))
      : op(in_place_type<connect_result_t<Sndr, Rcvr>>,
           emplace-from{[&] {
             return connect(std::forward<Sndr>(sndr), std::move(rcvr));
           }}) {}
  };
```

Let *combine-sigs* be the following exposition-only function template:

```
template<valid-completion-signatures... Sigs>
  consteval valid-completion-signatures auto combine-sigs(Sigs...);
```

Let `Fns` be the pack formed by concatenating, in order, the arguments of the `completion_signatures` specializations named by the types in `Sigs`.

Returns: `completion_signatures<Us...>()`, where `Us...` is `Fns...` with duplicate types removed.

```
template<sender... Sndrs>
  struct variant_sender {
    using sender_concept = sender_tag;

    template<class Self, class T>
      using like-t = decltype(                    // exposition only
        std::forward_like<Self>(declval<T&>()));

    template<class Self, class Rcvr>
      using op-t = variant-operation-state<       // exposition only
        Rcvr,
        like-t<Self, Sndrs>...>;

    using first-env-t = env_of_t<Sndrs...[0]>;    // exposition only
    using variant-t = variant<Sndrs...>;          // exposition only

    variant-t sndr;                               // exposition only

    template<class Self, class... Env>
      static consteval auto get_completion_signatures() {
        return combine-sigs(
          execution::get_completion_signatures<
            like-t<Self, Sndrs>, Env...>()...));
      }
```

```
  template<class Self, receiver Rcvr>
    requires (sender_in<like-t<Self, Sndrs>, env_of_t<Rcvr>> && ...)
  constexpr op-t<Self, Rcvr> connect(this Self&& self, Rcvr rcvr) noexcept(
    (is_nothrow_constructible_v<
      op-t<Self, Rcvr>,
      like-t<Self, Sndrs>,
      Rcvr> && ...))
  {
    return visit(
      [&](auto&& sndr) {
        return op-t<Self, Rcvr>(
          std::forward_like<Self>(sndr),
          std::move(rcvr));
      },
      self.sndr);
  }

  constexpr first-env-t get_env() const noexcept
    requires (is_same_v<first-env-t, env_of_t<Sndrs>> && ...)
  {
    return visit(execution::get_env, sndr);
  }

  template<class... Args>
    requires is_constructible_v<variant-t, Args...>
  constexpr explicit variant_sender(Args&&... args) noexcept(
    is_nothrow_constructible_v<variant-t, Args...>)
    : sndr(std::forward<Args>(args)...) {}

  variant_sender& operator=(const variant_sender&) = delete;
};
```

A program that instantiates the definition of `variant_sender` with no template arguments is ill-formed.

Implementation Experience

Nvidia's stdexec ships `exec::variant_sender` [2].

References

[1] C. Böhm et al. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. Communications of the ACM Volume 9 Number 5

[2] https://github.com/NVIDIA/stdexec/blob/f0e8ae6fdc6c188389b146bb854b80d399724b04/include/exec/variant_sender.hpp