Using Sender/Receiver to Implement Control Flow for Async Processing

Steve Downey
ABSTRACT

How can P2300 Senders be composed using sender adapters and sender factories to provide arbitrary program control flow?
ABSTRACT

How can P2300 Senders be composed using sender adapters and sender factories to provide arbitrary program control flow?

• How do I use these things?
ABSTRACT

How can P2300 Senders be composed using sender adapters and sender factories to provide arbitrary program control flow?

- How do I use these things?
- Where can I steal from?
std::execution

P2300

Recent version at https://isocpp.org/files/papers/P2300R7.html

A self-contained design for a Standard C++ framework for managing asynchronous execution on generic execution resources.
THREE KEY ABSTRACTIONS
THREE KEY ABSTRACTIONS

1. Schedulers
THREE KEY ABSTRACTIONS

1. Schedulers
2. Senders
THREE KEY ABSTRACTIONS

1. Schedulers
2. Senders
3. Receivers
SCHEDULERS

Responsible for scheduling work on execution resources.
SCHEDULERS

Responsible for scheduling work on execution resources.

Execution resources are things like threads, GPUs, and so on.
SCHEDULERS

Responsible for scheduling work on execution resources.

Execution resources are things like threads, GPUs, and so on.

Sends work to be done in a place.
SENDERS

Senders describe work.
RECEIVERS

 Receivers are where work terminates.
Work can terminate in three different ways.

- Return a value.
- Throw an exception
- Be canceled

It's been a few minutes. Let's see some simple code.
RECEIVERS

Receivers are where work terminates.

• Value channel
RECEIVERS

Receivers are where work terminates.

- Value channel
- Error channel
RECEIVERS

Receivers are where work terminates.

- Value channel
- Error channel
- Stopped channel
```cpp
#include <stdexec/execution.hpp>
#include <exec/static_thread_pool.hpp>
#include <iostream>

int main() {
    exec::static_thread_pool pool(8);
    stdexec::scheduler auto sch = pool.get_scheduler();
    stdexec::sender auto begin = stdexec::schedule(sch);
    stdexec::sender auto hi = stdexec::then(begin, [] {
        std::cout << "Hello world! Have an int.\n";
        return 13;
    });
    auto add_42 = stdexec::then(hi, [] (int arg) { return arg + 42; });
    auto [i] = stdexec::sync_wait(add_42).value();
    std::cout << "The int is " << i << '\n';
    return 0;
}
```
Hello world! Have an int.
The int is 55
WHEN ALL - CONCURENT ASYNC

```
1: exec::static_thread_pool pool(3);
2:
3: auto sched = pool.get_scheduler();
4:
5: auto fun = [](int i) { return i * i; };
6:
7: auto work = stdexec::when_all(
8:   stdexec::on(sched, stdexec::just(0) | stdexec::then(fun)),
9:   stdexec::on(sched, stdexec::just(1) | stdexec::then(fun)),
10:  stdexec::on(sched, stdexec::just(2) | stdexec::then(fun)));
11:
12: auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
13:
14: std::printf("%d %d %d\n", i, j, k);
```
Describe some work:

Creates 3 sender pipelines that are executed concurrently by passing to `when_all`

Each sender is scheduled on `sched` using `on` and starts with `just(n)` that creates a Sender that just forwards `n` to the next sender.

After `just(n)`, we chain `then(fun)` which invokes `fun` using the value provided from `just()`

Note: No work actually happens here. Everything is lazy and `work` is just an object that statically represents the work to later be executed
WHEN ALL - CONCURRENT ASYNC - RESULTS
Order of execution is by chance, order of results is determined.
DYNAMIC CHOICE OF SENDER

```cpp
1: exec::static_thread_pool pool(3);
2: auto sched = pool.get_scheduler();
3:
4: auto fun = [](int i) -> stdexec::sender auto {
5:   using namespace std::string_literals;
6:   if ((i % 2) == 0) {
7:     return stdexec::just("even"s);
8:   } else {
9:     return stdexec::just("odd"s);
10:   }
11: }; 
12:
13: auto work = stdexec::when_all(
14:   stdexec::on(sched, stdexec::just(0) | stdexec::let_value(fun)),
15:   stdexec::on(sched, stdexec::just(1) | stdexec::let_value(fun)),
16:   stdexec::on(sched, stdexec::just(2) | stdexec::let_value(fun)));
17:
18: auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
19: 
20: std::printf("%s %s %s", i.c_str(), j.c_str(), k.c_str());
21: 
22: Compiler Explorer
```
ENOUGH API TO TALK ABOUT CONTROL FLOW

The minimal set being:

- stdexec::on
- stdexec::just
- stdexec::then
- stdexec::let_value
- stdexec::sync_wait

I will mostly ignore the error and stop channels
VIGOROUS HANDWAVING
SOME THEORY

Continuation Passing Style
NOT AT ALL NEW

Sussman and Steele in 1975

AI Memo 349: "Scheme: An Interpreter for Extended Lambda Calculus"
PASS A "CONTINUATION"

Where to go next rather than return the value.

```haskell
add :: Float -> Float -> Float
add a b = a + b

add_cps :: Float -> Float -> (Float -> a) -> a
add_cps a b cont = cont (a + b)
```
auto add(float a, float b) -> float {
  return a + b;
}

template<typename Cont>
auto add_cps(float a, float b, Cont k) {
  return k(a+b);
}
INHERENTLY A TAIL CALL

In continuation passing style we never return.

We send a value to the rest of the program.

Hard to express in C++.

Extra machinery necessary to do the plumbing.

Also, some risk, so we don't always do TCO.

We keep the sender "thunks" live so we don't dangle references.
INTERMITTENTLY POPULAR AS A COMPILER TECHNIQUE

The transformations of direct functions to CPS are mechanical. The result is easier to optimize and mechanically reason about.

Equivalent to Single Static Assignment.

Structured Programming can be converted to CPS.
DELIMITED CONTINUATIONS

General continuations reified as a function.

Everyone knows that when a process executes a system call like ‘read’, it gets suspended. When the disk delivers the data, the process is resumed. That suspension of a process is its continuation. It is delimited: it is not the check-point of the whole OS, it is the check-point of a process only, from the invocation of main() up to the point main() returns. Normally these suspensions are resumed only once, but can be zero times (exit) or twice (fork).

Oleg Kiselyov Fest2008-talk-notes.pdf
If this quote reminds you of coroutines, you are paying attention.
HASKELL'S CONT TYPE

This is roughly equivalent to the sender value channel. A Cont takes a receiver, a function that consumes the value being sent, and produces an r, the result type.

The identity function is often used.
UNDERLIES std::execution

The plumbing is hidden.

Senders "send" to their continuations, delimited by the Receiver.
ANOTHER LEVEL OF INDIREDION
SOLVES ALL PROBLEMS
SOLVES ALL PROBLEMS

Adds two more.
SOLVES ALL PROBLEMS

Adds two more.

At least
CPS INDIRECTS FUNCTION RETURN

Transform a function

\[ A \rightarrow B \]

to

\[ A \rightarrow B \rightarrow (B \rightarrow R) \rightarrow R \]

add :: Float -> Float -> Float
add a b = a + b

add_cps :: Float -> Float -> (Float -> A) -> A
add_cps a b cont = cont (a + b)
SENDER CLOSES OVER A

\[ B \rightarrow (B \rightarrow R) \rightarrow R \]

The A is (mostly) erased from the Sender.
RECIEVER IS THE TRANSFORM TO RESULT

\[(B \rightarrow R) \rightarrow R\]
SOME PICTURES
SENDER
just

stdexec::just(0)
auto f(A a) -> B;
auto s = stdexec::just(a) | stdexec::then(f);
let_value

```cpp
sender_of<set_value_t(B)> auto snd(A a);
auto s = stdexec::just(a) | stdexec::let_value(snd);
```
IN WHICH WE USE THE M WORD
SENDER IS A MONAD
SENDER IS A MONAD

(surprise)
SENDER IS A MONAD

(surprise)

(shock, dismay)
FUNCTION COMPOSITION IS THE HINT

Functions are units of work.
We compose them into programs.
The question is if the rules apply.
MONADIC INTERFACE

**bind or and**\_**then**

\[
M\langle a \rangle \rightarrow (a \rightarrow M\langle b \rangle) \rightarrow M\langle b \rangle
\]

**fish or kleisli arrow**

\[
(a \rightarrow M\langle b \rangle) \rightarrow (b \rightarrow M\langle c \rangle) \rightarrow (a \rightarrow M\langle c \rangle)
\]

**join or flatten or mconcat**

\[
M\langle M\langle a \rangle \rangle \rightarrow M\langle a \rangle
\]
Monad Interface
APPLICATIVE AND FUNCTOR PARTS

make or pure or return

\[ a \to M\langle a \rangle \]

fmap or transform

\[ (a \to b) \to M\langle a \rangle \to M\langle b \rangle \]

Any one of the first three and one of the second two can define the other three
MONAD LAWS

**left identity**
\[ \text{bind}(\text{pure}(a), \text{h}) = \text{h}(a) \]

**right identity**
\[ \text{bind}(m, \text{pure}) = m \]

**associativity**
\[ \text{bind}(\text{bind}(m, g), \text{h}) = \text{bind}(m, \text{bind}(\lambda x \rightarrow g(x), \text{h})) \]
SENDER IS THREE MONADS IN A TRENCH-COAT

Stacked up.

- Value
- Error
- Stopped
The three channels can be crossed, mixed, and remixed. Focus on the value channel for simplicity.
THE THREE MONADIC PARTS
just
Send a value.
pure
just lifts a value into the monad
then

Send a value returned from a function that takes its argument from a Sender.

fmap or transform
then is the functor fmap
let_value

Send what is returned by a Sender returned from a function that takes its argument from a Sender.
bind
let value is the monadic bind
NECESSARY AND SUFFICIENT

The monadic bind gives us the runtime choices we need.
BASIS OF CONTROL

- Sequence
- Decision
- Recursion
SEQUENCE

```cpp
stdexec::sender auto work =
    stdexec::schedule(sch)
    | stdexec::then([] { std::cout << "Hello world! Have an int.";
                        return 13;
    });
    | stdexec::then([](int arg) { return arg + 42; });

auto [i] = stdexec::sync_wait(work).value();
```
One thing after another.
1: `exec::static_thread_pool pool(8);`
2: 
3: `stdexec::scheduler auto sch = pool.get_scheduler();`
4: 
5: `stdexec::sender auto begin = stdexec::schedule(sch);`
6: `stdexec::sender auto seven = stdexec::just(7);`
7: `stdexec::sender auto eleven = stdexec::just(11);`
8: 
9: `stdexec::sender auto branch =`
10: begin
11: | stdexec::then([]() { return std::make_tuple(5, 4); })
12: | stdexec::let_value("
13: | [=](auto tpl) {
14: | auto const& [i, j] = tpl;
15: |
16: | return tst((i > j),
17: | seven | stdexec::then([&](int k) noexcept {
18: | std::cout << "true branch " << k << 'n';
19: | }),
20: | eleven | stdexec::then([&](int k) noexcept {
21: | std::cout << "false branch " << k << 'n';
22: | })}));
23: |
24: 
25: `stdexec::sync_wait(std::move(branch));`
true branch 7
Control what sender is sent at runtime depending on the state of the program when the work is executing rather than in the structure of the senders.
tst FUNCTION

```cpp
inline auto tst = [] (bool cond,
  stdex::sender auto left,
  stdex::sender auto right) -> any_sender_of<stdexec::set_value_t> {
  if (cond) {
    return left;
  } else {
    return right;
  }
};
```
RECURSION
SIMPLE RECURSION

1: using any_int_sender =
2:    any_sender_of<stdexec::set_value_t<int>,
3:        stdexec::set_stopped_t(),
4:        stdexec::set_error_t(std::exception_ptr)>;
5: auto fac(int n) -> any_int_sender {
6:    std::cout << "factorial of " << n << "\n";
7:    if (n == 0)
8:        return stdexec::just(1);
9:    return stdexec::just(n - 1)
10:       | stdexec::let_value([](int k) { return fac(k); })
11:       | stdexec::then([n](int k) { return k * n; });
12:   }
13: }
int k = 10;

stdexec::sender auto factorial =
    begin
        | stdexec::then([=]() { return k; })
        | stdexec::let_value([](int k) { return fac(k); });
    std::cout << "factorial built\n\n";

auto [i] = stdexec::sync_wait(std::move(factorial)).value();
std::cout << "factorial " << k << " = " << i << '\n';
factorial built
factorial of 10
factorial of 9
factorial of 8
factorial of 7
factorial of 6
factorial of 5
factorial of 4
factorial of 3
factorial of 2
factorial of 1
factorial of 0
factorial 10 = 3628800
GENERAL RECURSION

```cpp
auto fib(int n) -> any_int_sender {
    if (n == 0)
        return stdexec::on(getDefaultScheduler(), stdexec::just(0));
    if (n == 1)
        return stdexec::on(getDefaultScheduler(), stdexec::just(1));
    auto work = stdexec::when_all(
        stdexec::on(getDefaultScheduler(), stdexec::just(n - 1)) |
        stdexec::let_value([[]](int k) { return fib(k); }) |
        stdexec::on(getDefaultScheduler(), stdexec::just(n - 2)) |
        stdexec::let_value([[]](int k) { return fib(k); }) |
        stdexec::then([[auto i, auto j]] { return i + j; })
    );
    return work;
}
```
```cpp
int k = 30;

stdexec::sender auto fibonacci = 
    begin | stdexec::then([]() { return k; }) | 
    stdexec::let_value([](int k) { return fib(k); });

std::cout << "fibonacci built\n";

auto [i] = stdexec::sync_wait(std::move(fibonacci)).value();
std::cout << "fibonacci " << k << " = " << i << "\n";
```
fibonacci built
fibonacci 30 = 832040
fibonacci 30 = 832040
if (first == last) {
    return stdexec::just(U(std::move(init)));
}

auto nxt =
    stdexec::just(std::invoke(f, std::move(init), *first)) |
    stdexec::let_value([this,
                        first = first,
                        last = last,
                        f = f
                        ](U u) {
                        I i = first;
                        return (*this)(++i, last, u, f);
                });
return std::move(nxt);
auto v = std::ranges::iota_view{1, 10'000};

stdexec::sender auto work =
    begin
    | stdexec::let_value([[i = std::ranges::begin(v),
                         s = std::ranges::end(v)]]() {
      return fold_left(i, s, 0, [](int i, int j) { return i + j; });
    });

auto [i] = stdexec::sync_wait(std::move(work)).value();
work = 49995000
using any_node_sender = any_sender_of<stdexec::set_value_t(tree::NodePtr<int>),
  stdexec::set_stopped_t(),
  stdexec::set_error_t(std::exception_ptr)>;

auto search_tree(auto test,
  tree::NodePtr<int> tree,
  stdexec::scheduler auto sch,
  any_node_sender&& fail) -> any_node_sender {
  if (tree == nullptr) {
    return std::move(fail);
  }
  if (test(tree)) {
    return stdexec::just(tree);
  }
  return stdexec::on(sch, stdexec::just()) |
    stdexec::let_value([=, fail = std::move(fail)] mutable {
      return search_tree(
        test,
        tree->left(),
        sch,
        stdexec::on(sch, stdexec::just()) |
          stdexec::let_value(
            [=, fail = std::move(fail)] mutable {
              return search_tree(
                test, tree->right(), sch, std::move(fail));
          })
        )
      )
    })
  return fail;
tree::NodePtr<int> t;
for (auto i : std::ranges::views::iota(1, 10'000)) {
  tree::Tree<int>::insert(i, t);
}

auto test = [](tree::NodePtr<int> t) -> bool {
  return t ? t->data() == 500 : false;
};

auto fail = begin | stdexec::then([]() { return tree::NodePtr<int>{}; });

stdexec::sender auto work =
  begin | stdexec::let_value([]() {
    return search_tree(test, t, sch, std::move(fail));
  });

auto [n] = stdexec::sync_wait(std::move(work)).value();

std::cout << "work " << " = " << n->data() << 'n';
work = 500
DON'T DO THAT
CAN IS NOT SHOULD
WRITE AN ALGORITHM
WHY YOU MIGHT

- Throughput
- Interruptable
THANK YOU